The Ciao Prolog System

A Next Generation Logic Programming Environment
REFERENCE MANUAL
The Ciao System Documentation Series
Technical Report CLIP 3/97.1
Generated/Printed on: 5 May 2000
Version 1.5#133 (2000/5/5, 17:32:14 CEST)

F. Bueno
D. Cabeza
M. Carro
M. Hermenegildo
P. López
G. Puebla
clip@dia.fi.upm.es
http://www.clip.dia.fi.upm.es/
The CLIP Group
School of Computer Science
Technical University of Madrid

Table of Contents

Summary

Ciao is a *public domain*, *next generation* programming environment with a unique set of features:

- Ciao is a complete Prolog system, supporting *ISO-Prolog*, but its novel modular design allows both *restricting* and *extending* the language. As a result, it allows working with *fully declarative subsets* of Prolog and also to *extend* these subsets (or ISO-Prolog) both syntactically and semantically. Most importantly, these restrictions and extensions can be activated separately on each program module so that several extensions can coexist in the same application for different modules.
- Ciao extensions currently include feature terms (records), higher-order (with predicate abstractions), functions, constraints, objects, persistence, several control rules (breadth-first search, iterative deepening, ...), concurrency (threads/engines), a good base for distributed execution (agents), and parallel execution. Libraries also support WWW programming, sockets, external interfaces (C, Java, TclTk, relational databases, etc.), etc.
- **Ciao** offers support for *programming in the large* with a robust module/object system, module-based separate/incremental compilation (automatically –no need for makefiles), an assertion language for declaring (*optional*) program properties (including types and modes, but also determinacy, non-failure, cost, etc.), automatic static inference and static/dynamic checking of such assertions, etc.
- **Ciao** also offers support for *programming in the small* producing small executables (including only those builtins used by the program) and support for writing scripts in Prolog.
- The **Ciao** programming environment includes a classical top-level and a rich emacs interface with an embeddable source-level debugger and a number of execution visualization tools.
- The **Ciao** compiler (which can be run outside the top level shell) generates several forms of architecture-independent and stand-alone executables, which run with speed, efficiency and executable size which are very competive with other commercial and academic Prolog/CLP systems. Library modules can be compiled into compact bytecode or C source files, and linked statically, dynamically, or autoloaded.
- The novel modular design of Ciao enables, in addition to modular program development, effective global program analysis and static debugging and optimization via source to source program transformation. These tasks are performed by the **Ciao preprocessor** (`ciaopp`, distributed separately).
- The **Ciao** programming environment also includes `lpdoc`, an automatic documentation generator for LP/CLP programs. It processes Prolog files adorned with (**Ciao**) assertions and machine-readable comments and generates manuals in many formats including `postscript`, `pdf`, `texinfo`, `info`, `HTML`, `man`, etc. , as well as on-line help, ascii `README` files, entries for indices of manuals (`info`, WWW, ...), and maintains WWW distribution sites.

Ciao is distributed under the GNU General Public License.

This documentation corresponds to version 1.5#133 (2000/5/5, 17:32:14 CEST).

1 Introduction

1.1 About this manual

This is the *Reference Manual* for the Ciao Prolog development system. It contains basic information on how to install Ciao Prolog and how to write, debug, and run Ciao Prolog programs from the command line, from inside GNU `emacs`, or form a windowing desktop. It also documents all the libraries available in the standard distribution.

This manual has been generated using the *LPdoc* semi-automatic documentation generator for LP/CLP programs [HC97,Her99b]. `lpdoc` processes Prolog files (and files in other CLP languages) adorned with assertions and machine-readable comments, which should be written in the Ciao assertion language [PBH97,PBH00]. From these, it generates manuals in many formats including `postscript`, `pdf`, `texinfo`, `info`, `HTML`, `man`, etc., as well as on-line help, ascii `README` files, entries for indices of manuals (`info`, WWW, ...), and maintains WWW distribution sites.

The big advantage of this approach is that it is easier to keep the on-line and printed documentation in sync with the source code [Knu84]. As a result, *this manual changes continually as the source code is modified*. Because of this, the manual has a version number. You should make sure the manual you are reading, whether it be printed or on-line, coincides with the version of the software that you are using.

The approach also implies that there is often a variability in the degree to which different libraries or system components are documented. Many libraries offer abundant documentation, but a few will offer little. The latter is due to the fact that we tend to include libraries in the manual if the code is found to be useful, even if they may still contain sparse documentation. This is because including a library in the manual will at the bare minimum provide formal information (such as the names of exported predicates and their arity, which other modules it loads, etc.), create index entries, pointers for on-line help in the electronic versions of the manuals, and command-line completion capabilities inside `emacs`. Again, the manual is being updated continuously as the different libraries (and machine-readable documentation in them) are improved.

1.2 About the Ciao Prolog development system

The Ciao system is a full programming environment for developing programs in the Prolog language and in several other languages which are extensions and modifications of Prolog in several interesting and useful directions. The programming environment offers a number of tools such as the Ciao standalone compiler (`ciaoc`), a traditional-style top-level interactive shell (`ciaosh` or `ciao`), an interpreter of scripts written in Prolog (`ciao-shell`), a Prolog `emacs` mode (which greatly helps the task of developing programs with support for editing, debugging, version/change tracking, etc.), numerous libraries, a powerful program preprocessor (`ciaopp` [BdlBH99,BLGPH99,HBPLG99], which supports static debugging and optimization from program analysis via source to source program transformation), and an automatic documentation generator (`lpdoc`) [HC97,Her99b]. A number of execution visualization tools [CGH93,CH00d,CH00c] are also available.

This manual documents the first four of the tools mentioned above [see PART I - The program development environment], and the Ciao Prolog language and library [see PART II - The Ciao basic language (engine)]. `ciaopp` and `lpdoc` are documented in separate manuals.

The Ciao language has been designed from the ground up to be minimal, but also highly extensible in a modular way. The first objective allows producing small executables (including only those builtins used by the program), providing basic support for pure logic programming, and being able to write scripts in Prolog. The second one allows supporting standard ISO-Prolog [see PART III - ISO-Prolog library (iso)], as well as powerful extensions such as constraint logic programming, functional logic programming, and object-oriented logic programming [see PART VII - Ciao Prolog extensions].

The design of Ciao has also focused on allowing modular program development, as well as automatic program manipulation and optimization. Ciao includes a robust module system [CH99a], module-based automatic incremental compilation [CH99b], and modular global program analysis, debugging and optimization [PH99], based on a rich assertion language [see PART V - Annotated Prolog library (assertions)] for declaring (optional) program properties (including types and modes), which can be checked either statically or dynamically. The program analysis, static debugging and optimization tasks related to these assertions are performed by the `ciaopp` preprocessor, as mentioned above. These assertions (together with special comment-style declarations) are also the ones used by the `lpdoc` autodocumenter to generate documentation for programs (the comment-style declarations are documented in the `lpdoc` manual).

Ciao also includes several other features and utilities, such as support for several forms of executables, concurrency (threads), distributed and parallel execution, higher-order, WWW programming (PiLLoW [CHV96b]), interfaces to other languages like C and Java, database interfaces, graphical interfaces, etc., etc. [see PARTS VI to XI].

1.3 ISO-Prolog compliance versus extensibility

One of the innovative features of Ciao is that it has been designed to subsume *ISO-Prolog* (International Standard ISO/IEC 13211-1, PROLOG: Part 1–General Core [DEDC96]), while at the same time extending it in many important ways. The intention is to ensure that all ISO-compliant Prolog programs run correctly under Ciao. At the same time, the Ciao module system (see [PART II - The Ciao basic language (engine)] and [CH00b] for a discussion of the motivations behind the design) allows selectively avoiding the loading of most ISO-builtins (and changing some other ISO characteristics) when not needed, so that it is possible to work with purer subsets of Prolog and also to build small executables. Also, this module system makes it possible to develop extensions using these purer subsets (or even the full ISO-standard) as a starting point. Using these features, the Ciao distribution includes libraries which significantly extend the language both syntactically and semantically.

Compliance with ISO is still not complete: currently there are some minor deviations in, e.g., the treatment of characters, the syntax, some of the arithmetic functions, and part of the error system. On the other hand, Ciao has been reported by independent sources (members of the standarization body) to be one of the most conforming Prologs at the moment of this writing, and the first one to be able to compile all the standard-conforming test cases. Also, Ciao does not offer a strictly conforming mode which rejects uses of non-ISO features. However, in order to aid programmers who wish to write standard compliant programs, library predicates that correspond to those in the ISO-Prolog standard are marked specially in the manuals, and differences between the Ciao and the prescribed ISO-Prolog behaviours, if any, are commented appropriately.

The intention of the Ciao developers is to progressively complete the compliance of Ciao with the published parts of the ISO standard as well as with other reasonable extensions of the standard may be published in the future. However, since one of the design objectives of Ciao is to address some shortcomings of previous implementations of Prolog and logic programming in general, we also hope that some of the better ideas present in the system will make it eventually into the standards.

1.4 About the name of the System

After reading the previous sections the sharp reader may have already seen the logic behind the 'Ciao Prolog' name. Ciao is an interesting word which means both 'hello' and 'goodbye'.

Ciao Prolog intends to be a really good all-round freely available ISO-Prolog system which can be used as an *introduction* to classical Prolog, in both academic and industrial environments. But Ciao is also a new generation language/system which we believe points the way *beyond* classical Prolog, to building much better LP/CLP systems while keeping all of Prolog's advantages.

1.5 Syntax terminology and notational conventions

This manual is not meant to be an introduction to the Prolog language. The reader is referred to standard textbooks on Prolog such as [SS86,CM81,Apt97,Hog84]. However, we would like to refresh herein some concepts for the sake of establishing terminology. Also, we will briefly introduce a few of the extensions that Ciao brings to the Prolog language.

1.5.1 Predicates and their components

In Prolog, procedures are called *predicates* and predicate calls *literals*. They all have the classical syntax of procedures (and of logic predications and of mathematical functions). Predicates are identified in this manual by a keyword 'PREDICATE' at the right margin of the place where they are documented.

Prolog instructions are expressions made up of control constructs (Chapter 13 [Control constructs/predicates], page 79) and literals, and are called *goals*. Literals are also (atomic) goals.

A predicate definition is a sequence of clauses. A clause has the form `H :- B.` (ending in '`.`'), where `H` is syntactically the same as a literal and is called the clause *head*, and `B` is a goal and is called the clause *body*. A clause with no body is written `H.` and is called a *fact*. Clauses with body are also called *rules*. A Prolog program is a sequence of predicate definitions.

1.5.2 Characters and character strings

We adopt the following convention for delineating character strings in the text of this manual: when a string is being used as a Prolog atom it is written thus: `user` or `'user'`; but in all other circumstances double quotes are used (as in `"hello"`).

When referring to keyboard characters, printing characters are written thus: ⟨a⟩, while control characters are written like this: ⟨^A⟩. Thus ⟨^C⟩ is the character you get by holding down the ⟨CTL⟩ key while you type ⟨c⟩. Finally, the special control characters carriage-return, line-feed and space are often abbreviated to ⟨RET⟩, ⟨LFD⟩ and ⟨SPC⟩ respectively.

1.5.3 Predicate specs

Predicates in Prolog are distinguished by their name *and* their arity. We will call `name/arity` a *predicate spec*. The notation `name/arity` is therefore used when it is necessary to refer to a predicate unambiguously. For example, `concatenate/3` specifies the predicate which is named "concatenate" and which takes 3 arguments.

(Note that different predicates may have the same name and different arity. Conversely, of course, they may have the same arity and different name.)

1.5.4 Modes

When documenting a predicate, we will often describe its usage with a mode spec which has the form `name(Arg1, ..., ArgN)` where each `Arg` may be preceded by a *mode*. A mode is a functor which is wrapped around an argument (or prepended if defined as an operator). Such a mode allows documenting in a compact way the instantiation state on call and exit of the argument to which it is applied. The set of modes which can be used in Ciao is not fixed. Instead, arbitrary modes can be defined by in programs using the `modedef/1` declarations of the

Ciao *assertion language* (Chapter 54 [The Ciao assertion package], page 227 for details). Modes are identified in this manual by a keyword 'MODE'.

Herein, we will use the set of modes defined in the Ciao `isomodes` library, which is essentially the same as those used in the ISO-Prolog standard (Chapter 59 [ISO-Prolog modes], page 253).

1.5.5 Properties and types

Although Ciao Prolog is *not* a typed language, it allows writing (and using) types, as well as (more general) properties. There may be properties of the states and of the computation. Properties of the states allow expressing characteristics of the program variables during computation, like in `sorted(X)` (`X` is a sorted list). Properties of the computation allow expressing characteristics of a whole computation, like in `is_det(p(X,Y))` (such calls yield only one solution). Properties are just a special form of predicates (Chapter 56 [Declaring regular types], page 241) and are identified in this manual by a keyword 'PROPERTY'.

Ciao types are *regular types* (Chapter 56 [Declaring regular types], page 241), which are a special form of properties themselves. They are identified in this manual by a keyword 'REGTYPE'.

1.5.6 Declarations

A *declaration* provides information to one of the Ciao environment tools. Declarations are interspersed in the code of a program. Usually the target tool is either the compiler (telling it that a predicate is dynamic, or a meta-predicate, etc.), the preprocessor (which understands declarations of properties and types, assertions, etc.), or the autodocumenter (which understands the previous declarations and also certain "comment" declarations).

A declaration has the form `:- D.` where `D` is syntactically the same as a literal. Declarations are identified in this manual by a keyword 'DECLARATION'.

In Ciao users can define (and document) new declarations. New declarations are typically useful when defining extensions to the language (which in Ciao are called packages). Such extensions are often implemented as expansions (see Chapter 27 [Extending the syntax], page 145). There are many such extensions in Ciao. The `functions` library, which provides fuctional syntax, is an example. The fact that in Ciao expansions are local to modules (as operators, see below) makes it possible to use a certain language extension in one module without affecting other modules.

1.5.7 Operators

An *operator* is a functor (or predicate name) which has been declared as such, thus allowing its use in a prefix, infix, or suffix fashion, instead of the standard procedure-like fashion. E.g., declaring `+` as an infix operator allows writing `X+Y` instead of `'+'(X,Y)` (which may still, of course, be written).

Operators in Ciao are local to the module/file where they are declared. However, some operators are standard and allowed in every program (see Chapter 32 [Standard operators], page 163). This manual documents the operator declarations in each (library) module where they are included. As with expansions, the fact that in Ciao operators are local to modules makes it possible to use a certain language extension in one module without affecting other modules.

1.6 A tour of the manual

The rest of the introductory chapters after this one provide a first "getting started" introduction for newcomers to the Ciao system. The rest of the chapters in the manual are organized into a sequence of major parts as follows:

1.6.1 PART I - The program development environment

This part documents the components of the basic Ciao program development environment. They include:

ciaoc: the standalone compiler, which creates executables without having to enter the interactive top-level.

ciaosh: (also invoked simply as `ciao`) is an interactive top-level shell, similar to the one found on most Prolog systems (with some enhancements).

debugger.pl:
a Byrd box-type debugger, similar to the one found on most Prolog systems (also with some enhancements, such as source-level debugging). This is not a standalone application, but is rather included in `ciaosh`, as is done in other Prolog systems. However, it is also *embeddable*, in the sense that it can be included as a library in executables, and activated dynamically and conditionally while such executables are running.

ciao-shell: an interpreter/compiler for *Prolog scripts* (i.e., files containing Prolog code which run without needing explicit compilation).

ciao.el: a *complete program development enviroment*, based on GNU emacs, with syntax coloring, direct access to all the tools described above (as well as the preprocessor and the documenter), atomatic location of errors, source-level debugging, context-sensitive access to on-line help/manuals, etc. The use of this environment is *very highly recommended*!

The Ciao program development environment also includes `ciaopp`, the preprocessor, and `lpdoc`, the documentation generator, which are described in separate manuals.

1.6.2 PART II - The Ciao basic language (engine)

This part documents the *Ciao basic builtins*. These predefined predicates and declarations are available in every program, unless the `pure` package is used (by using a `:- module(_,_,[pure]).` declaration or `:- use_package(pure).`). These predicates are contained in the `engine` directory within the `lib` library. The rest of the library predicates, including the packages that provide most of the ISO-Prolog builtins, are documented in subsequent parts.

1.6.3 PART III - ISO-Prolog library (iso)

This part documents the *iso* package which provides to Ciao programs (most of) the ISO-Prolog functionality , including the *ISO-Prolog builtins* not covered by the basic library. All these predicates are loaded by default in user files and in modules which use standard module declarations such as:

`:- module(`*modulename*`,`*exports*`).`

which are equivalent to:

`:- module(`*modulename*`,`*exports*`,[iso]).`

or

`:- module(`*modulename*`,`*exports*`).`

`:- use_package([iso]).`

If you do not want these ISO builtins loaded for a given file (in order to make the executable smaller) you can ask for this explicitly using:

`:- module(`*modulename*`,`*exports*`,[]).`

or

`:- module(`*modulename*`,`*exports*`).`

`:- use_package([]).`

See the description of the declarations for declaring modules and using other modules, and the documentation of the `iso` library for details.

1.6.4 PART IV - Classic Prolog library (classic)

This part documents some Ciao libraries which provide additional predicates and functionalities that, despite not being in the ISO standard, are present in many popular Prolog systems. This includes definite clause grammars (DCGs), "Quintus-style" internal database, list processing predicates, dictionaries, string processing, DEC-10 Prolog-style input/output, formatted output, dynamic loading of modules, activation of operators at run-time, etc.

1.6.5 PART V - Annotated Prolog library (assertions)

Ciao allows *annotating* the program code with *assertions*. Such assertions include type and instantiation mode declarations, but also more general properties as well as comments in the style of the *literate programming*. These assertions document predicates (and modules and whole applications) and can be used by the Ciao preprocessor/compiler while debugging and optimizing the program or library, and by the Ciao documenter to build the program or library reference manual.

1.6.6 PART VI - Ciao Prolog library miscellanea

This part documents several Ciao libraries which provide different useful additional functionalities. Such functionalities include performing operating system calls, gathering statistics from the Prolog engine, file and file name manipulation, error and exception handling, fast reading and writing of terms (marshalling and unmarshalling), file locking, program reporting messages, pretty-printing programs and assertions, a browser of the system libraries, additional expansion utilities, concurrent aggregates, graph visualization, etc.

1.6.7 PART VII - Ciao Prolog extensions

The libraries documented in this part extend the Ciao language in several different ways. The extensions include:

- pure Prolog programming (well, this can be viewed more as a restriction than an extension);
- feature terms or *records* (i.e., structures with names for each field);
- parallel programming (e.g., &-Prolog style);
- functional syntax;
- higher-order library;
- global variables;
- `setarg` and `undo`;
- delaying predicate execution;
- active modules;
- breadth-first execution;
- iterative deepening-based execution;
- constraint logic programming;
- object oriented programming.

1.6.8 PART VIII - Interfaces to other languages and systems

The following interfaces to/from Ciao Prolog are documented in this part:

- External interface (e.g., to C).
- Socket interface.
- Tcl/tk interface.
- Web interface (http, html, xml, etc.);
- Persistent predicate databases (interface between the Prolog internal database and the external file system).
- SQL-like database interface (interface between the Prolog internal database and external SQL/ODBC systems).
- Java interface.
- Calling emacs from Prolog.

1.6.9 PART IX - Abstract data types

This part includes libraries which implement some generic data structures (abstract data types) that are used frequently in programs or in the Ciao system itself.

1.6.10 PART X - Miscellaneous standalone utilities

This is the documentation for a set of miscellaneous standalone utilities contained in the `etc` directory of the Ciao distribution.

1.6.11 PART XI - Contributed libraries

This part includes a number of libraries which have contributed by users of the Ciao system. Over time, some of these libraries are moved to the main library directories of the system.

1.6.12 PART XII - Appendices

These appendices describe the installation of the Ciao environment on different systems and some other issues such as reporting bugs, signing up on the Ciao user's mailing list, downloading new versions, limitations, etc.

1.7 Acknowledgments

The Ciao system is a joint effort on one side of the present (*Francisco Bueno*, *Daniel Cabeza*, *Manuel Carro*, *Manuel Hermenegildo*, *Pedro López*, and *Germán Puebla*) and past (*María José García de la Banda*) members of the *CLIP group* at the School of Computer Science, *Technical University of Madrid* , and on the other side of several colleagues and students that have collaborated with us over the years of its development. The following is an (inevitably incomplete) list of those that have contributed to the development of Ciao:

- The *Ciao engine*, compiler, libraries and documentation, although completely rewritten at this point, have their origins in the *&-Prolog parallel Prolog* engine and *parallelizing compiler*, developed by *Manuel Hermenegildo*, *Kevin Greene*, *Kalyan Muthukumar*, and *Roger Nasr* at *MCC* and later at *UPM*. The &-Prolog engine and low-level (*WAM*) compilers in turn were derived from early versions (0.5 to 0.7) of *SICStus Prolog* [Car88]. SICStus is an excellent, high performance Prolog system, developed by *Mats Carlsson* and colleagues at the *Swedish Institute of Computer Science* (*SICS*), that every user of Prolog should check

out [Swe95,AAF91]. Very special thanks are due to *Seif Haridi*, *Mats Carlsson*, and colleagues at SICS for allowing the SICStus 0.5-0.7 components in *&-Prolog* and its successor, Ciao, to be distributed freely. Parts of the parallel abstract machine have been developed in collaboration with *Gopal Gupta* and *Enrico Pontelli* (*New Mexico State University*).

- Many aspects of the analyzers in the *Ciao preprocessor* (`ciaopp`) have been developed in collaboration with *Peter Stuckey* (*Melbourne U.*), *Kim Marriott* (*Monash U.*), *Maurice Bruynooghe*, *Gerda Janssens*, *Anne Mulkers*, and *Veroniek Dumortier* (*K.U. Leuven*), and *Saumya Debray* (*U. of Arizona*). The assertion system has been developed in collaboration with *Jan Maluzynski* and *Wlodek Drabent* (*Linkoping U.*) and *Pierre Deransart* (*INRIA*). The core of type inference system derives from the system developed by *John Gallagher* [GdW94] (*Bristol University*) and later adapted to CLP(FD) by *Pawel Pietrzak* (*Linkoping U.*).
- The constraint solvers for *R* and *Q* are derived from the code developed by *Christian Holzbauer* (*Austrian Research Institute for AI* in Vienna) [Hol94,Hol90,Hol92].
- The Ciao manuals include material from the *DECsystem-10 Prolog User's Manual* by *D.L. Bowen* (editor), *L. Byrd*, *F.C.N. Pereira*, *L.M. Pereira*, and *D.H.D. Warren* [BBP81]. They also contain material from the SICStus Prolog user manuals for SICStus versions 0.5-0.7 by *Mats Carlsson* and *Johan Widen* [Car88], as well as from the Prolog ISO standard documentation [DEDC96].
- Ciao is designed to be highly extendable in a modular way. Many of the libraries distributed with Ciao have been developed by other people all of which is impossible to mention here. Individual author names are included in the documentation of each library and appear in the indices.
- The development of the Ciao system has been supported in part by ESPRIT projects ACCLAIM, PARFORCE, and DISCIPL, and by CICYT/MEC projects ELLA and EDIPIA.

If you feel you have contributed to the development of Ciao and we have forgotten adding your name to this list or the acknowledgements given in the different chapters, please let us know and we will be glad to give proper credits.

1.8 Version/Change Log (`ciao`)

Version 1.5 (1999/11/29, 16:16:23 MEST)

Development version following even 1.4 distribution.

Major changes so far (which will be in 1.6):

- Source-level debugger in emacs, breakpts.
- Debugger embeddable in executables.
- Greatly improved emacs interface.
- Menu-based documenter interface.
- Threads now available in Win32.
- Many improvements to threads.
- Modular clp(R) / clp(Q).
- Improved syntax for pred. abstractions.
- Library of higher order list predicates.
- Better code expansion facilities (macros).
- Compressed object code/executables.
- The size of atoms is now unbound.
- Fast creation of new unique atoms.
- # of clauses/pred. essentially unbound.

- Delayed goals with freeze restored.
- Faster compilation and execution.
- Much faster fast write/read.
- Improved documentation.
- Other new libraries.
- Improved installation/deinstallation.
- Many improvements to autodocumenter.
- Many minor bug fixes.

Version 1.4 (1999/11/27, 19:0:0 MEST)
- Documentation greatly improved.
- Automatic (re)compilation of foreign files.
- Concurrency primitives revamped; restored &Prolog-like multiengine capability.
- Windows installation and overall operation greatly improved.
- New version of O'Ciao class/object library, with improved performance.
- Added support for "predicate abstractions" in call/N.
- Implemented reexportation through reexport declarations.
- Changed precedence of importations, last one is now higher.
- Modules can now implicitly export all predicates.
- Many minor bugs fixed.

Version 1.3 (1999/6/16, 17:5:58 MEST)
Development version following even 1.2 distribution.

Version 1.2 (1999/6/14, 16:54:55 MEST)
Temporary version distributed locally for extensive testing of reexportation and other 1.3 features.

Version 1.1 (1999/6/4, 13:30:37 MEST)
Development version following even 1.0 distribution.

Version 1.0 (1999/6/4, 13:27:42 MEST)
- Added Tcl/Tk interface library to distribution.
- Added push_prolog_flag/2 and pop_prolog_flag/1 declarations/builtins.
- Filename processing in Windows improved.
- Added redefining/1 declaration to avoid redefining warnings.
- Changed syntax/1 declaration to use_package/1.
- Added add_clause_trans/1 declaration.
- Changed format of .itf files such that a '+' stands for all the standard imports from engine, which are included in c_itf source internally (from engine(builtin_exports)). Further changes in itf data handling, so that once an .itf file is read in a session, the file is cached and next time it is needed no access to the file system is required.
- Many bugs fixed.

Version 0.9 (1999/3/10, 17:3:49 CET)
- Test version before 1.0 release. Many bugs fixed.

Version 0.8 (1998/10/27, 13:12:36 MET)
- Changed compiler so that only one pass is done, eliminated `.dep` files.
- New concurrency primitives.

- Changed assertion comment operator to #.
- Implemented high-order with call/N.
- Integrated SQL-interface to external databases with persistent predicate concept.
- First implementation of object oriented programming package.
- Some bugs fixed.

Version 0.7 (1998/9/15, 12:12:33 MEST)
- Improved debugger capabilities and made easier to use.
- Simplified assertion format.
- New arithmetic functions added, which complete all ISO functions.
- Some bugs fixed.

Version 0.6 (1998/7/16, 21:12:7 MET DST)
- Defining other path aliases (in addition to 'library') which can be loaded dynamically in executables is now possible.
- Added the posibility to define multifile predicates in the shell.
- Added the posibility to define dynamic predicates dynamically.
- Added addmodule meta-argument type.
- Implemented persistent data predicates.
- New version of PiLLoW WWW library (XML, templates, etc.).
- Ported active modules from "distributed Ciao" (independent development version of Ciao).
- Implemented lazy loading in executables.
- Modularized engine(builtin).
- Some bugs fixed.

Version 0.5 (1998/3/23)
- First Windows version.
- Integrated debugger in toplevel.
- Implemented DCG's as (Ciao-style) expansions.
- Builtins renamed to match ISO-Prolog.
- Made ISO the default syntax/package.

Version 0.4 (1998/2/24)
- First version with the new Ciao emacs mode.
- Full integration of concurrent engine and compiler/library.
- Added new_declaration/1 directive.
- Added modular syntax enhancements.
- Shell script interpreter separated from toplevel shell.
- Added new compilation warnings.

Version 0.3 (1997/8/20)
- Ciao builtins modularized.
- New prolog flags can be defined by libraries.
- Standalone comand-line compiler available, with automatic "make".
- Added assertions and regular types.
- First version using the automatic documentation generator.

Version 0.2 (1997/4/16)

- First module system implemented.
- Implemented exceptions using catch/3 and throw/1.
- Added functional & record syntax.
- Added modular sentence, term, and goal translations.
- Implemented attributed variables.
- First CLPQ/CLPR implementation.
- Added the posibility of linking external .so files.
- Changes in syntax to allow `P(X)` and `"string"||L`.
- Changed to be more similar to ISO-Prolog.
- Implemented Prolog shell scripts.
- Implemented data predicates.

Version 0.1 (1997/2/13)
First fully integrated, standalone Ciao distribution. Based on integrating into an evolution of the &-Prolog engine/libraries/preprocessor [Her86,HG91] many functionalities from several previous independent development versions of Ciao [HC93,HC94,HCC95,Bue95,CLI95,HBGP95,HBC96,CHV96b,HBC99].

2 Getting started on Un*x-like machines

Author(s): M.Hermenegildo.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

This part guides you through some very basic first steps with Ciao on a Un*x system. It assumes that Ciao is already installed correctly on your Windows system. If this is not the case, then follow the instructions in Chapter 135 [Installing Ciao from the source distribution], page 509 first.

We start with by describing the basics of using Ciao from a normal command shell such as `sh/bash`, `csh/tcsh`, etc. We strongly recommend reading also Section 2.3 [An introduction to the Ciao emacs environment (Un*x)], page 18 for the basics on using Ciao under `emacs`, which is a much simpler and much more powerful way of developing Ciao programs, and has the advantage of offering an almost identical environment under Windows and Un*x.

2.1 Testing your Ciao Un*x installation

It is a good idea to start by performing some tests to check that Ciao is installed correctly on your system (these are the same tests that you are instructed to do during installation, so you can obviously skip them if you have done them already at that time). If any of these tests do not succeed either your environment variables are not set properly (see Section 2.1.1 [Un*x user setup], page 15 for how to fix this):

- Typing `ciao` (or `ciaosh`) should start the typical Prolog top-level shell.
- In the top-level shell, Prolog library modules should load correctly. Type for example `use_module(library(dec10_io))` –you should get back a prompt with no errors reported.
- To exit the top level shell, type `halt.` as usual, or ⟨^D⟩.
- Typing `ciaoc` should produce the help message from the Ciao standalone compiler.
- Typing `ciao-shell` should produce a message saying that no code was found. This is a Ciao application which can be used to write scripts written in Prolog, i.e., files which do not need any explicit compilation to be run.

Also, the following documentation-related actions should work:

- If the `info` program is installed, typing `info` should produce a list of manuals which *should include Ciao manual(s) in a separate area* (you may need to log out and back in so that your shell variables are reinitialized for this to work).
- Opening with a WWW browser (e.g., `netscape`) the directory or `URL` corresponding to the `DOCROOT` setting should show a series of Ciao-related manuals. Note that *style sheets* should be activated for correct formatting of the manual.
- Typing `man ciao` should produce a man page with some very basic general information on Ciao (and pointing to the on-line manuals).
- The `DOCROOT` directory should contains the manual also in the other formats such as `postscript` or `pdf` which specially useful for printing. See Section 2.2.7 [Printing manuals (Un*x)], page 17 for instructions.

2.1.1 Un*x user setup

If the tests above have succeeded, the system is probably installed correctly and your environment variables have been set already. In that case you can skip to the next section.

Otherwise, if you have not already done so, make the following modifications in your startup scripts, so that these files are used (`<LIBROOT>` must be replaced with the appropriate value, i.e., where the Ciao library is installed):

- For users a *csh-compatible shell* (`csh`, `tcsh`, ...), add to `~/.cshrc`:

```
if ( -e <LIBROOT>/ciao/DOTcshrc ) then
   source <LIBROOT>/ciao/DOTcshrc
endif
```

- For users of an *sh-compatible shell* (`sh`, `bash`, ...), add to `~/.profile`:

```
if [ -f <LIBROOT>/ciao/DOTprofile ]; then
   . <LIBROOT>/ciao/DOTprofile
fi
```

  This will set up things so that the Ciao executables are found and you can access the Ciao system manuals using the `info` command. Note that, depending on your shell, *you may have to log out and back in* for the changes to take effect.

- Also, if you use `emacs` (highly recommended) add this line to your `~/.emacs` file:

```
(load-file "<LIBROOT>/ciao/DOTemacs.el")
```

If after following these steps things do not work properly, then the installation was probably not completed properly and you may want to try reinstalling the system.

2.2 Using Ciao from a Un*x command shell

2.2.1 Starting/exiting the top-level shell (Un*x)

The basic methods for starting/exiting the top-level shell have been discussed above. If upon typing `ciao` you get a "command not found" error or you get a longer message from Ciao before starting, it means that either Ciao was not installed correctly or you environment variables are not set up properly. Follow the instructions on the message printed by Ciao or refer to the installation instructions regarding user-setup for details.

2.2.2 Getting help (Un*x)

The basic methods for accessing the manual on-line have also been discussed above. Use the table of contents and the indices of *predicates*, *libraries*, *concepts*, etc. to find what you are looking for. Context-sensitive help is available within the `emacs` environment (see below).

2.2.3 Compiling and running programs (Un*x)

Once the shell is started, you can compile and execute Prolog modules inside the interactive top-level shell in the standard way. E.g., type `use_module(`*file*`).`, `use_module(library(`*file*`)).` for library modules, `ensure_loaded(`*file*`).` for files which are not modules, and `use_package(`*file*`).` for library packages (these are syntactic/semantic packages that extend the Ciao Prolog language in many different ways). Note that the use of `compile/1` and `consult/1` is discouraged in Ciao.

For example, you may want to type `use_package(iso)` to ensure Ciao has loaded all the ISO builtins (whether this is done by default or not depends on your `.ciaorc` file). Do not worry about any "module already in executable" messages –these are normal and simply mean that a certain module is already pre-loaded in the top-level shell. At this point, typing `write(hello).` should work.

Note that some predicates that may be built-ins in other Prologs are available through libraries in Ciao. This facilitates making small executables.

To change the working directory to, say, the `tests` directory in the Ciao source directory, first do:

```
?- use_module(library(system)).
```

(loading the `system` library makes a number of system-related predicates such as `cd/1` accessible) and then:

```
?- cd('$/tests').
```

(in Ciao the sequence `$/` *at the beginning of a path name* is replaced by the path of the Ciao root directory).

For more information see Chapter 5 [The interactive top-level shell], page 31.

2.2.4 Generating executables (Un*x)

Executables can be generated from the top-level shell (using `make_exec/2`) or using the standalone compiler (`ciaoc`). To be able to make an executable, the file should define the predicate `main/1` (or `main/0`), which will be called upon startup (see the corresponding manual section for details). In its simplest use, given a top-level *foo*`.pl` file for an application, the compilation process produces an executable `foo`, automatically detecting which other files used by `foo.pl` need recompilation.

For example, within the `tests` directory, you can type:

```
?- make_exec(hw,_).
```

which should produce an executable. Double-clicking on this executable should execute it.

Another way of creating Ciao executables from source files is by right-clicking on `.pl` files and choosing "make executable". This uses the standalone compiler (this has the disadvantage, however, that it is sometimes difficult to see the error messages).

For more information see Chapter 5 [The interactive top-level shell], page 31 and Chapter 4 [The stand-alone command-line compiler], page 25.

2.2.5 Running Ciao scripts (Un*x)

Ciao allows writing Prolog scripts. These are files containing Prolog source but which get executed without having to explicitly compile them (in the same way as, e.g., `.bat` files or programs in scripting languages). As an example, you can run the file `hw` in the `tests` directory of the Ciao distribution and look at the source with an editor. You can try changing the `Hello world` message and running the program again (no need to recompile!).

As you can see, the file should define the predicate `main/1` (not `main/0`), which will be called upon startup. The two header lines are necessary in Un*x in. In Windows you can leave them in or you can take them out, but you need to rename the script to `hw.pls`. Leaving the lines in has the advantage that the script will also work in Un*x without any change.

For more information see Chapter 8 [The script interpreter], page 53.

2.2.6 The Ciao initialization file (Un*x)

The Ciao toplevel can be made to execute upon startup a number of commands (such as, e.g., loading certain files or setting certain Prolog flags) contained in an initialization file. This file should be called `.ciaorc` and placed in your *home* directory (e.g., `~`, the same in which the `.emacs` file is put). You may need to set the environment variable `HOME` to the path of this directory for the Ciao toplevel shell to be able to locate this file on startup.

2.2.7 Printing manuals (Un*x)

As mentioned before, the manual is available in several formats in the `reference` directory within the `doc` directory in the Ciao distribution, including `postscript` or `pdf`, which are specially useful for printing. These files are also available in the `DOCROOT` directory specified during installation. Printing can be done using an application such as `ghostview` (freely available from `http://www.cs.wisc.edu/~ghost/index.html`) or `acrobat reader` (`http://www.adobe.com`, only `pdf`).

2.3 An introduction to the Ciao emacs environment (Un*x)

While it is easy to use Ciao with any editor of your choice, using it within the `emacs` editor/program development system is highly recommended: Ciao includes an `emacs` *mode* which provides a very complete *application development environment* which greatly simplifies many program development tasks. See Chapter 10 [Using Ciao inside GNU emacs], page 57 for details on the capabilities of `ciao/ emacs` combination.

If the (freely available) `emacs` editor/environment is not installed in your system, we highly recommend that you also install it at this point (there are instructions for where to find `emacs` and how to install it in the Ciao installation instructions). After having done this you can try for example the following things:

- A few basic things:
  - Typing ⟨^H⟩ ⟨i⟩ (or in the menus `Help->Manuals->Browse Manuals with Info`) should open a list of manuals in info format in which the Ciao manual(s) should appear.
  - When opening a Prolog file, i.e., a file with `.pl` or `.pls` ending, using ⟨^X⟩⟨^F⟩`filename` (or using the menus) the code should appear highlighted according to syntax (e.g., comments in red), and `Ciao/Prolog` menus should appear in the menu bar on top of the `emacs` window.
  - Loading the file using the `Ciao/Prolog` menu (or typing ⟨^C⟩ ⟨l⟩) should start in another emacs buffer the Ciao toplevel shell and load the file. You should now be able to switch the the toplevel shell and make queries from within `emacs`.

  Note: when using `emacs` it is *very convenient* to swap the locations of the (normally not very useful) ⟨Caps Lock⟩ key and the (very useful in `emacs`) ⟨Ctrl⟩ key on the keyboard. How to do this is explained in the `emacs` frequently asked questions FAQs (see the `emacs` download instructions for their location).

  (if these things do not work the system or emacs may not be installed properly).
- You can go to the location of most of the errors that may be reported during compilation by typing ⟨^C⟩ ⟨`⟩.
- You can also, e.g., create executables from the `Ciao/Prolog` menu, as well as compile individual files, or generate active modules.
- Loading a file for source-level debugging using the `Ciao/Prolog` menu (or typing ⟨^C⟩ ⟨d⟩) and then issuing a query should start the source-level debugger and move a marker on the code in a window while execution is stepped through in the window running the Ciao top level.
- You can add the lines needed in Un*x for turning any file defining `main/1` into a script from the Ciao/Prolog menu or by typing ⟨^C⟩ ⟨I⟩ ⟨S⟩.
- You can also work with the preprocessor and auto-documenter directly from emacs: see their manuals or browse through the corresponding menus that appear when editing `.pl` files.

We encourage you once more to read Chapter 10 [Using Ciao inside GNU emacs], page 57 to discover the many other functionalities of this environment.

2.4 Keeping up to date (Un*x)

You may want to read Chapter 137 [Beyond installation], page 521 for instructions on how to sign up on the Ciao user's mailing list, receive announcements regarding new versions, download new versions, report bugs, etc.

3 Getting started on Windows machines

Author(s): M.Hermenegildo.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

This part guides you through some very basic first steps with Ciao on an MSWindows ("Win32") system. It assumes that Ciao is already installed correctly on your Windows system. If this is not the case, then follow the instructions in Chapter 136 [Installing Ciao from a Win32 binary distribution], page 519 (or Chapter 135 [Installing Ciao from the source distribution], page 509) first.

We start with by describing the basics of using Ciao from the Windows explorer and/or a DOS command shell. We strongly recommend reading also Section 3.3 [An introduction to the Ciao emacs environment (Win32)], page 21 for the basics on using Ciao under `emacs`, which is a much simpler and much more powerful way of developing Ciao programs, and has the advantage of offering an almost identical environment under Windows and Un*x.

3.1 Testing your Ciao Win32 installation

It is a good idea to start by performing some tests to check that Ciao is installed correctly on your system (these are the same tests that you are instructed to do during installation, so you can obviously skip them if you have done them already at that time):

- Ciao-related file types (`.pl` source files, `.cpx` executables, `.itf,.po,.asr` interface files, `.pls` scripts, etc.) should have specific icons associated with them (you can look at the files in the folders in the Ciao distribution to check).
- Double-clicking on the shortcut to `ciaosh`(`.cpx`) on the desktop should start the typical Prolog top-level shell in a window. If this shortcut has not been created on the desktop, then double-clicking on the `ciaosh`(`.cpx`) icon inside the `shell` folder within the Ciao source folder should have the same effect.
- In the top-level shell, Prolog library modules should load correctly. Type for example `use_module(library(dec10_io))` at the Ciao top-level prompt –you should get back a prompt with no errors reported.
- To exit the top level shell, type `halt.` as usual, or ⟨^Z⟩.

Also, the following documentation-related actions should work:

- Double-clicking on the shortcut to `ciao.html` which appears on the desktop should show the Ciao manual in your default WWW browser. If this shortcut has not been created you can double-click on the `ciao.html` file in the `doc\reference\ciao_html` folder inside the Ciao source folder. Make sure you configure your browser to use *style sheets* for correct formatting of the manual (note, however, that some older versions of Explorer did not support style sheets well and will give better results turning them off).
- The `doc\reference` folder contains the manual also in the other formats present in the distribution, such as `info` (very convenient for users of the `emacs` editor/program development system) and `postscript` or `pdf`, which are specially useful for printing. See Section 3.2.7 [Printing manuals (Win32)], page 21 for instructions.

3.2 Using Ciao from the Windows explorer and command shell

3.2.1 Starting/exiting the top-level shell (Win32)

The basic methods for starting/exiting the top-level shell have been discussed above. The installation script also leaves a `ciaosh`(`.bat`) file inside the `shell` folder of the Ciao distribution which can be used to start the top-level shell from the command line in Windows95/98/NT.

3.2.2 Getting help (Win32)

The basic methods for accessing the manual on-line have also been discussed above. Use the table of contents and the indices of *predicates*, *libraries*, *concepts*, etc. to find what you are looking for. Context-sensitive help is available within the `emacs` environment (see below).

3.2.3 Compiling and running programs (Win32)

Once the shell is started, you can compile and execute Prolog modules inside the interactive toplevel shell in the standard way. E.g., type `use_module(`*file*`).`, `use_module(library(`*file*`)).` for library modules, `ensure_loaded(`*file*`).` for files which are not modules, and `use_package(`*file*`).` for library packages (these are syntactic/semantic packages that extend the Ciao Prolog language in many different ways). Note that the use of `compile/1` and `consult/1` is discouraged in Ciao.

For example, you may want to type `use_package(iso)` to ensure Ciao has loaded all the ISO builtins (whether this is done by default or not depends on your `.ciaorc` file). Do not worry about any "module already in executable" messages –these are normal and simply mean that a certain module is already pre-loaded in the toplevel shell. At this point, typing `write(hello).` should work.

Note that some predicates that may be built-ins in other Prologs are available through libraries in Ciao. This facilitates making small executables.

To change the working directory to, say, the `tests` directory in the Ciao source directory, first do:

```
?- use_module(library(system)).
```

(loading the `system` library makes a number of system-related predicates such as `cd/1` accessible) and then:

```
?- cd('$/tests').
```

(in Ciao the sequence `$/` *at the beginning of a path name* is replaced by the path of the Ciao root directory).

For more information see Chapter 5 [The interactive top-level shell], page 31.

3.2.4 Generating executables (Win32)

Executables can be generated from the toplevel shell (using `make_exec/2`) or using the standalone compiler (`ciaoc(.cpx`), located in the `ciaoc` folder). To be able to make an executable, the file should define the predicate `main/1` (or `main/0`), which will be called upon startup (see the corresponding manual section for details).

For example, within the `tests` directory, you can type:

```
?- make_exec(hw,_).
```

which should produce an executable. Double-clicking on this executable should execute it.

Another way of creating Ciao executables from source files is by right-clicking on `.pl` files and choosing "make executable". This uses the standalone compiler (this has the disadvantage, however, that it is sometimes difficult to see the error messages).

For more information see Chapter 5 [The interactive top-level shell], page 31 and Chapter 4 [The stand-alone command-line compiler], page 25.

3.2.5 Running Ciao scripts (Win32)

Double-clicking on files ending in `.pls`, *Ciao Prolog scripts*, will also execute them. These are files containing Prolog source but which get executed without having to explicitly compile them (in the same way as, e.g., `.bat` files or programs in scripting languages). As an example, you can double-click on the file `hw.pls` in the `tests` folder and look at the source with an editor. You can try changing the `Hello world` message and double-clicking again (no need to recompile!).

As you can see, the file should define the predicate `main/1` (not `main/0`), which will be called upon startup. The two header lines are only necessary in Un*x. In Windows you can leave them in or you can take them out, but leaving them in has the advantage that the script will also work in Un*x without any change.

For more information see Chapter 8 [The script interpreter], page 53.

3.2.6 The Ciao initialization file (Win32)

The Ciao toplevel can be made to execute upon startup a number of commands (such as, e.g., loading certain files or setting certain Prolog flags) contained in an initialization file. This file should be called `.ciaorc` and placed in your *home* folder (e.g., the same in which the `.emacs` file is put). You may need to set the environment variable `HOME` to the path of this folder for the Ciao toplevel shell to be able to locate this file on startup.

3.2.7 Printing manuals (Win32)

As mentioned before, the manual is available in several formats in the `reference` folder within Ciao's `doc` folder, including `postscript` or `pdf`, which are specially useful for printing. This can be done using an application such as `ghostview` (freely available from `http://www.cs.wisc.edu/~ghost/index.html`) or `acrobat reader` (`http://www.adobe.com`, only `pdf`).

3.3 An introduction to the Ciao emacs environment (Win32)

While it is easy to use Ciao with any editor of your choice, using it within the `emacs` editor/program development system is highly recommended: Ciao includes an `emacs` *mode* which provides a very complete *application development environment* which greatly simplifies many program development tasks. See Chapter 10 [Using Ciao inside GNU emacs], page 57 for details on the capabilities of `ciao/ emacs` combination.

If the (freely available) `emacs` editor/environment is not installed in your system, we highly recommend that you also install it at this point (there are instructions for where to find `emacs` and how to install it in the Ciao installation instructions). After having done this you can try for example the following things:

- A few basic things:
  - Typing ⟨^H⟩ ⟨i⟩ (or in the menus `Help->Manuals->Browse Manuals with Info`) should open a list of manuals in info format in which the Ciao manual(s) should appear.
  - When opening a Prolog file, i.e., a file with `.pl` or `.pls` ending, using ⟨^X⟩⟨^F⟩`filename` (or using the menus) the code should appear highlighted according to syntax (e.g., comments in red), and `Ciao/Prolog` menus should appear in the menu bar on top of the `emacs` window.
  - Loading the file using the `Ciao/Prolog` menu (or typing ⟨^C⟩ ⟨l⟩) should start in another emacs buffer the Ciao toplevel shell and load the file. You should now be able to switch the the toplevel shell and make queries from within `emacs`.

Note: when using `emacs` it is *very convenient* to swap the locations of the (normally not very useful) ⟨Caps Lock⟩ key and the (very useful in `emacs`) ⟨Ctrl⟩ key on the keyboard. How to do this is explained in the `emacs` frequently asked questions FAQs (see the `emacs` download instructions for their location).

(if these things do not work the system or emacs may not be installed properly).

- You can go to the location of most of the errors that may be reported during compilation by typing ⟨^C⟩ ⟨`⟩.
- You can also, e.g., create executables from the `Ciao/Prolog` menu, as well as compile individual files, or generate active modules.
- Loading a file for source-level debugging using the `Ciao/Prolog` menu (or typing ⟨^C⟩ ⟨d⟩) and then issuing a query should start the source-level debugger and move a marker on the code in a window while execution is stepped through in the window running the Ciao top level.
- You can add the lines needed in Un*x for turning any file defining `main/1` into a script from the Ciao/Prolog menu or by typing ⟨^C⟩ ⟨I⟩ ⟨S⟩.
- You can also work with the preprocessor and auto-documenter directly from emacs: see their manuals or browse through the corresponding menus that appear when editing `.pl` files.

We encourage you once more to read Chapter 10 [Using Ciao inside GNU emacs], page 57 to discover the many other functionalities of this environment.

3.4 Keeping up to date (Win32)

You may want to read Chapter 137 [Beyond installation], page 521 for instructions on how to sign up on the Ciao user's mailing list, receive announcements regarding new versions, download new versions, report bugs, etc.

PART I - The program development environment

This part documents the components of the basic Ciao program development environment. They include:

ciaoc: the standalone compiler, which creates executables without having to enter the interactive top-level.

ciaosh: (also invoked simply as `ciao`) is an interactive top-level shell, similar to the one found on most Prolog systems (with some enhancements).

debugger.pl: a Byrd box-type debugger, similar to the one found on most Prolog systems (also with some enhancements, such as source-level debugging). This is not a standalone application, but is rather included in `ciaosh`, as is done in other Prolog systems. However, it is also *embeddable*, in the sense that it can be included as a library in executables, and activated dynamically and conditionally while such executables are running.

ciao-shell: an interpreter/compiler for *Prolog scripts* (i.e., files containing Prolog code which run without needing explicit compilation).

ciao.el: a *complete program development enviroment*, based on GNU emacs, with syntax coloring, direct access to all the tools described above (as well as the preprocessor and the documenter), atomatic location of errors, source-level debugging, context-sensitive access to on-line help/manuals, etc. The use of this environment is *very highly recommended*!

The Ciao program development environment also includes `ciaopp`, the preprocessor, and `lpdoc`, the documentation generator, which are described in separate manuals.

4 The stand-alone command-line compiler

Author(s): Daniel Cabeza and the CLIP Group.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.5#91 (2000/3/24, 17:18:52 CET)

`ciaoc` [CH00a] is the Ciao stand-alone command-line compiler. `ciaoc` can be used to create executables or to compile individual files to object code (to be later linked with other files). `ciaoc` is specially useful when working from the command line. Also, it can be called to compile Ciao programs from other tools such as, e.g., shell scripts, `Makefiles`, or project files. All the capabilities of `ciaoc` are also available from the interactive top-level shell, which uses the ciaoc modules as its components.

4.1 Introduction to building executables

An *executable* can be built from a single file or from a collection of inter-related files. In the case of only one file, this file must define the predicate `main/0` or `main/1`. This predicate is the one which will be called when the executable is started. As an example, consider the following file, called `hello.pl`:

```
main :-
     write('Hello world'),
     nl.
```

To compile it from the command line using the `ciaoc` standalone compiler it suffices to type "`ciaoc hello`" (in Win32 you may have to put the complete path to the `ciaoc` folder of the Ciao distribution, where the installation process leaves a `ciaoc.bat` file):

```
/herme@clip:/tmp
[60]> ciaoc hello

/herme@clip:/tmp
[61]>
```

This produces an executable called `hello` in Un*x-like systems and `hello.cpx` under Win32 systems. This executable can then be run in Win32 by double-clicking on it and on Un*x systems by simply typing its name (see for Section 4.3 [Running executables from the command line], page 26 for how to run executables from the command line in Win32):

```
/herme@clip:/tmp
[61]> hello
Hello world
```

If the application is composed of several files the process is identical. Assume `hello.pl` is now:

```
:- use_module(aux,[p/1]).

main :-
     p(X),
     write(X),
     nl.
```

where the file `aux.pl` contains:

```
:- module(aux,[p/1]).

p('Hello world').
```

This can again be compiled using the `ciaoc` standalone compiler as before:

```
/herme@clip:/tmp
[60]> ciaoc hello

/herme@clip:/tmp
[61]> hello
Hello world
```

The invocation of `ciaoc hello` compiles the file `hello.pl` and all connected files that may need recompilation – in this case the file `aux.pl`. Also, if any library files used had not been compiled previously they would be compiled at this point (See Section 4.6 [Intermediate files in the compilation process], page 29). Also, if, say, `hello.pl` is changed and recompiled, the object code resulting from the previous compilation of `aux.pl` will be reused. This is all done without any need for `Makefile`s, and considerably accelerates the development process for large applications. This process can be observed by selecting the `-v` option when invoking `ciaoc` (which is equivalent to setting the `verbose_compilation` Prolog flag to `on` in the top-level interpreter).

If `main/1` is defined instead of `main/0` then when the executable is started the argument of `main/1` will be instantiated to a list of atoms, each one of them corresponding to a command line option. Consider the file `say.pl`:

```
main(Argv) :-
     write_list(Argv), nl.

write_list([]).
write_list([Arg|Args]) :-
     write(Arg),
     write(' '),
     write_list(Args).
```

Compiling this program and running it results in the following output:

```
/herme@clip:/tmp
[91]> ciaoc say

/herme@clip:/tmp
[91]> say hello dolly
hello dolly
```

The name of the generated executable can be controlled with the `-o` option (See Section 4.7 [Usage (ciaoc)], page 29).

4.2 Paths used by the compiler during compilation

The compiler will look for files mentioned in commands such as `use_module/1` or `ensure_loaded/1` in the current directory. Other paths can be added by including them in a file whose name is given to `ciaoc` using the `-u` option. This file should contain facts of the predicates `file_search_path/2` and `library_directory/1` (see the documentation for these predicates and also Chapter 9 [Customizing library paths and path aliases], page 55 for details).

4.3 Running executables from the command line

As mentioned before, what the `ciaoc` compiler generates and how it is started varies somewhat from OS to OS. In general, the product of compiling an application with `ciaoc` is a file that contains the bytecode (the product of the compilation) and invokes the Ciao engine on it.

- Un Un*x this is a *script* (see the first lines of the file) which invokes the ciao engine on this file. To run the generated executable from a Un*x shell, or from the `bash` shell that comes with the Cygwin libraries (see Section 135.6 [Installation and compilation under Windows], page 514) it suffices to type its name at the shell command line, as in the examples above.
- In a Win32 system, the compiler produces a similar file with a `.cpx` ending. The Ciao installation process typically makes sure that the Windows registry contains the right entries so that this executable will run upon double-cliking on it.

  In you want to run the executable from the command line an additional `.bat` file is typically needed. To help in doing this, the Win32 installation process creates a `.bat` skeleton file called `bat_skel` in the `Win32` folder of the distribution) which allows running Ciao executables from the command line. If you want to run a Ciao executable `file.cpx` from the command line, you normally copy the skeleton file to the folder were the executable is and rename it to `file.bat`, then change its contents as explained in a comment inside the file itself.

  Note that this `.bat` file is usually not necessary in NT, as its command shell understands file extension associations. I.e., in windows NT it is possible to run the `file.cpx` executable directly. Due to limitations of `.bat` files in Windows 95/98, in those OSs no more than 9 command line arguments can be passed to the executable (in NT there is no such restriction).

  Finally, in a system in which Cygnus Win32 is installed executables can also be used directly from the `bash` shell command line, without any associated `.bat` files, by simply typing their name at the `bash` shell command line, in the same way as in Un*x. This only requires that the `bash` shell which comes with Cygnus Win32 be installed and accessible: simply, make sure that `/bin/sh.exe` exists.

Except for a couple of header lines, the contents of executables are almost identical under different OSs. The bytecode they contain is architecture-independent. In fact, it is possible to create an executable under Un*x and run it on Windows or viceversa, by making only minor modifications (e.g., creating the `.bat` file and/or setting environment variables or editing the start of the file to point to the correct engine location).

4.4 Types of executables generated

While the default options used by `ciaoc` are sufficient for normal use, by selecting other options `ciaoc` can generate several different types of executables, which offer interesting tradeoffs among size of the generated executable, portability, and startup time [CH00a]:

Dynamic executables:

: `ciaoc` produces by default *dynamic* executables. In this case the executable produced is a platform-independent file which includes in compiled form all the user defined files. On the other hand, any system libraries used by the application are loaded dynamically at startup. More precisely, any files that appear as `library(...)` in `use_module/1` and `ensure_loaded/1` declarations will not be included explicitly in the executable and will intead be loaded dynamically. Is is also possible to mark other path aliases (see the documentation for `file_search_path/2`) for dynamic loading by using the `-d` option. Files accessed through such aliases will also be loaded dynamically.

  Dynamic loading allows making smaller executables. Such executables may be used directly in the same machine in which they were compiled, since suitable paths to the location of the libraries will be included as default in the executable by `ciaoc` during compilation.

  The executable can also be used in another machine, even if the architecture and OS are different. The requirement is that the Ciao libraries (which will also include the appropriate Ciao engine for that architecture and OS) be installed in the target machine, and that the `CIAOLIB` and `CIAOENGINE` environment variables are set appropriately for the executable to be able to find them (see Section 4.5 [Environment variables used by Ciao executables], page 28). How to do this differs slightly from OS to OS.

Static executables:

Selecting the `-s` option `ciaoc` produces a *static* executable. In this case the executable produced (again a platform-independent file) will include in it all the auxiliary files and any system libraries needed by the application. Thus, such an executable is almost complete, needing in order to run only the Ciao engine, which is platform-specific.[1] Again, if the executable is run in the same machine in which it was compiled then the engine is found automatically. If the executable is moved to another machine, the executable only needs access to a suitable engine (which can be done by setting the `CIAOENGINE` environment variable to point to this engine).

This type of compilation produces larger executables, but has the advantage that these executables can be installed and run in a different machine, with different architecture and OS, even if Ciao is not installed on that machine. To install (or distribute) such an executable, one only needs to copy the executable file itself and the appropriate engine for the target platform (See Chapter 135 [Installing Ciao from the source distribution], page 509 or Chapter 136 [Installing Ciao from a Win32 binary distribution], page 519 and Section 135.5 [Multiarchitecture support], page 513), and to set things so that the executable can find the engine.[2]

Dynamic executables, with lazy loading:

Selecting the `-ll` option is very similar to the case of dynamic executables above, except that the code in the library modules is not loaded when the program is started but rather it is done during execution, the first time a predicate defined in that file is called. This is advantageous if a large application is composed of many parts but is such that typically only some of the parts are used in each invocation. The Ciao preprocessor, `ciaopp`, is a good example of this: it has many capabilitites but typically only some of them are used in a given session. An executable with lazy load has the advantage that it starts fast, loading a minimal functionality on startup, and then loads the different modules automatically as needed.

4.5 Environment variables used by Ciao executables

The executables generated by the Ciao compiler (including the ciao development tools themselves) locate automatically where the Ciao engine and libraries have been installed, since those paths are stored as defaults in the engine and compiler at installation time. Thus, there is no need for setting any environment variables in order to *run* Ciao executables (on a single architecture – see Section 135.5 [Multiarchitecture support], page 513 for running on multiple architectures).

---

[1] Currently there is an exception to this related to libraries which are written in languages other than Prolog, as, e.g., C. C files are currently always compiled to dynamically loadable object files (`.so` files), and they thus need to be included manually in a distribution of an application. This will be automated in upcoming versions of the Ciao system.

[2] It is also possible to produce real standalone executables, i.e., executables that do not need to have an engine around. However, this is not automated yet, although it is planned for an upcoming version of the compiler. In particular, the compiler can generate a `.c` file for each `.pl` file. Then all the `.c` files can be compiled together into a real executable (the engine is added one more element during link time) producing a complete executable for a given architecture. The downside of course is that such an executable will not be portable to other architectures without recompilation.

However, the default paths can be overridden by using the environment variables `CIAOENGINE` and `CIAOLIB`. The first one will tell the Ciao executables where to look for an engine, and the second will tell them where to look for the libraries. Thus, it is possible to actually use the Ciao system without installing it by setting these variables to the following values:

- `CIAOENGINE`: `$(SRC)/bin/$(CIAOARCH)/ciaoengine`
- `CIAOLIB`: `$(SRC)`

where `$(CIAOARCH)` is the string echoed by the command `SRC/etc/ciao_get_arch` (or `BINROOT/ciao_get_arch`, after installation).

This allows using alternate engines or libraries, which can be very useful for system development and experimentation.

4.6 Intermediate files in the compilation process

Compiling an individual source (i.e., `.pl`) file produces a `.itf` file and a `.po` file. The `.itf` file contains information of the *modular interface* of the file, such as information on exported and imported predicates and on the other modules used by this module. This information is used to know if a given file should be recompiled at a given point in time and also to be able to detect more errors statically including undefined predicates, mismatches on predicate charaterictics across modules, etc. The `.po` file contains the platform-independent object code for a file, ready for linking (statically or dynamically).

It is also possible to use `ciaoc` to explicitly generate the `.po` file for one or more `.pl` files by using the `-c` option.

4.7 Usage (ciaoc)

The following provides details on the different command line options available when invoking `ciaoc`:

```
ciaoc <MiscOpts> <ExecOpts> [-o <execname>] <file> ...

  Make an executable from the listed files.  If there is
  more than one file, they must be non-module, and the
  first one must include the main predicate.  The -o
  option allows generating an arbitrary executable name.

ciaoc <MiscOpts> <ExecOpts> -a <publishmod> <module>

  Make an active module executable from <module> with
  address publish module <publishmod>.

ciaoc <MiscOpts> -c  <file> ...

  Compile listed files (make .po objects).

<MiscOpts> can be: [-v] [-ri] [-u <file>]

-v  verbose mode

-ri generate readable .itf files
```

```
-u  use <file> for compilation

<ExecOpts> can be: [-s|-e|-l|(-ll <module>)*] (-d <alias>)* [-x]

-s  make a static executable (otherwise dynamic files are not included)

-e  make executable with eager load of dynamic files at startup (default)

-l  idem with lazy load of dynamic files (except insecure cases)

-ll force <module> to be loaded lazily,  implies -l

-d  files using this path alias are dynamic (default: library)

-x  Extended recompilation: only useful for Ciao standard library developers

default extension for files is '.pl'
```

5 The interactive top-level shell

Author(s): Daniel Cabeza and the CLIP Group.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.5#11 (1999/12/14, 13:14:36 MET)

`ciaosh` is the Ciao interactive top-level shell. It provides the user with an interactive programming environment with tools for incrementally building programs, debugging programs by following their executions, and modifying parts of programs without having to start again from scratch. If available, it is strongly recommended to use it with the emacs interface provided, as it greatly simplifies the operation. This chapter documents general operation in the shell itself. Other chapters document the

5.1 Shell invocation and startup

When invoked, the shell responds with a message of identification and the prompt `?-` as soon as it is ready to accept input, thus:

```
Ciao-Prolog X.Y #PP: Thu Mar 25 17:20:55 MET 1999
?-
```

When the shell is initialized it looks for a file `.ciaorc` in the HOME directory and makes an `include` of it, if it exists. This file is useful for including `use_module/1` declarations for the modules one wants to be loaded by default, changing prolog flags, etc. If the initialization file does not exist, the ISO-Prolog compatibility package `iso` is included, to provide (almost) all the ISO builtins by default. Two command-line options control the loading of the initialization file:

`-f` Fast start, do not load any initialization file.

`-l File` Look for initialization file `File` instead of `~/.ciaorc`. If it does not exist, include the compatibility package `iso`.

5.2 Shell interaction

After the shell outputs the prompt, it is expecting either an internal command (see the following sections) or a *query* (a goal or sequence of goals). When typing in the input, which must be a valid prolog term, if the term does not end in the first line, subsequent lines are indented. For example:

```
?- X =
   f(a,
   b).

X = f(a,b) ?

yes
?-
```

The queries are executed by the shell as if they appeared in the user module. Thus, in addition to builtin predicates, predicates available to be executed directly are all predicates defined by loaded user files (files with no module declaration), and imported predicates from modules by the use of `use_module`.

The possible answers of the shell, after executing an internal command or query, are:

- If the execution failed (or produced an error), the answer is `no`.
- If the execution was successful, and no answer variable (see below) was bound (or constraints where imposed on such variables), the answer is simply `yes`.

- If the execution was successful and bindings where made (or constraints where imposed) on answer variables, then the shell outputs the values of answer variables, as a sequence of bindings (or constraints), and then prints a `?` as a prompt. At this point it is expecting an input line from the user. By entering a carriage-return (⟨RET⟩) or any line starting with `y`, the query terminates and the shell answer `yes`. Entering a '`,`' the shell enters a recursive level (see below). Finally, any other answer forces the system to backtrack and look for the next solution (answering as with the first solution).

To allow using connection variables in queries without having to report their results, variables whose name starts with `_` are not considered in answers, the rest being the *answer variable*s. This example illustrates the previous points:

```
?- member(a, [b, c]).

no
?- member(a, [a, b]).

yes
?- member(X, [a|L]).

X = a ? ;

L = [X|_] ?

yes
?- atom_codes(ciao, _C), member(L, _C).

L = 99 ? ;

L = 105 ? ;

L = 97 ? ;

L = 111 ? ;

no
?-
```

5.3 Entering recursive (conjunctive) shell levels

As stated before, when the user answers with '`,`' after a solution is presented, the shell enters a *recursive level*, changing its prompt to N `?-` (where N is the recursion level) and keeping the bindings or constraints of the solution (this is inspired by the *LogIn* language developed by *H. Ait-Kaci*, *P. Lincoln* and *Roger Nasr* [AKNL86]). Thus, the following queries will be executed within that context, and all variables in the lower level solutions will be reported in subsequent solutions at this level. To exit a recursive level, input an ⟨EOF⟩ character or the command `up`. The last solution after entering the level is repeated, to allow asking for more solutions. Use command `top` to exit all recursive levels and return to the top level. Example interaction:

```
?- directory_files('.',_Fs), member(F,_Fs).

F = 'file_utils.po' ? ,

1 ?- file_property(F, mod_time(T)).
```

```
F = 'file_utils.po',
T = 923497679 ?

yes
1 ?- up.

F = 'file_utils.po' ? ;

F = 'file_utils.pl' ? ;

F = 'file_utils.itf' ? ,

1 ?- file_property(F, mod_time(T)).

F = 'file_utils.itf',
T = 923497679 ?

yes
1 ?- ^D
F = 'file_utils.itf' ?

yes
?-
```

5.4 Usage and interface (ciaosh)

- **Library usage:**
  The following predicates can be used at the top-level shell natively (but see also the commands available in Chapter 6 [The interactive debugger], page 39 which are also available within the top-level shell).
- **Exports:**
  - *Predicates:*
    use_module/1, use_module/2, ensure_loaded/1, include/1, use_package/1, consult/1, compile/1, ./2, make_po/1, unload/1, set_debug_mode/1, set_nodebug_mode/1, make_actmod/2, force_lazy/1, undo_force_lazy/1, dynamic_search_path/1, multifile/1.
- **Other modules used:**
  - *Application modules:*
    toplev.
  - *System library modules:*
    libpaths, compiler/compiler, compiler/exemaker, compiler/c_itf, debugger/debugger.

5.5 Documentation on exports (ciaosh)

use_module/1: PREDICATE

Usage: `use_module(Module)`

- *Description:* Load into the top-level the module defined in `Module`, importing all the predicates it exports.
- *The following properties should hold at call time:*
  `Module` is a source name. (`streams_basic:sourcename/1`)

use_module/2: PREDICATE

Usage: `use_module(Module,Imports)`

- *Description:* Load into the top-level the module defined in `Module`, importing the predicates in `Imports`.
- *The following properties should hold at call time:*
  `Module` is a source name. (`streams_basic:sourcename/1`)
  `Imports` is a list of `prednames`. (`basic_props:list/2`)

ensure_loaded/1: PREDICATE

Usage: `ensure_loaded(File)`

- *Description:* Load into the top-level the code residing in file (or files) `File`, which is user (i.e. non-module) code.
- *The following properties should hold at call time:*
  `File` is a source name or a list of source names. (`ciaosh_doc:sourcenames/1`)

make_exec/2: (UNDOC_REEXPORT)

Imported from `toplev` (see the corresponding documentation for details).

include/1: PREDICATE

Usage: `include(File)`

- *Description:* The contents of the file `File` are included in the top-level shell. For the moment, it only works with some directives, which are interpreted by the shell, or with normal clauses (which are asserted), if library(dynamic) is loaded beforehand.
- *The following properties should hold at call time:*
  `File` is a source name. (`streams_basic:sourcename/1`)

use_package/1: PREDICATE

Usage: `use_package(Package)`

- *Description:* Equivalent to issuing an include(library(`Package`)) for each listed file. By now some package contents cannot be handled.
- *The following properties should hold at call time:*
  `Package` is a source name or a list of source names. (`ciaosh_doc:sourcenames/1`)

consult/1: PREDICATE

Usage: `consult(File)`

- *Description:* Provided for backward compatibility. Similar to `ensure_loaded/1`, but ensuring each listed file is loaded in consult mode (see Chapter 6 [The interactive debugger], page 39).
- *The following properties should hold at call time:*

  `File` is a source name or a list of source names. (`ciaosh_doc:sourcenames/1`)

compile/1: PREDICATE

Usage: `compile(File)`

- *Description:* Provided for backward compatibility. Similar to `ensure_loaded/1`, but ensuring each listed file is loaded in compile mode (see Chapter 6 [The interactive debugger], page 39).
- *The following properties should hold at call time:*

  `File` is a source name or a list of source names. (`ciaosh_doc:sourcenames/1`)

./2: PREDICATE

Usage: `[File|Files]`

- *Description:* Provided for backward compatibility, obsoleted by `ensure_loaded/1`.
- *The following properties should hold at call time:*

  `File` is a source name. (`streams_basic:sourcename/1`)

  `Files` is a list of `sourcename`s. (`basic_props:list/2`)

make_po/1: PREDICATE

Usage: `make_po(Files)`

- *Description:* Make object (`.po`) files from `Files`. Equivalent to executing `"ciaoc -c"` on the files.
- *The following properties should hold at call time:*

  `Files` is a source name or a list of source names. (`ciaosh_doc:sourcenames/1`)

unload/1: PREDICATE

Usage: `unload(File)`

- *Description:* Unloads dynamically loaded file `File`.
- *The following properties should hold at call time:*

  `File` is a source name. (`streams_basic:sourcename/1`)

set_debug_mode/1: PREDICATE

Usage: `set_debug_mode(File)`

- *Description:* Set the loading mode of `File` to *consult*. See Chapter 6 [The interactive debugger], page 39.
- *The following properties should hold at call time:*

  `File` is a source name. (`streams_basic:sourcename/1`)

set_nodebug_mode/1: PREDICATE

Usage: `set_nodebug_mode(File)`

- *Description:* Set the loading mode of `File` to *compile*. See Chapter 6 [The interactive debugger], page 39.
- *The following properties should hold at call time:*

  `File` is a source name. (`streams_basic:sourcename/1`)

make_actmod/2: PREDICATE

Usage: `make_actmod(ModuleFile,PublishMod)`

- *Description:* Make an active module executable from the module residing in `ModuleFile`, using address publish module of name `PublishMod` (which needs to be in the library paths).
- *The following properties should hold at call time:*

  `ModuleFile` is a source name. (`streams_basic:sourcename/1`)

  `PublishMod` is an atom. (`basic_props:atm/1`)

force_lazy/1: PREDICATE

Usage: `force_lazy(Module)`

- *Description:* Force module of name `Module` to be loaded lazily in the subsequent created executables.
- *The following properties should hold at call time:*

  `Module` is an atom. (`basic_props:atm/1`)

undo_force_lazy/1: PREDICATE

Usage: `undo_force_lazy(Module)`

- *Description:* Disable a previous `force_lazy/1` on module `Module` (or, if it is uninstantiated, all previous `force_lazy/1`).
- *Calls should, and exit will be compatible with:*

  `Module` is an atom. (`basic_props:atm/1`)

dynamic_search_path/1: PREDICATE

Usage: `dynamic_search_path(Name)`

- *Description:* Asserting a fact to this data predicate, files using path alias `Name` will be treated as dynamic in the subsequent created executables.
- *The following properties should hold at call time:*

  `Name` is an atom. (`basic_props:atm/1`)

multifile/1: PREDICATE

Usage: `multifile Pred`

- *Description:* Dynamically declare predicate `Pred` as multifile. This is useful at the top-level shell to be able to call multifile predicates of loaded files.

– *The following properties should hold at call time:*

`Pred` is a Name/Arity structure denoting a predicate name:

```
predname(P/A) :-
        atm(P),
        int(A).
```

(`basic_props:predname/1`)

5.6 Documentation on internals (`ciaosh`)

sourcenames/1: PROPERTY

Is defined as follows:

```
sourcenames(File) :-
        sourcename(File).
sourcenames(Files) :-
        list(Files,sourcename).
```

See `sourcename/1` in Chapter 22 [Basic file/stream handling], page 115

Usage: `sourcenames(Files)`

– *Description:* `Files` is a source name or a list of source names.

6 The interactive debugger

Author(s): D. Cabeza, Manuel C. Rodriguez, (A. Ciepielewski, M. Carlsson, T. Chikayama, K. Shen).

The Ciao program development environment includes a number of advanced debugging tools, such as a source-level debugger, the `ciaopp` preprocessor, and some execution visualizers. Herein we discuss the interactive debugger available in the standard top-level, which allows tracing the control flow of programs, in a similar way to other popular Prolog systems. This is a classical Byrd *box-type debugger* [Byr80,BBP81], with some enhancements, most notably being able to track the execution on the source program. We also discuss the embedded debugger, which is a version of the debugger which can be embedded into executables so that an interactive debugging session can be triggered at any time while running that executable without needing the top-level shell.

Byrd's Procedure Box model of debugging execution provides a simple way of visualising control flow, including backtracking. Control flow is in principle viewed at the predicate level, rather than at the level of individual clauses. The Ciao debugger has the ability to mark selected modules and/or files for debugging (traditional and source debugging), rather than having to exhaustively trace your program. It also allows to selectively set spy-points and breakpoints. Spy-points allow the programmer to nominate interesting predicates at which program execution is to pause so that the programmer can interact with the debugger. Breakpoints are similar to spy-points, but allow pausing at a specific line in the code, corresponding to a particular literal. There is a wide choice of control and information options available during debugging interaction.

Note: While the debugger described herein can be used in a standalone way (i.e., from a operating system shell or terminal window) in the same way as other Prolog debuggers, the most convenient way of debugging Ciao programs is by using the emacs mode (see Chapter 10 [Using Ciao inside GNU emacs], page 57), i.e., debugging from within the `emacs` editor / programming environment.

6.1 Marking modules and files for debugging in the top-level debugger

Usually, when a program is not working properly, the programmer has a feeling of which are the modules where the fault may be. Since full-fledged debugging is only available on *interpreted* (called *interpreted mode* in traditional Prolog systems) modules, which are executed much slower than compiled modules, there is the posibility of telling the top level which particular modules are to be loaded in *interpreted mode*, with the aim of debugging them. The simplest way of achieving this is by executing in the Ciao shell prompt, for each suspicious module `Module` in the program, a command like this:

```
?- debug_module(Module).
```

An alternative way of loading a module in interpreted mode exists which will instruct the debugger to keep track of the line numbers in the source file and to report them during debugging. This feature can be selected for a suspicious module `Module` in the program by executing a command such as:

```
?- debug_module_source(Module).
```

This is most useful when running the top-level inside the `emacs` editor since in that case the Ciao emacs mode allows performing full source-level debugging in each module marked as above, i.e., the source lines being executed will be highlighted dynamically during debugging in a window showing the source code of the module.

Note that all files with no module declaration belong to the pseudo-module `user`, so the command to be issued for debugging a user file, say `foo.pl`, would be `debug_module(user)` or `debug_module_source(user)`, and not `debug_module(foo)`.

The two ways of performing source-level debugging are fully compatible between them, i.e., Ciao allows having some modules loaded with `debug_module/1` and others with `debug_module_source/1`. To change from one interpreted mode to the other mode it suffices to select the module with the new interpreted mode (debugger mode), using the appropiate command, and reload the module.

The commands above perform in fact two related actions: first, they let the compiler know that if a file containing a module with this name is loaded, it should be loaded in interpreted mode (source or traditional). In addition, they instruct the debugger to actually prepare for debugging the code belonging to that module. After that, the modules which are to be debugged have to be (re)loaded so that they are compiled or loaded for interpretation in the appropriate way. The nice thing is that, due to the modular behaviour of the compiler/top-level, if the modules are part of a bigger application, it suffices to load the main module of the application, since this will automatically force the dependent modules which have changed to be loaded in the appropriate way, including those whose *loading mode* has changed (i.e., changing the loading mode has the effect of forcing the required re-loading of the module at the appropriate time).

Later in the debugging process, as the bug location is isolated, typically one will want to restrict more and more the modules where debugging takes place. To this end, and without the need for reloading, one can tell the debugger to not consider a module for debugging issuing a `nodebug_module/1` command, which counteracts a `debug_module/1` or `debug_module_source/1` command with the same module name, and reloading it (or the main file).

There are also two top-level commands `set_debug_mode/1` and `set_nodebug_mode/1`, which accept as argument a file spec (i.e., `library(foo)` or `foo`, even if it is a user file) to be able to load a file in interpreted mode without changing the set of modules that the debugger will try to spy.

6.2 The debugging process

Once modules or user files are marked for debugging and reloaded, the traditional debugging shell commands can be used (the documentation of the `debugger` library following this chapter contains all the commands and their description), with the same meaning as in other classical Prolog systems. The differences in their behavior are:

- Debugging takes place only in the modules in which it was activated,
- `nospy/1` and `spy/1` accept sequences of predicate specs, and they will search for those predicates only in the modules marked for debugging (traditional or source-level debugging).
- `breakpt/6` and `nobreakpt/6` allow setting breakpoints at selected clause literals and will search for those literals only in the modules marked for source-level debugging (modules marked with `debug_module_source/1`).

In particular, the system is initially in nodebug mode, in which no tracing is performed. The system can be put in debug mode by a call to `debug/0` in which execution of queries will proceed until the first *spy-point* or *breakpoint*. Alternatively, the system can be put in trace mode by a call to `trace/0` in which all predicates will be trace.

6.3 Marking modules and files for debugging with the embedded debugger

The embedded debugger, as the interpreted debugger, has three different modes of operation: debug, trace or nodebug. These debuggers modes can be set by adding a package declaration in the module, as follows:

```
:- use_package(debug).
:- use_package(trace).
:- use_package(nodebug).
```

and recompiling the application.

In order to debug, or trace, correctly the complete code these declarations *must* be the last ones. Also it is possible to add the package in the module declaration using the predicate `module/3`.

The embedded debugger has limitations over the interpreted debugger. The most important is that the "retry" option is not available. But it is possible to add, and remove, spy-points and breakpoins using the predicates `spy/1`, `nospy/1`, `breakpt/6` and `nobreakpt/6`, etc. These can be used in a clause declaration or as declarations. Also it is possible to add in the code predicates for issuing the debugger (i.e., use debug mode, and in a clause add the predicate `trace/1`).

The nodebug mode allows keeping the spy-points and breakpoints in the code instead of removing them from the code.

Note that in order to use the embedded debugger the buffer where the program will run must be in the Ciao inferior mode (`M-x ciao-inferior-mode`).

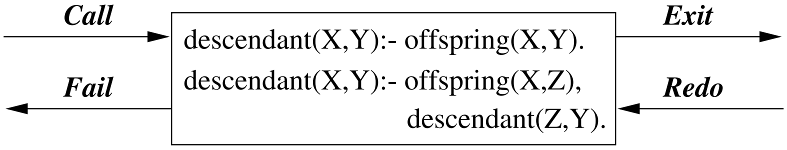
See the following example:

```
:- module( foo,[main/1],[assertions, debug]).

:- entry main/1.

main(X) :-
         display(X),
         spy(foo),
         foo(X),
         notrace,
         nl.

foo([]).
foo([X|T]) :-
        trace,
        bar(X),
        foo(T).

bar(X) :-
        display(X).
```

6.4 The procedure box control flow model

During debugging the interpreter prints out a sequence of goals in various states of instantiation in order to show the state that the program has reached in its execution. However, in order to understand what is occurring it is necessary to understand when and why the interpreter prints out goals. As in other programming languages, key points of interest are procedure entry and return, but in Prolog there is the additional complexity of backtracking. One of the major confusions that novice Prolog programmers have to face is the question of what actually happens when a goal fails and the system suddenly starts backtracking. The Procedure Box model of Prolog execution views program control flow in terms of movement about the program text. This model provides a basis for the debugging mechanism in the interpreter, and enables the user to view the behaviour of the program in a consistent way. It also provides the basis for the visualization performed on the source level program when source level program when source-level debugging is activated within `emacs`.

Let us look at an example Prolog procedure:



The first clause states that `Y` is a descendant of `X` if `Y` is an offspring of `X`, and the second clause states that `Y` is a descendant of `X` if `Z` is an offspring of `X` and `Y` is a descendant of `Z`. In the diagram a box has been drawn around the whole procedure and labelled arrows indicate the control flow in and out of this box. There are four such arrows which we shall look at in turn.

- **Call**

  This arrow represents initial invocation of the procedure. When a goal of the form `descendant(X,Y)` is required to be satisfied, control passes through the Call port of the descendant box with the intention of matching a component clause and then satisfying any subgoals in the body of that clause. Note that this is independent of whether such a match is possible; i.e. first the box is called, and then the attempt to match takes place. Textually we can imagine moving to the code for descendant when meeting a call to descendant in some other part of the code.

- **Exit**

  This arrow represents a successful return from the procedure. This occurs when the initial goal has been unified with one of the component clauses and any subgoals have been satisfied. Control now passes out of the Exit port of the descendant box. Textually we stop following the code for descendant and go back to the place we came from.

- **Redo**

  This arrow indicates that a subsequent goal has failed and that the system is backtracking in an attempt to find alternatives to previous solutions. Control passes through the Redo port of the descendant box. An attempt will now be made to resatisfy one of the component subgoals in the body of the clause that last succeeded; or, if that fails, to completely rematch the original goal with an alternative clause and then try to satisfy any subgoals in the body of this new clause. Textually we follow the code backwards up the way we came looking for new ways of succeeding, possibly dropping down on to another clause and following that if necessary.

- **Fail**

  This arrow represents a failure of the initial goal, which might occur if no clause is matched, or if subgoals are never satisfied, or if any solution produced is always rejected by later processing. Control now passes out of the Fail port of the descendant box and the system continues to backtrack. Textually we move back to the code which called this procedure and keep moving backwards up the code looking for choice points.

In terms of this model, the information we get about the procedure box is only the control flow through these four ports. This means that at this level we are not concerned with which clause matches, and how any subgoals are satisfied, but rather we only wish to know the initial goal and the final outcome. However, it can be seen that whenever we are trying to satisfy subgoals, what we are actually doing is passing through the ports of *their* respective boxes. If we were following this (e.g., activating source-level debugging), then we would have complete information about the control flow inside the procedure box.

Note that the box we have drawn around the procedure should really be seen as an invocation box. That is, there will be a different box for each different invocation of the procedure. Obviously, with something like a recursive procedure, there will be many different Calls and Exits in the control flow, but these will be for different invocations. Since this might get confusing each invocation box is given a unique integer identifier in the messages, as described below.

Note that not all procedure calls are traced; there are a few basic predicates which have been made invisible since it is more convenient not to trace them. These include debugging directives, basic control structures, and some builtins. This means that messages will never be printed for these predicates during debugging.

6.5 Format of debugging messages

This section explains the two formats of the message output by the debugger at a port. All trace messages are output to the terminal regardless of where the current output stream is directed (which allows tracing programs while they are performing file I/O). The basic format, which will be shown in traditional debug and in source-level debugging within Ciao `emacs` mode, is as follows:

```
S  13  7  Call: T user:descendant(dani,_123) ?
```

`S` is a spy-point or breakpoint indicator. It is printed as '+', indicating that there is a spy-point on `descendant/2` in module `user`, as `'B'` denoting a breakpoint, or as ' ', denoting no spy-point or breakpoint. If there is a spy-point and a breakpoint in the same predicate the spy-point indicator takes preference over breakpoint indicator.

`T` is a subterm trace. This is used in conjunction with the `^` command (set subterm), described below. If a subterm has been selected, `T` is printed as the sequence of commands used to select the subterm. Normally, however, `T` is printed as ' ', indicating that no subterm has been selected.

The first number is the unique invocation identifier. It is always nondecreasing (provided that the debugger is switched on) regardless of whether or not the invocations are being actually seen. This number can be used to cross correlate the trace messages for the various ports, since it is unique for every invocation. It will also give an indication of the number of procedure calls made since the start of the execution. The invocation counter starts again for every fresh execution of a command, and it is also reset when retries (see later) are performed.

The number following this is the *current depth*; i.e., the number of direct *ancestors* this goal has. The next word specifies the particular port (`Call`, `Exit`, `Redo` or `Fail`). The goal is then printed so that its current instantiation state can be inspected. The final `?` is the prompt indicating that the debugger is waiting for user interaction. One of the option codes allowed (see below) can be input at this point.

The second format, quite similar to the format explained above, is shown when using source-level debugging outside the Ciao `emacs` mode, and it is as follows:

```
     In /home/mcarlos/ciao/foo.pl (5-9) descendant-1
S  13  7  Call: T user:descendant(dani,_123) ?
```

This format is identical to the format above except for the first line, which contains the information for location of the point in the source program text where execution is currently at. The first line contains the name of the source file, the start and end lines where the literal can be found, the substring to search for between those lines and the number of substrings to locate. This information for locating the point on the source file is not shown when executing the source-level debugger from the Ciao `emacs` mode.

Ports can be "unleashed" by calling the `leash/1` predicate omiting that port in the argument. This means that the debugger will stop but user interaction is not possible for an unleashed port. Obviously, the `?` prompt will not be shown in such messages, since the user has specified that no interaction is desired at this point.

6.6 Options available during debugging

This section describes the particular options that are available when the debugger prompts after printing out a debugging message. All the options are one letter mnemonics, some of which can be optionally followed by a decimal integer. They are read from the terminal with any blanks being completely ignored up to the next terminator (carriage-return, line-feed, or escape). Some options only actually require the terminator; e.g., the creep option, only requires ⟨RET⟩.

The only option which really needs to be remembered is 'h' (followed by ⟨RET⟩). This provides help in the form of the following list of available options.

```
<cr>   creep             c      creep
 l     leap              s      skip
 r     retry             r <i>  retry i
 f     fail              f <i>  fail i
 d     display           p      print
 w     write
 g     ancestors         g <n>  ancestors n
 n     nodebug           =      debugging
 +     spy this          -      nospy this
 a     abort
 @     command           u      unify
 <     reset printdepth  < <n>  set printdepth
 ^     reset subterm     ^ <n>  set subterm
 ?     help              h      help
```

- `c` (*creep*)

  causes the debugger to single-step to the very next port and print a message. Then if the port is leashed the user is prompted for further interaction. Otherwise it continues creeping. If leashing is off, creep is the same as leap (see below) except that a complete trace is printed on the terminal.

- `l` (*leap*)

  causes the interpreter to resume running the program, only stopping when a spy-point or breakpoint is reached (or when the program terminates). Leaping can thus be used to follow the execution at a higher level than exhaustive tracing. All that is needed to do is to set spy-points and breakpoints on an evenly spread set of pertinent predicates or lines, and then follow the control flow through these by leaping from one to the other.

- `s` (*skip*)

  is only valid for Call and Redo ports, if it is issued in Exit or Fail ports it is equivalent to creep. It skips over the entire execution of the predicate. That is, no message will be seen until control comes back to this predicate (at either the Exit port or the Fail port). Skip is particularly useful while creeping since it guarantees that control will be returned after the (possibly complex) execution within the box. With skip then no message at all will appear until control returns to the Exit port or Fail port corresponding to this Call port or Redo port. This includes calls to predicates with spy-points and breakpoints set: they will be masked out during the skip. There is a way of overriding this: the `t` option after a ⟨^C⟩ interrupt will disable the masking. Normally, however, this masking is just what is required!

- `r` (*retry*)

  can be used at any of the four ports (although at the Call port it has no effect). It transfers control back to the Call port of the box. This allows restarting an invocation when, for example, it has left the programmer with some weird result. The state of execution is exactly the same as in the original call (unless the invocation has performed side effects, which will not be undone). When a retry is performed the invocation counter is reset so that counting will continue from the current invocation number regardless of what happened before the retry. This is in accord with the fact that execution has, in operational terms, returned to the state before anything else was called.

If an integer is supplied after the retry command, then this is taken as specifying an invocation number and the system tries to get to the Call port, not of the current box, but of the invocation box specified. It does this by continuously failing until it reaches the right place. Unfortunately this process cannot be guaranteed: it may be the case that the invocation the programmer is looking for has been cut out of the search space by cuts in the program. In this case the system fails to the latest surviving Call port before the correct one.

- `f` (*fail*)

  can be used at any of the four ports (although at the Fail port it has no effect). It transfers control to the Fail port of the box, forcing the invocation to fail prematurely. If an integer is supplied after the command, then this is taken as specifying an invocation number and the system tries to get to the Fail port of the invocation box specified. It does this by continuously failing until it reaches the right place. Unfortunately, as before, this process cannot be guaranteed.
- `d` (*display*)

  displays the current goal using `display/1`. See write (below).
- `p` (*print*)

  re-prints the current goal using `print/1`. Nested structures will be printed to the specified *printdepth* (below).
- `w` (*write*)

  writes the current goal on the terminal using `write/1`.
- `g` (*ancestors*)

  provides a list of ancestors to the current goal, i.e., all goals that are hierarchically above the current goal in the calling sequence. It is always possible to jump to any goal in the ancestor list (by using retry, etc.). If an integer `n` is supplied, then only `n` ancestors will be printed. That is to say, the last `n` ancestors will be printed counting back from the current goal. Each entry in the list is preceded by the invocation number followed by the depth number (as would be given in a trace message).
- `n` (*nodebug*)

  switches the debugger off. Note that this is the correct way to switch debugging off at a trace point. The `@` option cannot be used because it always returns to the debugger.
- `=` (*debugging*)

  outputs information concerning the status of the current debugging session.
- `+` *spy*

  sets a spy-point on the current goal.
- `-` (*nospy*)

  removes the spy-point from the current goal.
- `a` (*abort*)

  causes an abort of the current execution. All the execution states built so far are destroyed and the system is put right back at the top-level of the interpreter. (This is the same as the built-in predicate `abort/0`.)
- `@` (*command*)

  allows calling arbitrary goals. The initial message `| :-` will be output on the terminal, and a command is then read from the terminal and executed as if it was at top-level.
- `u` (*unify*()

  is available at the Call port and gives the option of providing a solution to the goal from the terminal rather than executing the goal. This is convenient, e.g., for providing a "stub" for a predicate that has not yet been written. A prompt `|:` will be output on the terminal, and the solution is then read from the terminal and unified with the goal.

- `<` (*printdepth*)

  sets a limit for the subterm nesting level that is printed in messages. While in the debugger, a printdepth is in effect for limiting the subterm nesting level when printing the current goal. When displaying or writing the current goal, all nesting levels are shown. The limit is initially 10. This command, without arguments, resets the limit to 10. With an argument of `n` the limit is set to `n`.
- `^` (*subterm*)

  sets the subterm to be printed in messages. While at a particular port, a current subterm of the current goal is maintained. It is the current subterm which is displayed, printed, or written when prompting for a debugger command. Used in combination with the printdepth, this provides a means for navigating in the current goal for focusing on the part which is of interest. The current subterm is set to the current goal when arriving at a new port. This command, without arguments, resets the current subterm to the current goal. With an argument of `n` (greater than 0 and less or equal to the number of subterms of the current subterm), the current subterm is replaced by its `n`'th subterm. With an argument of `0`, the current subterm is replaced by its parent term.
- `?` or `h` (*help*)

  displays the table of options given above.

6.7 Calling predicates that are not exported by a module

The Ciao module system does not allow calling predicates which are not exported during debugging. However, as an aid during debuggig, this is allowed (only from the top-level and for modules which are in debug mode or source-level debug mode) using the `call_in_module/2` predicate.

Note that this does not affect analysis or optimization issues, since it only works on modules which are loaded in debug mode or source-level debug mode, i.e. unoptimized.

6.8 Acknowledgements

Originally written by Andrzej Ciepielewski. Minor modifications by Mats Carlsson. Later modifications (17 Dec 87) by Takashi Chikayama (making tracer to use `print/1` rather than `write/1`, temporarily switching debugging flag off while writing trace message and within "break" level). Additional modifications by Kish Shen (May 88): subterm navigation, handle unbound args in `spy/1` and `nospy/1`, trapping arithmetics errors in debug mode. Adapted then to &-Prolog and Ciao by D. Cabeza and included in the Ciao version control system. Extended for source-level debugging by Manuel C. Rodríguez. (See changelog if included in the document for more detailed documentation of the later changes.)

7 Predicates controlling the interactive debugger

Author(s): A. Ciepielewski, M. Carlsson, T. Chikayama, K. Shen, D. Cabeza, M. Rodriguez.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.5#107 (2000/4/5, 9:55:16 CEST)

This library implements predicates which are normally used in the interactive top-level shell to debug programs. A subset of them are available in the embeddable debugger.

7.1 Usage and interface (`debugger`)

- **Library usage:**
  `:- use_module(library(debugger)).`
- **Exports:**
  - *Predicates:*
    `debug_module/1`, `nodebug_module/1`, `debug_module_source/1`, `debug/0`, `nodebug/0`, `trace/0`, `notrace/0`, `spy/1`, `nospy/1`, `nospyall/0`, `breakpt/6`, `nobreakpt/6`, `nobreakall/0`, `list_breakpt/0`, `debugging/0`, `leash/1`, `maxdepth/1`, `call_in_module/2`.
- **Other modules used:**
  - *System library modules:*
    `debugger/debugger_lib`, `format`, `ttyout`, `read`, `write`, `aggregates`, `sort`, `prolog_sys`.

7.2 Documentation on exports (`debugger`)

debug_module/1: PREDICATE

Usage: `debug_module(Module)`

- *Description:* The debugger will take into acount module `Module` (assuming it is loaded in interpreted mode). When issuing this command at the toplevel shell, the compiler is instructed also to set to *interpret* the loading mode of files defining that module and also to mark it as 'modified' so that (re)loading this file or a main file that uses this module will force it to be reloaded for source-level debugging.
- *The following properties should hold at call time:*
  `Module` is an atom. (`basic_props:atm/1`)

nodebug_module/1: PREDICATE

Usage: `nodebug_module(Module)`

- *Description:* The debugger will not take into acount module `Module`. When issuing this command at the toplevel shell, the compiler is instructed also to set to *compile* the loading mode of files defining that module.
- *The following properties should hold at call time:*
  `Module` is an atom. (`basic_props:atm/1`)

debug_module_source/1: PREDICATE

Usage: `debug_module_source(Module)`

- *Description:* The debugger will take into acount module `Module` (assuming it is is loaded in source-level debug mode). When issuing this command at the toplevel shell, the compiler is instructed also to set to *interpret* the loading mode of files defining that module and also to mark it as 'modified' so that (re)loading this file or a main file that uses this module will force it to be reloaded for source-level debugging.
- *The following properties should hold at call time:*
  `Module` is an atom. (`basic_props:atm/1`)

debug/0: PREDICATE

Usage:

- *Description:* Switches the debugger on. The interpreter will stop at all ports of procedure boxes of spied predicates.

nodebug/0: PREDICATE

Usage:

- *Description:* Switches the debugger off. If there are any spy-points set then they will be kept but disabled.

trace/0: PREDICATE

Usage:

- *Description:* Start tracing, switching the debugger on if needed. The interpreter will stop at all leashed ports of procedure boxes of predicates either belonging to debugged modules or called from clauses of debugged modules. A message is printed at each stop point, expecting input from the user (write `h` to see the available options).

notrace/0: PREDICATE

Usage:

- *Description:* Equivalent to `nodebug/0`.

spy/1: PREDICATE

Usage: `spy(PredSpec)`

- *Description:* Set spy-points on predicates belonging to debugged modules and which match `PredSpec`, switching the debugger on if needed. This predicate is defined as a prefix operator by the toplevel.
- *The following properties should hold at call time:*
  `PredSpec` is a sequence of `multpredspecs`. (`basic_props:sequence/2`)

nospy/1: PREDICATE

Usage: `nospy(PredSpec)`

– *Description:* Remove spy-points on predicates belonging to debugged modules which match `PredSpec`. This predicate is defined as a prefix operator by the toplevel.

– *The following properties should hold at call time:*

`PredSpec` is a sequence of `multpredspecs`. (`basic_props:sequence/2`)

nospyall/0: PREDICATE

Usage:

– *Description:* Remove all spy-points.

breakpt/6: PREDICATE

Usage: `breakpt(Pred,Src,Ln0,Ln1,Number,RealLine)`

– *Description:* Set a *breakpoint* in file `Src` between lines `Ln0` and `Ln1` at the literal corresponding to the `Number`'th occurence of (predicate) name `Pred`. The complication comes from the fact that the reader only gives the start and end line numbers for the predicate. The rest of the arguments provide enough information to be able to locate the exact predicate that the `RealLine` line refers to. This is normally not issued by users but rather by the `emacs` mode, which automatically computes the different argument after selecting a point in the source file.

– *The following properties should hold at call time:*

`Pred` is an atom. (`basic_props:atm/1`)

`Src` is a source name. (`streams_basic:sourcename/1`)

`Ln0` is an integer. (`basic_props:int/1`)

`Ln1` is an integer. (`basic_props:int/1`)

`Number` is an integer. (`basic_props:int/1`)

`RealLine` is an integer. (`basic_props:int/1`)

nobreakpt/6: PREDICATE

Usage: `nobreakpt(Pred,Src,Ln0,Ln1,Number,RealLine)`

– *Description:* Remove a breakpoint in file `Src` between lines `Ln0` and `Ln1` at the `Number`'th occurence of (predicate) name `Pred` (see `breakpt/6`). Also normally used from de `emacs` mode.

– *The following properties should hold at call time:*

`Pred` is an atom. (`basic_props:atm/1`)

`Src` is a source name. (`streams_basic:sourcename/1`)

`Ln0` is an integer. (`basic_props:int/1`)

`Ln1` is an integer. (`basic_props:int/1`)

`Number` is an integer. (`basic_props:int/1`)

`RealLine` is an integer. (`basic_props:int/1`)

nobreakall/0: PREDICATE

Usage:

– *Description:* Remove all breakpoints.

list_breakpt/0: PREDICATE

Usage:

– *Description:* Prints out the location of all breakpoints. The location of the breakpoints is showed usual by referring to the source file, the lines between which the predicate can be found, the predicate name and the number of ocurrence of the predicate name of the literal.

debugging/0: PREDICATE

Usage:

– *Description:* Display debugger state.

leash/1: PREDICATE

Usage: `leash(Ports)`

– *Description:* Leash on ports `Ports`, some of `call`, `exit`, `redo`, `fail`. By default, all ports are on leash.

– *The following properties should hold at call time:*

`Ports` is a list of `ports`. (`basic_props:list/2`)

maxdepth/1: PREDICATE

Usage: `maxdepth(MaxDepth)`

– *Description:* Set maximum invocation depth in debugging to `MaxDepth`. Calls to compiled predicates are not included in the computation of the depth.

– *The following properties should hold at call time:*

`MaxDepth` is an integer. (`basic_props:int/1`)

call_in_module/2: PREDICATE

Usage: `call_in_module(Module,Predicate)`

– *Description:* Calls predicate `Predicate` belonging to module `Module`, even if that module does not export the predicate. This only works for modules which are in debug (interpreted) mode (i.e., they are not optimized).

– *The following properties should hold at call time:*

`Module` is an atom. (`basic_props:atm/1`)

`Predicate` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

7.3 Documentation on internals (debugger)

multpredspec/1: PROPERTY

A property, defined as follows:

```
multpredspec(Mod:Spec) :-
        atm(Mod),
        multpredspec(Spec).
multpredspec(Name/Low-High) :-
        atm(Name),
        int(Low),
        int(High).
multpredspec(Name/(Low-High)) :-
        atm(Name),
        int(Low),
        int(High).
multpredspec(Name/Arity) :-
        atm(Name),
        int(Arity).
multpredspec(Name) :-
        atm(Name).
```

8 The script interpreter

Author(s): Daniel Cabeza, Manuel Hermenegildo.

Version: 1.5#133 (2000/5/5, 17:32:14 CEST)

Version of last change: 1.5#130 (2000/5/3, 20:19:4 CEST)

`ciao-shell` is the Ciao script interpreter. It can be used to write *Prolog shell scripts* (see [Her96,CHV96b]), that is, executable files containing source code, which are compiled on demand.

Writing Prolog scripts can sometimes be advantageous with respect to creating binary executables for small- to medium-sized programs that are modified often and perform relatively simple tasks. The advantage is that no compilation is necessary, and thus changes and updates to the program imply only editing the source file. The disadvantage is that startup of the script (the first time after it is modified) is slower than for an application that has been compiled previously.

An area of application is, for example, writing *CGI executables*: the slow speed of the network connection in comparison with that of executing a program makes program execution speed less important has made scripting languages very popular for writing these scripts. Logic languages are, a priori, excellent candidates to be used as scripting languages. For example, the built-in grammars and databases can sometimes greatly simplify many typical script-based applications.

8.1 How it works

Essentially, `ciao-shell` is a smaller version of the Ciao top-level, which starts by loading the file given to it as the first argument and then starts execution at `main/1` (the argument is instantiated to a list containing the command line options, in the usual way). However, while doing this, `ciao-shell` changes the prolog flag `quiet` so that no informational or warning messages are printed (error messages will be reported to `user_error`, however). The operation of `ciao-shell` in Unix-like systems is based in a special compiler feature: when the first character of a file is '`#`', the compiler skips the first lines until an empty line is found.

For example, in a Linux/Unix system, assume a file called `hello` contains the following program:

```
#!/bin/sh
exec ciao-shell $0 "$@" # -*- mode: ciao; -*-

main(_) :-
     write('Hello world'), nl.
```

Then, the file `hello` can be *run* by simply making it executable and invoking it from the command line:

```
/herme@clip:/tmp
[86]> chmod +x hello

/herme@clip:/tmp
[87]> hello
Hello world
```

The line:

```
#!/bin/sh
```

invokes the `/bin/sh` shell which will interpret the following line:

```
exec ciao-shell $0 "$@" # -*- mode: ciao; -*-
```

and invoke `ciao-shell`, instructing it to read this same file (`$0`), passing it the rest of the arguments to `hello` as arguments to the prolog program. The second part of the line `# -*-`

mode: ciao; -*- is simply a comment which is seen by emacs and instructs it to edit this file in Ciao mode (this is needed because these script files typically do not have a .pl ending). When ciao-shell starts, if it is the first time, it compiles the program (skipping the first lines, as explained above), or else at successive runs loads the .po object file, and then calls main/1.

Note that the process of creating Prolog scripts is made very simple by the Ciao emacs mode, which automatically inserts the header and makes the file executable (See Chapter 10 [Using Ciao inside GNU emacs], page 57).

8.2 Command line arguments in scripts

The following example illustrates the use of command-line arguments in scripts. Assume that a file called say contains the following lines:

```
#!/bin/sh
exec ciao-shell $0 "$@" # -*- mode: ciao; -*-

main(Argv) :-
     write_list(Argv), nl.

write_list([]).
write_list([Arg|Args]) :-
     write(Arg),
     write(' '),
     write_list(Args).
```

An example of use is:

```
/herme@clip:/tmp
[91]> say hello dolly
hello dolly
```

9 Customizing library paths and path aliases

Author(s): D.Cabeza.

This library provides means for customizing, from environment variables, the libraries and path aliases known by an executable. Many applications of Ciao, including `ciaoc`, `ciaosh`, and `ciao-shell` make use of this library. Note that if an executable is created dynamic, it will try to load its components at startup, before the procedures of this module can be invoked, so in this case all the components should be in standard locations.

9.1 Usage and interface (`libpaths`)

- **Library usage:**
  `:- use_module(library(libpaths)).`
- **Exports:**
  - *Predicates:*
    `get_alias_path/0`.
  - *Multifiles:*
    `file_search_path/2`, `library_directory/1`.
- **Other modules used:**
  - *System library modules:*
    `system`, `lists`.

9.2 Documentation on exports (`libpaths`)

get_alias_path/0: PREDICATE

`get_alias_path`

Consult the environment variable 'CIAOALIASPATH' and add facts to predicates `library_directory/1` and `file_search_path/2` to define new library paths and path aliases. The format of 'CIAOALIASPATH' is a sequence of paths or alias assignments separated by colons, an alias assignment is the name of the alias, an '=' and the path represented by that alias (no blanks allowed). For example, given

```
CIAOALIASPATH=/home/bardo/ciao:contrib=/usr/local/lib/ciao
```

the predicate will define /home/bardo/ciao as a library path and /usr/local/lib/ciao as the path represented by 'contrib'.

9.3 Documentation on multifiles (`libpaths`)

file_search_path/2: PREDICATE

See Chapter 22 [Basic file/stream handling], page 115.

The predicate is *multifile*.

The predicate is of type *dynamic*.

library_directory/1: PREDICATE

See Chapter 22 [Basic file/stream handling], page 115.

The predicate is *multifile*.

The predicate is of type *dynamic*.

10 Using Ciao inside GNU emacs

Author(s): Manuel Hermenegildo, Manuel C. Rodriguez, Daniel Cabeza, `clip@dia.fi.upm.es`, `http://www.clip.dia.fi.upm.es/`, The CLIP Group, School of Computer Science, Technical University of Madrid.

Version: 1.5#133 (2000/5/5, 17:32:14 CEST)

Version of last change: 1.5#118 (2000/4/19, 18:13:43 CEST)

The Ciao/Prolog emacs interface (or *mode* in `emacs` terms) provides a rich, integrated user interface to the Ciao *program development environment* components, including the `ciaosh` interactive top level and the `ciaopp` preprocessor. While most features of the Ciao development environment are available from the command line of the preprocessor and the top-level shell, using Ciao inside `emacs` is highly recommended. The facilities that this mode provides include:

- *Syntax highlighting* and syntax coloring, *auto-indentation*, *auto-fill*, etc. of code. This includes the assertions used by the preprocessor and the documentation strings used by the Ciao auto-documenter, `lpdoc`.
- Providing automatic access to on-line help for all predicates by accessing the Ciao system manuals in `info` format.
- Starting and communicating with `ciaopp`, the *Ciao preprocessor*, running in its own sub-shell. This allows easily performing certain kinds of *static checks* (useful for finding errors in programs before running them), program analysis tasks, and *program transformations* on source programs.
- Starting and communicating with the *Ciao top-level*, running in its own sub-shell. This facilitates loading programs, checking the *syntax* of programs (and of *assertions* within programs), marking and unmarking modules for interactive debugging, *tracing the source code* during debugging, making stand-alone executables, compiling modules to dynamically linkable Prolog objects, compiling modules to active objects, etc.
- Syntax highlighting and coloring of the error and warning messages produced by the top level, preprocessor, or any other tool using the same message format (such as the `lpdoc` auto-documenter), and *locating automatically the points in the source files where such errors occur.*
- Performing automatic *version control* and keeping a *changelog* of individual files or whole applications. This is done by automatically including changelog entries in source files, which can then be processed by the `lpdoc` auto-documenter.

This chapter explains how to use the Ciao/Prolog `emacs` interface and how to set up your `emacs` environment for correct operation. The Ciao `emacs` interface can also be used to work with other Prolog or CLP systems.

10.1 Conventions for writing Ciao programs under Emacs

This is particularly important for the source-level debugger and the syntax coloring capabilities. This is due to the fact that it would be unrealistic to write a complete Prolog parser in Emacs lisp. These conventions are the following, in order of importance:

- Clauses should begin on the first column (this is used to recognize the beginning of a clause).
- C style comments should not be used in a clause, but can be used outside any clause.

The following are suggestions which are not strictly necessary but which can improve operation:

- Body literals should be indented, and there should be not more than one literal per line. This allows more precision in the location of program points during source-level debugging, i.e., when marking breakpoints and during line tracing.

10.2 Checking the installation

It may be possible that a complete pre-installation of the Ciao/Prolog `emacs` interface was already completed during Ciao installation. To check this, open a file with a `.pl` ending. You should see that `emacs` enters Ciao/Prolog mode: the mode is identified in the status bar below the buffer and, if the emacs menu bar is enabled, you should see the Ciao/Prolog menus. You should be able from the menu-bar, for example, to go to the Ciao manuals in the info or load the `.pl` file that you just opened into a ciao top level.

If things don't work properly, see the section Section 10.8 [Installation of the Ciao/Prolog emacs interface], page 68 latter in this chapter.

10.3 Functionality and associated key sequences (bindings)

The following is a summary of the capabilities of the Ciao/Prolog emacs interface and the (default) *key sequences* used to access those capabilities. Most of these functions are accessible also from the menu bar.

10.3.1 Syntax coloring and syntax based editors

Syntax-based coloring is provided automatically when opening Ciao/Prolog files. The mode should be set to Ciao/Prolog and the Ciao mode menus should appear on the menu bar. Limited syntax-based indentation is also provided:

⟨TAB⟩
: Indent current line as Ciao/Prolog code. With argument, indent any additional lines of the same clause rigidly along with this one.

10.3.2 Getting on-line help

The following commands are useful for getting on-line help. This is done by accessing the `info` version of the Ciao manuals or the `emacs` built-in help strings. Note also that the `info` standard `search` command (generally bound to ⟨s⟩) can be used inside `info` buffers to search for a given string.

⟨^C⟩ ⟨TAB⟩
: Find help for the symbol (e.g., predicate, directive, declaration, type, etc.) that is currently under the cursor. Opens a (hopefully) relevant part of the Ciao manuals in `info` mode. Requires that the Ciao manuals in `info` format be installed and accessible to `emacs` (i.e., they should appear somewhere in the info directory when typing `M-x info`). It also requires `word-help.el`, which is provided with Ciao. Refer to the installation instructions if this is not the case.

⟨^C⟩ ⟨/⟩
: Find a completion for the symbol (e.g., predicate, directive, declaration, type, etc.) that is currently under the cursor. Uses for completion the contents of the indices of the Ciao manuals. Same requirements as for finding help for the symbol.

⟨^C⟩ ⟨^M⟩
: Go to the part of the info directory containing the Ciao manuals.

⟨^H⟩ ⟨m⟩
: Show a short description of the Ciao/Prolog emacs mode, including all key bindings.

10.3.3 Loading and compiling programs

These commands allow *loading programs*, *creating executables*, etc. by issuing the appropriate commands to a Ciao/Prolog top level shell, running in its own buffer as a subprocess. See Chapter 5 [The interactive top-level shell], page 31 for details. The following commands implement the communication with the Ciao/Prolog top level:

^C t
: Ensure that an inferior Ciao/Prolog top-level process is running. Normally, it is not necessary to use this function since execution of any of the other functions related to the top level ensures that a top level is started (starting one if required). This function is useful when one would like to start a top level in order to type a command directly by hand into it.

^C l
: Load the current buffer (and any auxiliary files it may use) into the top level. The type of compilation performed (*compiling* or *interpreting*) is selected automatically depending on whether the buffer has been marked for debugging or not – see below. In case you try to load a file while in the middle of the debugging process the debugger is first aborted and then the buffer is loaded. Also, if there is a defined query, the user is asked whether it should be called.

^C L
: Load the module designed as *main module* (and any auxiliary files it may use) into the top level. If no main module is defined it will load the current buffer. The type of compilation performed (*compiling* or *interpreting*) is selected automatically depending on whether the buffer has been marked for debugging or not – see below. In case you try to load a file while in the middle of the debugging process the debugger is first aborted and then the buffer is loaded. Also, if there is a defined query, the user is asked whether is should be called.

^C q
: Set a default query. It is possible to set things up so that this query will be issued any time a program is (re)loaded. The functionality is available in the major mode (i.e., from a buffer containing a source file) and in the inferior mode (i.e., from the buffer running the top-level shell. When called from the major mode it will ask in the minibuffer for the query and when calling from the inferior mode it will get the query on the current line.

^C Q
: Ask user if the predefined query should be issued.

^C s
: Set the current buffer as the principal module in a multiple module programming environment.

^C x
: Make an executable from the code in the current buffer. The buffer must contain a `main/0` or `main/1` predicate. Note that compiler options can be set to determine whether the libraries and auxiliary files used by the executable will be statically linked, dynamically linked, auto-loaded, etc.

^C o
: Make a Prolog object (.po) file from the code in the current buffer. This is very useful during debugging or program development, when only one or a few files of a large application are modified. If the application executable is dynamically linked, i.e., the component .po files are loaded dynamically during startup of the application, then this command can be used to recompile only the file or files which have changed, and the correct version will be loaded dynamically the next time the application is started. However, note that this only works if the inter-module interfaces have not changed. A safer, but possibly slower way is to generate the executable again, letting the Ciao compiler, which is inherently incremental, determine what needs to be recompiled.

^C a
: Make an active module executable from the code in the current buffer. An active module is a remote procedure call server (see the `activemod` library documentation for details).

10.3.4 Locating errors and checking the syntax of assertions

These commands allow several syntactic checks of assertions, and locating quickly the point in the source code corresponding to errors flagged by the compiler or preprocessor:

⟨^C⟩ ⟨`⟩ Go to the location in the source file containing the next error reported by the last Ciao/Prolog subprocess (preprocessor or toplevel) which was run.

⟨^C⟩ ⟨E⟩ Check the *syntax* of the code and assertions in the current buffer, as well as imports and exports. Note that full (semantic) assertion checking must be done with the preprocessor.

10.3.5 Commands which help in typing in programs

The following commands are intended to help in the process of writing programs:

⟨^C⟩ ⟨I⟩ ⟨S⟩ Insert a (Unix) header at the top of the current buffer so that the ciao script interpreter will be called on this file if *run* from the command line. It also makes the file "executable" (e.g., '`chmod +x <file>`' in Unix). See Chapter 8 [The script interpreter], page 53 for details.

10.3.6 Debugging programs

These commands allow marking modules for *debugging* by issuing the appropiate commands to a Ciao/Prolog top level shell, running in its own buffer as a subprocess. There are two differents types of debugging: traditional Prolog debugging (using the byrd-box model and spy-points) and *source-level debugging* (same as traditional debugging plus source tracing and breakpoints). In order to use *breakpoints*, source debugging must be on. The following commands implement comunication with the Ciao/Prolog top level:

⟨^C⟩ ⟨d⟩ This is a shortcut which is particularly useful when debugging a single module or file. It corresponds to several lower-level actions. Those lower-level actions depend on how the module was selected for debugging. In case the module was not marked for source-level debugging, it marks the module corresponding to the current buffer for source-level debugging, reloads it to make sure that it is loaded in the correct way for debugging (same as C-c l), and sets the debugger in trace mode (i.e., issues the `trace.` command to the top-level shell). Conversely, if the module was already marked for source-level debugging then it will take the opposite actions, i.e., it unmarks the module for source-level debugging, reloads it, and sets the debugger to non-debug mode.

⟨^C⟩ ⟨m⟩ Mark, or unmkark, the current buffer for debugging (traditional debugging or source debugging). Note that if the buffer has already been loaded while it was unmarked for debugging (and has therefore been loaded in "compile" mode) it has to be loaded again. The minibuffer shows how the module is loaded now and allows selecting another mode for it. There are three posibilities: N for no debug, S for source debug and D for traditional debug.

⟨^C⟩ ⟨M-m⟩ Visits all Ciao/Prolog files which are currently open in a buffer allowing selecting for each of them whether to debug them or not and the type of debugging performed. When working on a multiple module program, it is possible to have many modules open at a time. In this case, you will navigate through all open Ciao/Prolog files and select the debug mode for each of them (same as doing C-c m for each).

⟨^C⟩ ⟨S⟩ ⟨b⟩ Set a breakpoint on the current literal (goal). This can be done at any time (while debugging or not). The cursor must be *on the predicate symbol of the literal.* Breakpoints are only useful when using source-level debugging.

⟨^C⟩ ⟨S⟩ ⟨v⟩ Remove a breakpoint from the current literal (goal). This can be done at any time (while debugging or not). The cursor must be *on the predicate symbol of the literal.*

⟨^C⟩ ⟨S⟩ ⟨n⟩ Remove all breakpoints. This can be done at any time (while debugging or not).

^C S l
: Redisplay breakpoints in all Ciao buffers. This ensures that the marks in the source files and the Ciao/Prolog toplevel are synchronized.

^C S r
: Remove breakpoints color in all Ciao/Prolog files.

^C S t
: Set the debugger to the trace state. In this state, the program is executed step by step.

^C S d
: Set the debugger to the debug state. In this state, the program will only stop in breakpoints and spypoints. Breakpoints are specially supported in `emacs` and using source debug.

^C r
: Load the current region (between the cursor and a previous mark) into the top level for debugging. Since loading a region of a file is typically done for debugging and/or testing purposes, this command always loads the region in debugging mode (interpreted).

^C p
: Load the predicate around the cursor into the top level. Since loading a single predicate is typically done for debugging and/or testing purposes, this command always loads the predicate in debugging mode (interpreted).

10.3.7 Preprocessing programs

These commands allow *preprocessing programs* with `ciaopp`, the *Ciao preprocessor*.

`ciaopp` is the precompiler of the Ciao Prolog development environment. `ciaopp` can perform a number of program debugging, analysis and source-to-source transformation tasks on (Ciao) Prolog programs. These tasks include:

- *Inference of properties* of the predicates and literals of the progam, including *types*, *modes* and other *variable instantiation* properties, *non-failure*, *determinacy*, bounds on *computational cost*, bounds on *sizes of terms* in the program, etc.
- Certain kinds of *static debugging*, finding errors before running the program. This includes checking the ways in which programs call the system library predicates and also *checking the assertions* present in the program or in other modules used by the program. Such assertions essentially represent partial *specifications* of the program.
- Several kinds of source to source *program transformations* such as *program specialization*, *program parallelization* (including *granularity control*), inclusion of *run-time tests* for assertions which cannot be checked completely at compile-time, etc.

The information generated by analysis, the assertions in the system libraries, and the assertions optionally included in user programs as specifications are all written in the same *assertion language*, which is in turn also used by the Ciao system documentation generator, `lpdoc`.

`ciaopp` is distributed under the GNU general public license.

See the preprocessor manual for details. The following commands implement the communication with the Ciao preprocessor:

^C M
: Preprocess the buffer, selecting options. Instructs the preprocessor to load the current buffer and start an interactive dialog in which the different options available in the preprocessor can be set.

^C P
: Preprocess the buffer, using the previously selected options. If no options were set previously, then the preprocessor defaults are used.

^C ^V
: Show last output file produced by Ciao preprocessor. The preprocessor works by producing a file which is a transformed and/or adorned (with assertions) version of the input file. This command is often used after running the preprocessor in order to visit the output file and see the results from running the preprocessor.

^C V Preprocess the buffer, using the previously selected (or default) options, waits for preprocessing to finish and displays the preprocessor output (leaving the cursor at the same point if already on a preprocessor output file). This allows running the preprocessor over and over and watching the output while modifying the source code.

^C ^R Ensure that an inferior Ciao preprocessor process is running

10.3.8 Version control

The following commands can be used to carry out a simple form of version control by keeping a log of changes on a file or a group of related files. This log is kept in a format that is understood by `lpdoc`, the Ciao documenter [Her99a]. As a result, if these version comments are present, then `lpdoc` will be able to automatically assign up to date version numbers to the manuals that it generates. This way it is always possible to identify to which version of the software a manual corresponds. Also, `lpdoc` can create automatically sections describing the changes made since previous versions, which are extracted from the comments in the changelog entries.

The main effect of these commands is automatically associate the following information to a set of changes performed in the file and/or in a set of related files:

- a *version number* (such as, e.g., `1.2`, where `1` is the major version number and `2` is the minor version number),
- a patch number (such as, e.g., the `4` in `1.2#4`),
- a time stamp (such as, e.g., `1998/12/14,17:20*28+MET`),
- the author of the change, and
- a comment explaining the change.

The version numbering used can be local to a single file or common to a number of related files. A simple version numbering policy is implemented: when a relevant change is made, the user typically inserts a changelog entry for it, using the appropriate command (or selecting the corresponding option when prompted while saving a file). This will cause the *patch number* for the file (or for the whole system that the file is part of) to be incremented automatically and the corresponding machine-readable comment to be inserted in the file. Major and minor version numbers can also be changed, but this is always invoked by hand (see below).

The changelog entry is written in the form of a `comment/2` declaration. As mentioned before, the advantage of using this kind of changelog entries is that these declarations can be processed by the `lpdoc` automatic documenter (see the `lpdoc` reference manual [Her99a] or the `assertions` library documentation for more details on these declarations).

Whether the user is asked or not to introduce such changelog entries, and how the patch and version numbers should be increased is controlled by the presence in the file of a `comment/2` declaration of the type:

```
:- comment(version_maintenance,<type>).
```

(note that this requires including the `assertions` library in the source file). These declarations themselves are also typically introduced automatically when using this mode (see below).

The version maintenance mode can also be set alternatively by inserting a comment such as:

```
%% Local Variables:
%% mode: ciao
%% update-version-comments: "off"
%% End:
```

The lines above instruct emacs to put the buffer visiting the file in emacs Ciao/Prolog mode and to turn version maintenance off. Setting the version maintenance mode in this way has the disadvantage that `lpdoc`, the auto-documenter, and other related tools will not be aware of the type of version maintenance being performed (the lines above are comments for Prolog). However, this can be useful in fact for setting the *version maintenance mode for packages* and other files meant for inclusion in other files, since that way the settings will not affect the file in which the package is included.

The following commands implement the version control support:

⟨^X⟩ ⟨^S⟩
: This is the standard `emacs` command that saves a buffer by writing the contents into the associated `.pl` file. However, in Ciao/Prolog mode this command can be set to ask the user before saving whether to introduce a changelog entry documenting the changes performed.

  If the buffer does not already contain a comment specifying the type of version control to be performed, and before saving the buffer, the Ciao/Prolog mode prompts the user to choose among the following options:

  ⟨q⟩
  : Turn off prompting for the introduction of changelog entries for now. `emacs` will not ask again while the buffer is loaded, but it will ask again next time you load the buffer.

  ⟨n⟩
  : Turn off version control for this file. A version control comment such as:

    `:- comment(version_maintenance,off).`

    is added to the buffer and the file is saved. `emacs` will not perform any version control on this file until the line above is removed or modified (i.e., from now on C-x C-s simply saves the buffer).

  ⟨y⟩
  : Turn version control on for this file.

  If ⟨y⟩ is selected, then the system prompts again regarding how and where the version and patch number information is to be maintained. The following options are available:

  `on`
  : All version control information will be contained within this file. When saving a buffer (C-x C-s) emacs will ask if a changelog entry should be added to the file before saving. If a comment is entered by the user, a new patch number is assigned to it and the comment is added to the file. This patch number will be the one that follows the most recent changelog entry already in the file. This is obviously useful when maintaining version numbers individually for each file.

  <directory_name>
  : Global version control will be performed coherently on several files. When saving a buffer (C-x C-s) emacs will ask if a changelog entry should be added to the file before saving. If a comment is given, the global patch number (which will be kept in the file: <directory_name>/GlobalPatch) is atomically incremented and the changelog entry is added to the current file, associated to that patch number. Also, a small entry is added to a file <directory_name>/GlobalChangeLog which points to the current file. This allows inspecting all changes sequentially by visiting all the files where the changes were made (see C-c C-n). This is obviously useful when maintaining a single thread of version and patch numbers for a set of files.

  `off`
  : Turns off version control: C-x C-s then simply saves the file as usual.

  Some useful tips:

  - If a changelog entry is in fact introduced, the cursor is left at the point in the file where the comment was inserted and the mark is left at the original file point. This allows inspecting (and possibly modifying) the changelog entry, and then returning to the original point in the file by simply typing C-x C-x.

- The first changelog entry is entered by default at the end of the buffer. Later, the changelog entries can be moved anywhere else in the file. New changelog entries are always inserted just above the first changelog entry which appears in the file.
- The comments in changelog entries can be edited at any time.
- If a changelog entry is moved to another file, and version numbers are shared by several files through a directory, the corresponding file pointer in the <directory_name>/GlobalChangeLog file needs to be changed also, for the entry to be locatable later using C-c C-n.

⟨^C⟩ ⟨^S⟩ Same as C-x C-s except that it forces prompting for inclusion of a changelog entry even if the buffer is unmodified.

⟨^C⟩ ⟨n⟩ Force a move to a new major/minor version number (the user will be prompted for the new numbers). Only applicable if using directory-based version maintenance. Note that otherwise it suffices with introducing a changelog entry in the file and changing its version number by hand.

⟨^C⟩ ⟨^N⟩ When a unique version numbering is being maintained across several files, this command allows inspecting all changes sequentially by visiting all the files in which the changes were made:

- If in a source file, find the next changelog entry in the source file, open in another window the corresponding GlobalChangeLog file, and position the cursor at the corresponding entry. This allows browsing the previous and following changes made, which may perhaps reside in other files in the system.
- If in a GlobalChangeLog file, look for the next entry in the file, and open in another window the source file in which the corresponding comment resides, positioning the corresponding comment at the top of the screen. This allows going through a section of the GlobalChangeLog file checking all the corresponding comments in the different files in which they occur.

10.3.9 Generating program documentation

These commands provide some bindings and facilities for generating the documentation corresponding to the current buffer. This is specially useful while modifying the documentation for a file, in order to check the output that will be produced.

⟨^C⟩ ⟨D⟩ ⟨S⟩ Visit, or create, the SETTINGS file (which controls all auto-documenter options).

⟨^C⟩ ⟨D⟩ ⟨F⟩ Change the default output format used by the LPdoc auto-documenter. It is set by default to dvi or to the environment variable LPDOCFORMAT if it is defined.

⟨^C⟩ ⟨D⟩ ⟨G⟩ Generate the documentation according to SETTINGS in the default format.

⟨^C⟩ ⟨D⟩ ⟨B⟩ Generate the documentation for the current buffer in the default format. Basically, it sets MAIN in SETTINGS to the current buffer and then generates the documentation.

⟨^C⟩ ⟨D⟩ ⟨V⟩ Start a viewer on the documentation for the current buffer in the default format.

⟨^C⟩ ⟨D⟩ ⟨W⟩

Change the root working dir used by the LPdoc auto-documenter. It is set by default to a new dir under /tmp or to the environment variable LPDOCWDIR if it is defined.

10.3.10 Setting the top-level, preprocessor, and documenter executables

These commands allow *changing the executables used* when starting a Prolog top-level, the preprocessor, or the auto-documenter. They also allow changing the arguments that these executables take, and changing the path where the libraries reside. In the case of the top-level and preprocessor, this should be done only by users which understand the implications, but it is very useful if several versions of Ciao/Prolog or the preprocessor are available in the system. All these arguments can be changed through the *customize* options in the help menu (see Section 10.7 [Customization], page 66).

⟨^C⟩ ⟨S⟩ ⟨C⟩ Change the Ciao/Prolog executable used to run the Prolog-like top level. It is set by default to `ciao` or, to the environment variable `CIAO` if it is defined.

⟨^C⟩ ⟨S⟩ ⟨^C⟩
Change the arguments passed to the Ciao/Prolog executable. They are set by default to none or, to the environment variable `CIAOARGS` if it is defined.

⟨^C⟩ ⟨S⟩ ⟨P⟩ Change the executable used to run the Ciao Preprocessor toplevel. It is set by default to `ciaopp` or, to the environment variable `CIAOPP` if it is defined.

⟨^C⟩ ⟨S⟩ ⟨^P⟩
Change the arguments passed to the Ciao preprocessor executable. They are set by default to none or to the environment variable `CIAOPPARGS` if it is defined.

⟨^C⟩ ⟨S⟩ ⟨L⟩ Change the location of the Ciao/Prolog library paths (changes the environment variable `CIAOLIB`).

⟨^C⟩ ⟨S⟩ ⟨D⟩ Change the executable used to run the LPdoc auto-documenter. It is set by default to `lpdoc` or to the environment variable `LPDOC` if it is defined.

⟨^C⟩ ⟨S⟩ ⟨^D⟩
Change the arguments passed to the LPdoc auto-documenter. They are set by default to none or to the environment variable `LPDOCARGS` if it is defined.

⟨^C⟩ ⟨S⟩ ⟨^L⟩
Change the path in which the LPdoc library is installed. It is set by default to `/home/clip/lib` or to the environment variable `LPDOCLIB` if it is defined.

10.3.11 Traditional Prolog Mode Commands

These commands provide some bindings and facilities for loading programs, which are present in emacs Prolog modes of other Prolog systems (e.g., SICStus). This is useful mainly if the Ciao/Prolog emacs mode is used with such Prolog systems. Note that the behavior of these commands in Ciao is slightly different from that of SICStus and their use (`compile/1` and `consult/1`) in the Ciao top-level are not recommended.

⟨^C⟩ ⟨K⟩ Compile the entire buffer.

⟨^C⟩ ⟨k⟩ Compile a given region.

⟨^C⟩ ⟨^K⟩ Compile the predicate around point.

⟨^C⟩ ⟨C⟩ Consult the entire buffer.

⟨^C⟩ ⟨c⟩ Consult a given region.

⟨^C⟩ ⟨^C⟩ Consult the predicate around point.

10.3.12 Other commands

Some other commands which are active in the Ciao/Prolog mode:

⟨^C⟩ ⟨^L⟩ Recenter the most recently used Ciao/Prolog inferior process buffer (top-level or preprocessor).

10.3.13 Getting the Ciao/Prolog mode version

^C v Report the version of the emacs Ciao/Prolog mode.

10.4 Commands available in toplevel and preprocessor buffers

The interactive top level and the preprocessor both are typically run in an iteractive buffer, in which it is possible to communicate with them in the same way as is they had been started from a standard shell. In addition, the commands and key bindings available in buffers containing source code are also available in these interactive buffers, when applicable.

10.5 Using Ciao/Prolog mode capabilities in standard shells

The capabilities (commands, coloring, ...) which are active in the Ciao/Prolog "inferior" mode (the mode of the buffers in which the top level and the preprocessor run) can also be made available in any standard command line shell which is being run within emacs. This can be enabled by going to the buffer in which the shell is running and typing "M-x `ciao-inferior-mode`". This is very useful for example when running the stand-alone compiler, the `lpdoc` auto-documenter, or even certain user applications (those that use the standard error message library) in an emacs sub-shell. Turning the Ciao/Prolog inferior mode on on that sub-shell will highlight and color the error messages, and automatically find and visit the locations in the files in which the errors are reported.

10.6 Coexistence with other Prolog interfaces

As mentioned previously, the Ciao/Prolog `emacs` interface can also be used to work with other Prolog or CLP systems. Also, the Ciao/Prolog `emacs` interface (*mode*) can coexist with other Prolog-related `emacs` interfaces (*modes*) (such as, e.g., the `SICStus` Prolog interface). Only one of the interfaces can be active at a time for a given buffer (i.e., for each given file opened inside `emacs`). In order the change a buffer to a given interface, move the cursor to that buffer and type `M-x ...-mode` (e.g., for the Ciao/Prolog mode, `M-x ciao-mode`).

If several Prolog-related `emacs` interfaces are loaded, then typically the *last* one to be loaded takes precedence, in the sense that this will be the interface in which `emacs` will be set when opening files which have a `.pl` ending (this depends a bit on how things are set up in your `.emacs` file).

10.7 Customization

This section explains all variables used in the Ciao/Prolog emacs mode which can be customized by users. Such customization can be performed (in later versions of `emacs`) from the `emacs` menus (`Help -> Customize -> Top-level Customization Group`), or also by adding a `setq` expression in the `.emacs` file. Such `setq` expression should be similar to:

```
(setq <variable> <new_value>)
```

The following sections list the different variables which can be customized for `ciao`, `ciaopp` and `lpdoc`.

10.7.1 Ciao variables

`ciao-debug-breakpoint-color` (*face*)
Color to use with breakpoints in source debugger.

`ciao-debug-call-color` (*face*)
: Color to use with the call port in source debugger.

`ciao-debug-exit-color` (*face*)
: Color to use with the exit port in source debugger.

`ciao-debug-expansion` (*face*)
: Color to use in source debugger when the predicate was not found.

`ciao-debug-fail-color` (*face*)
: Color to use with the fail port in source debugger.

`ciao-debug-redo-color` (*face*)
: Color to use with the redo port in source debugger.

`ciao-directives-face` (*face*)
: Color to fontify the Ciao/Prolog predicate directives.

`ciao-indent-width` (*integer*)
: Indentation for a new goal.

`ciao-main-buffer` (*string*)
: Name of main module in a multiple module program. It is very useful when reloading from an inferior module because reload is performed from the main module, thus also reloading all dependent modules.

`ciao-ociao-builtin-face` (*face*)
: Color to fontify the Ciao/Prolog builtin directives.

`ciao-ociao-directives-face` (*face*)
: Color to fontify the Ciao/Prolog O'Ciao directives.

`ciao-query` (*string*)
: Query to use in Ciao. Setting this is useful when using a long or complicated query because it saves from having to type it over and over again. It is possible to set that this query will be issued any time a program is (re)loaded.

`ciao-system` (*string*)
: Name of Ciao or Prolog executable which runs the classical Prolog-like top level.

`ciao-system-args` (*string*)
: *Arguments passed to Ciao/Prolog toplevel executable.

`ciao-toplevel-buffer-name` (*string*)
: Basic name of the buffer running the Ciao/Prolog toplevel inferior process.

10.7.2 CiaoPP variables

`ciao-ciaopp-buffer-name` (*string*)
: Basic name of the buffer running the Ciao preprocessor inferior process.

`ciao-ciaopp-system` (*string*)
: Name of Ciao preprocessor executable.

`ciao-ciaopp-system-args` (*string*)
: Arguments passed to Ciao preprocessor executable.

10.7.3 LPdoc variables

`ciao-lpdoc-buffer-name` (*string*)
Basic name of the buffer running the auto-documenter inferior process.

`ciao-lpdoc-docformat` (*symbol*)
Name of default output format used by LPdoc.

`ciao-lpdoc-libpath` (*directory*)
Path in which the LPdoc library is installed.

`ciao-lpdoc-system` (*string*)
Name of LPdoc auto-documenter executable.

`ciao-lpdoc-system-args` (*string*)
Arguments passed to LPdoc executable.

`ciao-lpdoc-wdir-root` (*directory*)
Name of root working dir used by LPdoc.

10.8 Installation of the Ciao/Prolog emacs interface

If opening a file ending with `.pl` puts emacs in another mode (such as `perl` mode, which is the –arguably incorrect– default setting in some `emacs` distributions), then you can set things so that the Ciao/Prolog mode is loaded by default in your system. This can be done by including in your `.emacs` file a line such as:

```
(load <CIAOLIBDIR>/DOTemacs)
```

This loads the above mentioned file from the Ciao library, which contains the following lines (except that the paths are changed during installation to appropriate values for your system):

```
;; Ciao / &-Prolog / Prolog mode
;; (can be used for SICStus, and coexist with SICStus Prolog prolog.el)
;;
(setq load-path (cons "<CIAOLIBDIR>" load-path))
(autoload 'run-ciao-toplevel "ciao"
          "Start a Ciao / &-Prolog / Prolog top-level sub-process." t)
(autoload 'run-ciao-preprocessor "ciao"
          "Start a Ciao / &-Prolog / Prolog preprocessor sub-process." t)
(autoload 'ciao-mode "ciao"
          "Major mode for edit/run Ciao, Prolog, &-Prolog" t)
(autoload 'ciao-inferior-mode "ciao"
          "Major mode for running Ciao, Prolog, lpdoc, etc." t)
(setq auto-mode-alist (cons '("\\.pl$" . ciao-mode) auto-mode-alist))
(setq auto-mode-alist (cons '("\\.pls$" . ciao-mode) auto-mode-alist))
(setq auto-mode-alist (cons '("\\.lpdoc$" . ciao-mode) auto-mode-alist))
(setq completion-ignored-extensions
      (append '(".dep" ".itf" ".po" ".asr" ".cpx")
              completion-ignored-extensions))
;; ----------------------------------------------------------------------
;; In Un*x, the following (or similar) lines should be included in your
;; .cshrc or .profile to find the manuals (the Ciao installation leaves
;; in the Ciao library directory 'DOTcshrc' and 'DOTprofile' files with
;; the right paths which can be included directly in your startup scripts):
;;
;; setenv INFOPATH /usr/local/info:/usr/info:<LPDOCDIR>
```

```
;; ----------------------------------------------------------------------
```

If you would like to configure things in a different way, you can also copy the contents of this file to your `.emacs` file and make the appropriate changes. For example, if you do not want `.pl` files to be put automatically in Ciao/Prolog mode, then comment out (or remove) the line:

`(setq auto-mode-alist ... )`

You will then need to switch manually to Ciao/Prolog mode by typing `M-x ciao-mode` after opening a Prolog file.

If you are able to open the Ciao/Prolog menu but the Ciao manuals are not found or the `ciao` command (the top-level) is not found when loading `.pl` files, the probable cause is that you do not have the Ciao paths in the `INFOPATH` and `MANPATH` *environment variables* (whether these variables are set automatically or not for users depends on how the Ciao system was installed). Under Un*x, you can add these paths easily by including the line:

`source <CIAOLIBDIR>/DOTcshrc`

in your `.login` or `.cshrc` files if you are using `csh` (or `tcsh`, etc.), or, alternatively, the line:

`. <CIAOLIBDIR>/DOTprofile`

in your `.login` or `.profile` files if you are using `sh` (or `bash`, etc.). See the Ciao installation instructions (Chapter 135 [Installing Ciao from the source distribution], page 509 or Chapter 136 [Installing Ciao from a Win32 binary distribution], page 519) for details.

10.9 Emacs version compatibility

This mode is currently being developed within `GNU emacs` version 20.4. It should also work with all other 20.XX and later 19.XX versions.

10.10 Acknowledgments (ciao.el)

This code is derived from the 1993 version of the emacs interface for &-Prolog by M. Hermenegildo, itself derived from the original `prolog.el` by *Masanobu Umeda* with changes by *Johan Andersson*, *Peter Olin*, *Mats Carlsson*, and *Johan Bevemyr* of *SICS*, Sweden. Other changes also by Daniel Cabeza and Manuel C. Rodriguez. See the changelog for details.

PART II - The Ciao basic language (engine)

This part documents the *Ciao basic builtins*. These predefined predicates and declarations are available in every program, unless the `pure` package is used (by using a `:- module(_,_,[pure]).` declaration or `:- use_package(pure).`). These predicates are contained in the `engine` directory within the `lib` library. The rest of the library predicates, including the packages that provide most of the ISO-Prolog builtins, are documented in subsequent parts.

11 The module system

Author(s): Daniel Cabeza and the CLIP Group.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Modularity is a basic notion in a modern computer language. Modules allow dividing programs in several parts, which have its own independent name spaces. The module system in Ciao [CH00b], as in many other Prolog implementations, is procedure based. This means that predicate names are local to a module, but functor/atom names in data are shared.

The predicates visible in a module are the predicates defined in that module, plus the predicates imported from other modules. Only predicates exported by a module can be imported from other modules. The default module of a given predicate name is the local one if the predicate is defined locally, else the last module from which the predicate is imported, having explicit imports priority. To refer to a predicate from a module which is not the default for that predicate the name has to be module qualified. A module qualified predicate name has the form `Module:Predicate` as in the call `debugger:debug_module(M)`. Note that this does not allow having access to predicates not imported, nor defining clauses of other modules.

All predicates defined in files with no module declaration belong to a special module called `user`, and all are implicitly exported. This allows dividing programs in several files without being aware of the module system at all. Note that this feature is only supported for compatibility reasons, being its use discouraged. Many attractive compilation features of Ciao cannot be performed in `user` modules.

The case of multifile predicates (defined with the declaration `multifile/1`) is also special. Multifile predicates can be defined by clauses distributed in several modules, and all modules which define a predicate as multifile can use that predicate. The name space of multifile predicates is independent, as if they belonged to special module `multifile`.

Every `user` or module file imports implicitly a number of modules called builtin modules. They are imported before all other importations of the module, allowing thus redefining any of their predicates (with the exception of `true/0`) by defining local versions or importing them from other modules. Importing explicitly from a builtin module, however, disables the implicit importation of the rest (this feature is used by package `library(pure)` to define pure prolog code).

11.1 Usage and interface (`modules`)

- **Library usage:**
  Modules are an intrinsic feature of Ciao, so nothing special has to be done to use them.

11.2 Documentation on internals (`modules`)

module/3: DECLARATION

Usage: :- `module(Name,Exports,Packages)`.

- *Description:* Declares a module of name `Name` which exports the predicates in `Exports`, and uses the packages in `Packages`. `Name` must match with the name of the file where the module resides, without extension. For each source in `Packages`, a package file is included, as if by an `include/1` declaration. If the source is specified with a path alias, this is the file included, if it is an atom, the library paths are searched. Package files provide functionalities by declaring imports from other modules, defining operators, new declarations, translations of code, etc.

This directive must appear the first in the file.

Also, if the compiler finds an unknown declaration as the first term in a file, the name of the declaration is regarded as a package library to be included, and the arguments of the declaration (if present) are interpreted like the arguments of `module/3`.

– *The following properties hold at call time:*

  `Name` is a module name (an atom). (`modules:modulename/1`)

  `Exports` is a list of `prednames`. (`basic_props:list/2`)

  `Packages` is a list of `sourcenames`. (`basic_props:list/2`)

module/2: DECLARATION

Usage: :- `module(Name,Exports)`.

– *Description:* Same as directive `module/3`, with an implicit package `iso`, which enables to include ISO-Prolog compatible code (compatibility not 100% yet).

– *The following properties hold at call time:*

  `Name` is a module name (an atom). (`modules:modulename/1`)

  `Exports` is a list of `prednames`. (`basic_props:list/2`)

export/1: DECLARATION

Usage 1: :- `export(Pred)`.

– *Description:* Adds `Pred` to the set of exported predicates.

– *The following properties hold at call time:*

  `Pred` is a Name/Arity structure denoting a predicate name:

```
predname(P/A) :-
        atm(P),
        int(A).
```

(`basic_props:predname/1`)

Usage 2: :- `export(Exports)`.

– *Description:* Adds `Exports` to the set of exported predicates.

– *The following properties hold at call time:*

  `Exports` is a list of `prednames`. (`basic_props:list/2`)

use_module/2: DECLARATION

Usage: :- `use_module(Module,Imports)`.

– *Description:* Specifies that this code imports from the module defined in `Module` the predicates in `Imports`. The imported predicates must be exported by the other module.

– *The following properties hold at call time:*

  `Module` is a source name. (`streams_basic:sourcename/1`)

  `Imports` is a list of `prednames`. (`basic_props:list/2`)

use_module/1: DECLARATION

Usage: :- `use_module(Module)`.

- *Description:* Specifies that this code imports from the module defined in `Module` all the predicates exported by it. The previous version with the explicit import list is preferred to this as it minimizes the chances to have to recompile this code if the other module changes.
- *The following properties hold at call time:*

  `Module` is a source name. (`streams_basic:sourcename/1`)

import/2: DECLARATION

Usage: :- `import(Module,Imports)`.

- *Description:* Declares that this code imports from the module with name `Module` the predicates in `Imports`.

  Important note: this declaration is intended to be used when the current module or the imported module is going to be dynamically loaded, and so the compiler does not include the code of the imported module in the current executable (if only because the compiler cannot know the location of the module file at the time of compilation). For the same reason the predicates imported are not checked to be exported by `Module`. Its use in other cases is strongly discouraged, as it disallows many compiler optimizations.
- *The following properties hold at call time:*

  `Module` is a module name (an atom). (`modules:modulename/1`)

  `Imports` is a list of `prednames`. (`basic_props:list/2`)

reexport/2: DECLARATION

Usage: :- `reexport(Module,Preds)`.

- *Description:* Specifies that this code reexports from the module defined in `Module` the predicates in `Preds`. This implies that this module imports from the module defined in `Module` the predicates in `Preds`, an also that this module exports the predicates in `Preds` .
- *The following properties hold at call time:*

  `Module` is a source name. (`streams_basic:sourcename/1`)

  `Preds` is a list of `prednames`. (`basic_props:list/2`)

reexport/1: DECLARATION

Usage: :- `reexport(Module)`.

- *Description:* Specifies that this code reexports from the module defined in `Module` all the predicates exported by it. This implies that this module imports from the module defined in `Module` all the predicates exported by it, an also that this module exports all such predicates .
- *The following properties hold at call time:*

  `Module` is a source name. (`streams_basic:sourcename/1`)

meta_predicate/1: DECLARATION

Usage: :- `meta_predicate MetaSpecs`.

- *Description:* Specifies that the predicates in `MetaSpecs` have arguments which represent predicates and thus have to be module expanded. The directive is not mandatory in programs which do not use modules. This directive is defined as a prefix operator in the compiler.
- *The following properties hold at call time:*
  `MetaSpecs` is a sequence of `metaspecs`. (`basic_props:sequence/2`)

modulename/1: REGTYPE

Usage: `modulename(M)`

- *Description:* `M` is a module name (an atom).

metaspec/1: REGTYPE

A meta-predicate specification for a predicate is the functor of that predicate applied to atoms which represent the kind of module expansion that should be done with the arguments. Possible contents are represented as:

`goal` This argument will be a term denoting a goal (either a simple or complex one) which will be called. For commpatibility reasons it can be named as `:` as well.

`clause` This argument will be a term denoting a clause.

`fact` This argument should be instantiated to a term denoting a fact (head-only clause).

`spec` This argument should be instantiated to a predicate name, as Functor/Arity.

`pred(`N`)` This argument should be instantiated to a predicate construct to be called by means of a `call/`N predicate call (see `call/2`). Thus, it should be an atom equal to the name of a predicate of arity N, or a structure with functor the name of a predicate of arity M (greater than N) and with M-N arguments.

`?,+,-,_` These other values denote that this argument is not module expanded.

Usage: `metaspec(M)`

- *Description:* `M` is a meta-predicate specification.

12 Directives for using code in other files

Author(s): Daniel Cabeza.
Version: 1.5#118 (2000/4/19, 18:13:43 CEST)
Version of last change: 1.3#107 (1999/11/18, 13:6:14 MET)

12.1 Usage and interface (`loading_code`)

- **Library usage:**
  These directives are builtin in Ciao, so nothing special has to be done to use them.

12.2 Documentation on internals (`loading_code`)

ensure_loaded/1: DECLARATION

Usage: :- `ensure_loaded(File)`. ⟨• ISO •⟩

- *Description:* Specifies that the code present in `File` will be included in the executable being prepared, in the `user` module. The file `File` cannot have a module declaration. This directive is intended to be used by programs not divided in modules. Dividing programs into modules is however strongly encouraged, since most of the attractive features of Ciao (such as static debugging and global optimization) are only partially available for `user` modules.
- *The following properties should hold at call time:*
  `File` is a source name. (`streams_basic:sourcename/1`)

include/1: DECLARATION

Usage: :- `include(File)`. ⟨• ISO •⟩

- *Description:* The contents of the file `File` are included in the current program text exactly as if they had been written in place of this directive.
- *The following properties should hold at call time:*
  `File` is a source name. (`streams_basic:sourcename/1`)

use_package/1: DECLARATION

:- `use_package(Package)`.

Specifies the use in this file of the packages defined in `Package`. See the description of the third argument of `module/3` for an explanation of package files.

This directive must appear the first in the file, or just after a `module/3` declaration. A file with no module declaration, in the absence of this directive, uses an implicit package `iso`, which enables to include ISO-Prolog compatible code (compatibility not 100% yet).

Usage 1: :- `use_package(Package)`.

- *The following properties should hold at call time:*
  `Package` is a source name. (`streams_basic:sourcename/1`)

Usage 2: :- `use_package(Package)`.

- *The following properties should hold at call time:*
  `Package` is a list of `sourcename`s. (`basic_props:list/2`)

13 Control constructs/predicates

Author(s): Daniel Cabeza, Manuel Hermenegildo.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.5#66 (2000/3/16, 17:3:30 CET)

This module contains the set of basic control predicates, except the predicates dealing with exceptions, which are in Chapter 23 [Exception handling], page 123.

13.1 Usage and interface (`basiccontrol`)

- **Library usage:**
  These predicates/constructs are builtin in Ciao, so nothing special has to be done to use them. In fact, as they are hardwired in some parts of the system, most of them cannot be redefined.
- **Exports:**
  - *Predicates:*
    `,/2`, `;/2`, `->/2`, `!/0`, `\+/1`, `if/3`, `true/0`, `fail/0`, `repeat/0`, `call/1`.

13.2 Documentation on exports (`basiccontrol`)

,/2: PREDICATE

`P , Q`

Conjunction (`P` *and* `Q`).

Usage 2: ⟨• ISO •⟩

- *The following properties should hold at call time:*

  `P` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

  `Q` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

;/2: PREDICATE

`P ; Q`

Disjunction (`P` *or* `Q`).

Usage 2: ⟨• ISO •⟩

- *The following properties should hold at call time:*

  `P` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

  `Q` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

->/2: PREDICATE

`P -> Q`

If `P` then `Q` else fail, using first solution of `P` only. Also, `(P -> Q ; R)`, if `P` then `Q` else `R`, using first solution of `P` only. No cuts are allowed in `P`.

Usage 2: ⟨• ISO •⟩

– *The following properties should hold at call time:*

`P` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

`Q` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

!/0: PREDICATE

Usage 1: ⟨• ISO •⟩

– *Description:* Commit to any choices taken in the current predicate.

Usage 2: ⟨• ISO •⟩

– *Description:* Commit to any choices taken in the current predicate.

\+/1: PREDICATE

`\+ P`

Goal `P` is not provable (negation by failure). Fails if `P` has a solution, and succeeds otherwise. No cuts are allowed in `P`.

Usage 2: ⟨• ISO •⟩

– *The following properties should hold at call time:*

`P` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

if/3: PREDICATE

`if(P,Q,R)`

If `P` then `Q` else `R`, exploring all solutions of `P`. No cuts are allowed in `P`.

Usage 2:

– *The following properties should hold at call time:*

`P` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

`Q` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

`R` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

true/0: PREDICATE

Usage 1: ⟨• ISO •⟩

– *Description:* Succeed (noop).

Usage 2: ⟨• ISO •⟩

– *Description:* Succeed (noop).

fail/0: PREDICATE

Usage 1: ⟨• ISO •⟩

- *Description:* Fail, backtrack immediately.

Usage 2: ⟨• ISO •⟩

- *Description:* Fail, backtrack immediately.

repeat/0: PREDICATE

Usage 1: ⟨• ISO •⟩

- *Description:* Generates an infinite sequence of backtracking choices.

Usage 2: ⟨• ISO •⟩

- *Description:* Generates an infinite sequence of backtracking choices.

call/1: PREDICATE

`call(G)`

Executes goal `G`, restricting the scope of the cuts to the execution of `G`. Writing a variable `G` in a goal position is equivalent.

Meta-predicate with arguments: `call(goal)`.

Usage 2: ⟨• ISO •⟩

- *The following properties should hold at call time:*

  `G` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

13.3 Documentation on internals (`basiccontrol`)

`|`/2: PREDICATE

An alias for disjunction (when appearing outside a list). The alias is performed when terms are read in.

call/2: PREDICATE

`call(Pred,Arg1)`

There exists a set of builtin predicates of the form `call/N` such that execute predicate `Pred` given arguments `Arg1` ... `ArgX`, if `Pred` has already arguments the new ones are added to the start. This predicate can be written using the special Ciao syntax `Pred(Arg1,...,ArgX)`.

14 Basic builtin directives

Author(s): Daniel Cabeza.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 0.9#11 (1999/3/18, 21:28:24 MET)

This chapter documents the basic builtin directives in Ciao, additional to the documented in other chapters. These directives are natively interpreted by the Ciao compiler (`ciaoc`).

Unlike in other Prolog systems, directives in Ciao are not goals to be *executed* by the compiler or top level. Instead, they are *read* and acted upon by these programs. The advantage of this is that the effect of the directives is consistent for executables, code loaded in the top level, code analyzed by the preprocessor, etc.

As a result, by default only the builtin directives or declarations defined in this manual can be used in user programs. However, it is possible to define new declarations by using the `new_declaration/1` and `new_declaration/2` directives documented in Chapter 27 [Extending the syntax], page 145.

14.1 Usage and interface (`builtin_directives`)

- **Library usage:**
  These directives are builtin in Ciao, so nothing special has to be done to use them.

14.2 Documentation on internals (`builtin_directives`)

multifile/1: DECLARATION

Usage: :- `multifile Predicates`. ⟨• ISO •⟩

- *Description:* Specifies that each predicate in `Predicates` may have clauses in more than one file. Each file that contains clauses for a multifile predicate must contain a directive multifile for the predicate. The directive should precede all clauses of the affected predicates. This directive is defined as a prefix operator in the compiler.
- *The following properties should hold at call time:*
  `Predicates` is a sequence or list of `prednames`. (`basic_props:sequence_or_list/2`)

discontiguous/1: DECLARATION

Usage: :- `discontiguous Predicates`. ⟨• ISO •⟩

- *Description:* Specifies that each predicate in `Predicates` may be defined in this file by clauses which are not in consecutive order. Otherwise, a warning is signaled by the compiler when clauses of a predicate are not consecutive (this behavior is controllable by the prolog flag *discontiguous_warnings*). The directive should precede all clauses of the affected predicates. This directive is defined as a prefix operator in the compiler.
- *The following properties should hold at call time:*
  `Predicates` is a sequence or list of `prednames`. (`basic_props:sequence_or_list/2`)

impl_defined/1: DECLARATION

Usage: :- `impl_defined(Predicates)`.

– *Description:* Specifies that each predicate in `Predicates` is *impl*icitly *defined* in the current prolog source, either because it is a builtin predicate or because it is defined in a C file. Otherwise, a warning is signaled by the compiler when an exported predicate is not defined in the module or imported from other module.

– *The following properties should hold at call time:*

`Predicates` is a sequence or list of `prednames`. (`basic_props:sequence_or_list/2`)

redefining/1: DECLARATION

Usage: :- `redefining(Predicate)`.

– *Description:* Specifies that this module redefines predicate `Predicate`, also imported from other module, or imports it from more than one module. This prevents the compiler giving warnings about redefinitions of that predicate. `Predicate` can be partially (or totally) uninstantiated, to allow disabling those warnings for several (or all) predicates at once.

– *The following properties should hold at call time:*

`Predicate` is *compatible* with `predname` (`basic_props:compat/2`)

initialization/1: DECLARATION

Usage: :- `initialization(Goal)`. ⟨• ISO •⟩

– *Description:* `Goal` will be executed at the start of the execution of any program containing the current code.

– *The following properties should hold at call time:*

`Goal` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

on_abort/1: DECLARATION

Usage: :- `on_abort(Goal)`.

– *Description:* `Goal` will be executed after an abort of the execution of any program containing the current code.

– *The following properties should hold at call time:*

`Goal` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

15 Basic data types and properties

Author(s): Daniel Cabeza, Manuel Hermenegildo.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.3#100 (1999/11/12, 17:14:25 MET)

This library contains the set of basic properties used by the builtin predicates, and which constitute the basic data types and properties of the language. They can be used both as type testing builtins within programs (by calling them explicitly) and as properties in assertions.

15.1 Usage and interface (`basic_props`)

- **Library usage:**
  `:- use_module(library(basic_props)).`
- **Exports:**
  - *Properties:*
    `member/2`, `compat/2`, `iso/1`, `not_further_inst/2`, `regtype/1`.
  - *Regular Types:*
    `term/1`, `int/1`, `nnegint/1`, `flt/1`, `num/1`, `atm/1`, `struct/1`, `gnd/1`, `constant/1`, `callable/1`, `operator_specifier/1`, `list/1`, `list/2`, `sequence/2`, `sequence_or_list/2`, `character_code/1`, `string/1`, `predname/1`, `atm_or_atm_list/1`.

15.2 Documentation on exports (`basic_props`)

term/1: REGTYPE

The most general type (includes all possible terms).

Usage 1: `term(X)`
- *Description:* `X` is any term.

Usage 2: `term(X)`
- *Description:* `X` is any term.

int/1: REGTYPE

The type of integers. The range of integers is `[-2^2147483616, 2^2147483616)`. Thus for all practical purposes, the range of integers can be considered infinite.

Usage 1: `int(T)`
- *Description:* `T` is an integer.

Usage 2: `int(T)`
- *Description:* `T` is an integer.

nnegint/1: REGTYPE

The type of non-negative integers, i.e., natural numbers.

Usage 1: `nnegint(T)`

– *Description:* `T` is a non-negative integer.

Usage 2: `nnegint(T)`

– *Description:* `T` is a non-negative integer.

flt/1: REGTYPE

The type of floating-point numbers. The range of floats is the one provided by the C `double` type, typically `[4.9e-324, 1.8e+308]` (plus or minus).

Usage 1: `flt(T)`

– *Description:* `T` is a float.

Usage 2: `flt(T)`

– *Description:* `T` is a float.

num/1: REGTYPE

The type of numbers, that is, integer or floating-point.

Usage 1: `num(T)`

– *Description:* `T` is a number.

Usage 2: `num(T)`

– *Description:* `T` is a number.

atm/1: REGTYPE

The type of atoms, or non-numeric constants. The name of an atom must be of less than 512 characters.

Usage 1: `atm(T)`

– *Description:* `T` is an atom.

Usage 2: `atm(T)`

– *Description:* `T` is an atom.

struct/1: REGTYPE

The type of compound terms, or terms with non-zeroary functors.

Usage 1: `struct(T)`

– *Description:* `T` is a compound term.

Usage 2: `struct(T)`

– *Description:* `T` is a compound term.

gnd/1: REGTYPE

The type of all terms without variables.

Usage 1: `gnd(T)`

– *Description:* `T` is ground.

Usage 2: `gnd(T)`

– *Description:* `T` is ground.

constant/1: REGTYPE

Usage 1: `constant(T)`

– *Description:* `T` is an atomic term (an atom or a number).

Usage 2: `constant(T)`

– *Description:* `T` is an atomic term (an atom or a number).

callable/1: REGTYPE

Usage 1: `callable(T)`

– *Description:* `T` is a term which represents a goal, i.e., an atom or a structure.

Usage 2: `callable(T)`

– *Description:* `T` is a term which represents a goal, i.e., an atom or a structure.

operator_specifier/1: REGTYPE

The type and associativity of an operator is described by the following mnemonic atoms:

`xfx` Infix, non-associative: it is a requirement that both of the two subexpressions which are the arguments of the operator must be of *lower* precedence than the operator itself.

`xfy` Infix, right-associative: only the first (left-hand) subexpression must be of lower precedence; the right-hand subexpression can be of the *same* precedence as the main operator.

`xfx` Infix, left-associative: same as above, but the other way around.

`fx` Prefix, non-associative: the subexpression must be of *lower* precedence than the operator.

`fy` Prefix, associative: the subexpression can be of the *same* precedence as the operator.

`xf` Postfix, non-associative: the subexpression must be of *lower* precedence than the operator.

`yf` Postfix, associative: the subexpression can be of the *same* precedence as the operator.

Usage 1: `operator_specifier(X)`

– *Description:* `X` specifies the type and associativity of an operator.

Usage 2: `operator_specifier(X)`

– *Description:* `X` specifies the type and associativity of an operator.

list/1: REGTYPE

A list is formed with successive applications of the functor `'.'/2`, and its end is the atom `[]`

Usage 1: `list(L)`

– *Description:* `L` is a list.

Usage 2: `list(L)`

– *Description:* `L` is a list.

list/2: REGTYPE

Meta-predicate with arguments: `list(?,pred(1))`.

Usage 1: `list(L,T)`

- *Description:* `L` is a list of `T`s.

Usage 2: `list(L,T)`

- *Description:* `L` is a list of `T`s.

member/2: PROPERTY

Usage 1: `member(X,L)`

- *Description:* `X` is an element of `L`.

Usage 2: `member(X,L)`

- *Description:* `X` is an element of `L`.

sequence/2: REGTYPE

A sequence is formed with zero, one or more occurrences of the operator `','/2`. For example, `a, b, c` is a sequence of three atoms, `a` is a sequence of one atom.

Meta-predicate with arguments: `sequence(?,pred(1))`.

Usage 1: `sequence(S,T)`

- *Description:* `S` is a sequence of `T`s.

Usage 2: `sequence(S,T)`

- *Description:* `S` is a sequence of `T`s.

sequence_or_list/2: REGTYPE

Meta-predicate with arguments: `sequence_or_list(?,pred(1))`.

Usage 1: `sequence_or_list(S,T)`

- *Description:* `S` is a sequence or list of `T`s.

Usage 2: `sequence_or_list(S,T)`

- *Description:* `S` is a sequence or list of `T`s.

character_code/1: REGTYPE

Usage 1: `character_code(T)`

- *Description:* `T` is an integer which is a character code.
- *The following properties hold upon exit:*

  `T` is an integer. (`basic_props:int/1`)

Usage 2: `character_code(T)`

- *Description:* `T` is an integer which is a character code.
- *The following properties hold upon exit:*

  `T` is an integer. (`basic_props:int/1`)

string/1: REGTYPE

Usage 1: `string(T)`

- *Description:* `T` is a string (a list of character codes).
- *The following properties hold upon exit:*
  `T` is a list of `character_codes`. (`basic_props:list/2`)

Usage 2: `string(T)`

- *Description:* `T` is a string (a list of character codes).
- *The following properties hold upon exit:*
  `T` is a list of `character_codes`. (`basic_props:list/2`)

predname/1: REGTYPE

Usage 1: `predname(P)`

- *Description:* `P` is a Name/Arity structure denoting a predicate name:

```
predname(P/A) :-
        atm(P),
        int(A).
predname(P/A) :-
        atm(P),
        int(A).
```

Usage 2: `predname(P)`

- *Description:* `P` is a Name/Arity structure denoting a predicate name:

```
predname(P/A) :-
        atm(P),
        int(A).
predname(P/A) :-
        atm(P),
        int(A).
```

atm_or_atm_list/1: REGTYPE

Usage 1: `atm_or_atm_list(T)`

- *Description:* `T` is an atom or a list of atoms.

Usage 2: `atm_or_atm_list(T)`

- *Description:* `T` is an atom or a list of atoms.

compat/2: PROPERTY

This property captures the notion of type or property compatibility. The instantiation or constraint state of the term is compatible with the given property, in the sense that assuming that imposing that property on the term does not render the store inconsistent. For example, terms `X` (i.e., a free variable), `[Y|Z]`, and `[Y,Z]` are all compatible with the regular type `list/1`, whereas the terms `f(a)` and `[1|2]` are not.

Meta-predicate with arguments: `compat(?,pred(1))`.

Usage 1: `compat(Term,Prop)`

- *Description:* `Term` is *compatible* with `Prop`

Usage 2: `compat(Term,Prop)`

- *Description:* `Term` is *compatible* with `Prop`

iso/1: PROPERTY

Usage 1: `iso(G)`

– *Description: Complies with the ISO-Prolog standard.*

Usage 2: `iso(G)`

– *Description: Complies with the ISO-Prolog standard.*

not_further_inst/2: PROPERTY

Usage 1: `not_further_inst(G,V)`

– *Description:* `V` is not further instantiated.

Usage 2: `not_further_inst(G,V)`

– *Description:* `V` is not further instantiated.

regtype/1: PROPERTY

Usage 1: `regtype(G)`

– *Description:* Defines a regular type.

Usage 2: `regtype(G)`

– *Description:* Defines a regular type.

16 Extra-logical properties for typing

Author(s): Daniel Cabeza, Manuel Hermenegildo.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 0.7#8 (1998/9/23, 19:21:44 MEST)

This library contains traditional Prolog predicates for testing types. They depend on the state of instantiation of their arguments, thus being of extra-logical nature.

16.1 Usage and interface (`term_typing`)

- **Library usage:**
  These predicates are builtin in Ciao, so nothing special has to be done to use them.
- **Exports:**
  - *Properties:*
    `var/1`, `nonvar/1`, `atom/1`, `integer/1`, `float/1`, `number/1`, `atomic/1`, `ground/1`, `type/2`.

16.2 Documentation on exports (`term_typing`)

var/1: PROPERTY

Usage 1: `var(X)`
- *Description:* `X` is a free variable.
- *The following properties hold globally:*
  `X` is not further instantiated. (`basic_props:not_further_inst/2`)

Usage 2: `var(X)`
- *Description:* `X` is a free variable.
- *The following properties hold globally:*
  `X` is not further instantiated. (`basic_props:not_further_inst/2`)

nonvar/1: PROPERTY

Usage 1: `nonvar(T)`
- *Description:* `T` is currently a term which is not a free variable.
- *The following properties hold globally:*
  `T` is not further instantiated. (`basic_props:not_further_inst/2`)

Usage 2: `nonvar(T)`
- *Description:* `T` is currently a term which is not a free variable.
- *The following properties hold globally:*
  `T` is not further instantiated. (`basic_props:not_further_inst/2`)

atom/1: PROPERTY

Usage 1: `atom(X)`

- *Description:* `X` is currently instantiated to an atom.
- *The following properties hold globally:*
  `X` is not further instantiated. (`basic_props:not_further_inst/2`)

Usage 2: `atom(X)`

- *Description:* `X` is currently instantiated to an atom.
- *The following properties hold globally:*
  `X` is not further instantiated. (`basic_props:not_further_inst/2`)

integer/1: PROPERTY

Usage 1: `integer(X)`

- *Description:* `X` is currently instantiated to an integer.
- *The following properties hold globally:*
  `X` is not further instantiated. (`basic_props:not_further_inst/2`)

Usage 2: `integer(X)`

- *Description:* `X` is currently instantiated to an integer.
- *The following properties hold globally:*
  `X` is not further instantiated. (`basic_props:not_further_inst/2`)

float/1: PROPERTY

Usage 1: `float(X)`

- *Description:* `X` is currently instantiated to a float.
- *The following properties hold globally:*
  `X` is not further instantiated. (`basic_props:not_further_inst/2`)

Usage 2: `float(X)`

- *Description:* `X` is currently instantiated to a float.
- *The following properties hold globally:*
  `X` is not further instantiated. (`basic_props:not_further_inst/2`)

number/1: PROPERTY

Usage 1: `number(X)`

- *Description:* `X` is currently instantiated to a number.
- *The following properties hold globally:*
  `X` is not further instantiated. (`basic_props:not_further_inst/2`)

Usage 2: `number(X)`

- *Description:* `X` is currently instantiated to a number.
- *The following properties hold globally:*
  `X` is not further instantiated. (`basic_props:not_further_inst/2`)

atomic/1: PROPERTY

Usage 1: `atomic(X)`

- *Description:* `X` is currently instantiated to an atom or a number.
- *The following properties hold globally:*
  `X` is not further instantiated. (`basic_props:not_further_inst/2`)

Usage 2: `atomic(X)`

- *Description:* `X` is currently instantiated to an atom or a number.
- *The following properties hold globally:*
  `X` is not further instantiated. (`basic_props:not_further_inst/2`)

ground/1: PROPERTY

Usage 1: `ground(X)`

- *Description:* `X` is currently ground (it contains no variables).
- *The following properties hold globally:*
  `X` is not further instantiated. (`basic_props:not_further_inst/2`)

Usage 2: `ground(X)`

- *Description:* `X` is currently ground (it contains no variables).
- *The following properties hold globally:*
  `X` is not further instantiated. (`basic_props:not_further_inst/2`)

type/2: PROPERTY

Usage 1: `type(X,Y)`

- *Description:* `X` is internally of type `Y` (`var`, `attv`, `float`, `integer`, `structure`, `atom` or `list`).
- *The following properties hold upon exit:*
  `Y` is an atom. (`basic_props:atm/1`)

Usage 2: `type(X,Y)`

- *Description:* `X` is internally of type `Y` (`var`, `attv`, `float`, `integer`, `structure`, `atom` or `list`).
- *The following properties hold upon exit:*
  `Y` is an atom. (`basic_props:atm/1`)

17 Basic term manipulation

Author(s): Daniel Cabeza, Manuel Hermenegildo.
Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

17.1 Usage and interface (`term_basic`)

- **Library usage:**
  These predicates are builtin in Ciao, so nothing special has to be done to use them.
- **Exports:**
  - *Predicates:*
    `arg/3`, `functor/3`, `=../2`, `copy_term/2`, `C/3`.
  - *Properties:*
    `=/2`.

17.2 Documentation on exports (`term_basic`)

=/2: PROPERTY

Usage 1: `X = Y` ⟨• ISO •⟩
- *Description:* Unify `X` and `Y`.

Usage 2: `X = Y` ⟨• ISO •⟩
- *Description:* Unify `X` and `Y`.

arg/3: PREDICATE

Usage 1: `arg(+ArgNo,+Term,?(Arg))` ⟨• ISO •⟩
- *Description:* Argument `ArgNo` of the term `Term` is `Arg`.
- *The following properties should hold at call time:*
  `ArgNo` is currently instantiated to an integer. (`term_typing:integer/1`)

Usage 2: `arg(ArgNo,Term,Arg)` ⟨• ISO •⟩
- *Description:* Argument `ArgNo` of the term `Term` is `Arg`.
- *The following properties should hold at call time:*
  `ArgNo` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Term` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `ArgNo` is currently instantiated to an integer. (`term_typing:integer/1`)

functor/3: PREDICATE

Usage 1: `functor(?(Term),?(Name),?(Arity))` ⟨• ISO •⟩
- *Description:* The principal functor of the term `Term` has name `Name` and arity `Arity`.

Usage 2: `functor(Term,Name,Arity)` ⟨• ISO •⟩
- *Description:* The principal functor of the term `Term` has name `Name` and arity `Arity`.

=../2: PREDICATE

Usage 1: `?(Term) =.. ?(List)` ⟨• ISO •⟩

– *Description:* The functor and arguments of the term `Term` comprise the list `List`.

Usage 2: `Term =.. List` ⟨• ISO •⟩

– *Description:* The functor and arguments of the term `Term` comprise the list `List`.

copy_term/2: PREDICATE

Usage 1: `copy_term(Term,Copy)` ⟨• ISO •⟩

– *Description:* `Copy` is a renaming of `Term`, such that brand new variables have been substituted for all variables in `Term`. If any of the variables of `Term` have attributes, the copied variables will have copies of the attributes as well. It behaves as if defined by:

```
:- data 'copy of'/1.

copy_term(X, Y) :-
        asserta_fact('copy of'(X)),
        retract_fact('copy of'(Y)).
```

Usage 2: `copy_term(Term,Copy)` ⟨• ISO •⟩

– *Description:* `Copy` is a renaming of `Term`, such that brand new variables have been substituted for all variables in `Term`. If any of the variables of `Term` have attributes, the copied variables will have copies of the attributes as well. It behaves as if defined by:

```
:- data 'copy of'/1.

copy_term(X, Y) :-
        asserta_fact('copy of'(X)),
        retract_fact('copy of'(Y)).
```

C/3: PREDICATE

Usage 1: `C(?(S1),?(Terminal),?(S2))`

– *Description:* `S1` is connected by the terminal `Terminal` to `S2`. Used in *DCG grammar rules*. Defined as if by the single clause: `'C'([X|S], X, S).`

Usage 2: `C(S1,Terminal,S2)`

– *Description:* `S1` is connected by the terminal `Terminal` to `S2`. Used in *DCG grammar rules*. Defined as if by the single clause: `'C'([X|S], X, S).`

18 Comparing terms

Author(s): Daniel Cabeza, Manuel Hermenegildo.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

These built-in predicates are extra-logical. They treat uninstantiated variables as objects with values which may be compared, and they never instantiate those variables. They should *not* be used when what you really want is arithmetic comparison or unification.

The predicates make reference to a *standard total ordering* of terms, which is as follows:

- Variables, by age (roughly, oldest first – the order is *not* related to the names of variables).
- Floats, in numeric order (e.g. -1.0 is put before 1.0).
- Integers, in numeric order (e.g. -1 is put before 1).
- Atoms, in alphabetical (i.e. character code) order.
- Compound terms, ordered first by arity, then by the name of the principal functor, then by the arguments in left-to-right order. Recall that lists are equivalent to compound terms with principal functor `'.'/2`.

For example, here is a list of terms in standard order:

```
[ X, -1.0, -9, 1, bar, foo, [1], X = Y, foo(0,2), bar(1,1,1) ]
```

18.1 Usage and interface (term_compare)

- **Library usage:**
  These predicates are builtin in Ciao, so nothing special has to be done to use them.
- **Exports:**
  - *Predicates:*
    `compare/3`.
  - *Properties:*
    `==/2`, `\==/2`, `@</2`, `@=</2`, `@>/2`, `@>=/2`.

18.2 Documentation on exports (term_compare)

==/2: PROPERTY

Usage 1: `Term1 == Term2`

- *Description:* The terms `Term1` and `Term2` are strictly identical.
- *The following properties should hold globally:*
  `Term1` is not further instantiated. (`basic_props:not_further_inst/2`)
  `Term2` is not further instantiated. (`basic_props:not_further_inst/2`)

Usage 2: `Term1 == Term2`

- *Description:* The terms `Term1` and `Term2` are strictly identical.
- *The following properties should hold globally:*
  `Term1` is not further instantiated. (`basic_props:not_further_inst/2`)
  `Term2` is not further instantiated. (`basic_props:not_further_inst/2`)

\==/2: PROPERTY

Usage 1: `Term1 \== Term2`

- *Description:* The terms `Term1` and `Term2` are not strictly identical.
- *The following properties should hold globally:*
  `Term1` is not further instantiated. (`basic_props:not_further_inst/2`)
  `Term2` is not further instantiated. (`basic_props:not_further_inst/2`)

Usage 2: `Term1 \== Term2`

- *Description:* The terms `Term1` and `Term2` are not strictly identical.
- *The following properties should hold globally:*
  `Term1` is not further instantiated. (`basic_props:not_further_inst/2`)
  `Term2` is not further instantiated. (`basic_props:not_further_inst/2`)

@</2: PROPERTY

Usage 1: `@<(Term1,Term2)`

- *Description:* The term `Term1` precedes the term `Term2` in the standard order.
- *The following properties should hold globally:*
  `Term1` is not further instantiated. (`basic_props:not_further_inst/2`)
  `Term2` is not further instantiated. (`basic_props:not_further_inst/2`)

Usage 2: `@<(Term1,Term2)`

- *Description:* The term `Term1` precedes the term `Term2` in the standard order.
- *The following properties should hold globally:*
  `Term1` is not further instantiated. (`basic_props:not_further_inst/2`)
  `Term2` is not further instantiated. (`basic_props:not_further_inst/2`)

@=</2: PROPERTY

Usage 1: `@=<(Term1,Term2)`

- *Description:* The term `Term1` precedes or is identical to the term `Term2` in the standard order.
- *The following properties should hold globally:*
  `Term1` is not further instantiated. (`basic_props:not_further_inst/2`)
  `Term2` is not further instantiated. (`basic_props:not_further_inst/2`)

Usage 2: `@=<(Term1,Term2)`

- *Description:* The term `Term1` precedes or is identical to the term `Term2` in the standard order.
- *The following properties should hold globally:*
  `Term1` is not further instantiated. (`basic_props:not_further_inst/2`)
  `Term2` is not further instantiated. (`basic_props:not_further_inst/2`)

@>/2: PROPERTY

Usage 1: `@>(Term1,Term2)`

- *Description:* The term `Term1` follows the term `Term2` in the standard order.

- *The following properties should hold globally:*
  Term1 is not further instantiated. (basic_props:not_further_inst/2)
  Term2 is not further instantiated. (basic_props:not_further_inst/2)

Usage 2: `@>(Term1,Term2)`
- *Description:* The term `Term1` follows the term `Term2` in the standard order.
- *The following properties should hold globally:*
  Term1 is not further instantiated. (basic_props:not_further_inst/2)
  Term2 is not further instantiated. (basic_props:not_further_inst/2)

@>=/2: PROPERTY

Usage 1: `@>=(Term1,Term2)`
- *Description:* The term `Term1` follows or is identical to the term `Term2` in the standard order.
- *The following properties should hold globally:*
  Term1 is not further instantiated. (basic_props:not_further_inst/2)
  Term2 is not further instantiated. (basic_props:not_further_inst/2)

Usage 2: `@>=(Term1,Term2)`
- *Description:* The term `Term1` follows or is identical to the term `Term2` in the standard order.
- *The following properties should hold globally:*
  Term1 is not further instantiated. (basic_props:not_further_inst/2)
  Term2 is not further instantiated. (basic_props:not_further_inst/2)

compare/3: PREDICATE

`compare(Op,Term1,Term2)`

`Op` is the result of comparing the terms `Term1` and `Term2`.

Usage 1: `compare(?(atm),@(term),@(term))`
- *The following properties should hold upon exit:*
  ?(atm) is an element of [=,>,<]. (basic_props:member/2)
  @(term) is any term. (basic_props:term/1)
  @(term) is any term. (basic_props:term/1)

Usage 2:
- *Call and exit should be compatible with:*
  Op is an atom. (basic_props:atm/1)
- *The following properties should hold at call time:*
  Term1 is any term. (basic_props:term/1)
  Term2 is any term. (basic_props:term/1)
- *The following properties should hold upon exit:*
  Op is an atom. (basic_props:atm/1)
  Term1 is any term. (basic_props:term/1)
  Term2 is any term. (basic_props:term/1)
  Op is an element of [=,>,<]. (basic_props:member/2)
  Term1 is any term. (basic_props:term/1)
  Term2 is any term. (basic_props:term/1)

- *The following properties should hold globally:*
  `Term1` is not further instantiated. (`basic_props:not_further_inst/2`)
  `Term2` is not further instantiated. (`basic_props:not_further_inst/2`)

19 Basic predicates handling names of constants

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

The Ciao system provides builtin predicates which allow dealing with names of constants (atoms or numbers). As an atom name must be of less than 512 characters, to handle sequences of more characters, strings (character code lists) must be used.

19.1 Usage and interface (`atomic_basic`)

- **Library usage:**
  These predicates are builtin in Ciao, so nothing special has to be done to use them.
- **Exports:**
  - *Predicates:*
    `name/2`, `atom_codes/2`, `number_codes/2`, `number_codes/3`, `atom_length/2`, `atom_concat/3`, `sub_atom/4`.

19.2 Documentation on exports (`atomic_basic`)

name/2: PREDICATE

`name(Const,String)`

`String` is the list of the ASCII codes of the characters comprising the name of `Const`. Note that if `Const` is an atom whose name can be interpreted as a number (e.g. '96'), the predicate is not reversible, as that atom will not be constructed when `Const` is uninstantiated. Thus it is recommended that new programs use the ISO-compliant predicates `atom_codes/2` or `number_codes/2`, as these predicates do not have this inconsistency.

Usage 2: `name(-(constant),+string)`

- *Description:* If `String` can be interpreted as a number, `Const` is unified with that number, otherwise with the atom whose name is `String`.

Usage 3:

- *Calls should, and exit will be compatible with:*
  `String` is a string (a list of character codes). (`basic_props:string/1`)
- *The following properties should hold at call time:*
  `Const` is an atomic term (an atom or a number). (`basic_props:constant/1`)
- *The following properties hold upon exit:*
  `String` is a string (a list of character codes). (`basic_props:string/1`)

Usage 4:

- *Description:* If `String` can be interpreted as a number, `Const` is unified with that number, otherwise with the atom whose name is `String`.
- *The following properties should hold at call time:*
  `Const` is a free variable. (`term_typing:var/1`)
  `String` is a string (a list of character codes). (`basic_props:string/1`)
- *The following properties hold upon exit:*
  `Const` is an atomic term (an atom or a number). (`basic_props:constant/1`)

atom_codes/2: PREDICATE

`atom_codes(Atom,String)`

`String` is the list of the ASCII codes of the characters comprising the name of `Atom`.

Usage 3: ⟨• ISO •⟩

- *Calls should, and exit will be compatible with:*
  `String` is a string (a list of character codes). (`basic_props:string/1`)
- *The following properties should hold at call time:*
  `Atom` is an atom. (`basic_props:atm/1`)
- *The following properties hold upon exit:*
  `String` is a string (a list of character codes). (`basic_props:string/1`)

Usage 4: ⟨• ISO •⟩

- *The following properties should hold at call time:*
  `Atom` is a free variable. (`term_typing:var/1`)
  `String` is a string (a list of character codes). (`basic_props:string/1`)
- *The following properties hold upon exit:*
  `Atom` is an atom. (`basic_props:atm/1`)

number_codes/2: PREDICATE

`number_codes(Number,String)`

`String` is the list of the ASCII codes of the characters comprising a representation of `Number`.

Usage 3: ⟨• ISO •⟩

- *Calls should, and exit will be compatible with:*
  `String` is a string (a list of character codes). (`basic_props:string/1`)
- *The following properties should hold at call time:*
  `Number` is a number. (`basic_props:num/1`)
- *The following properties hold upon exit:*
  `String` is a string (a list of character codes). (`basic_props:string/1`)

Usage 4: ⟨• ISO •⟩

- *The following properties should hold at call time:*
  `Number` is a free variable. (`term_typing:var/1`)
  `String` is a string (a list of character codes). (`basic_props:string/1`)
- *The following properties hold upon exit:*
  `Number` is a number. (`basic_props:num/1`)

number_codes/3: PREDICATE

`number_codes(Number,String,Base)`

`String` is the list of the ASCII codes of the characters comprising a representation of `Number` in base `Base`.

Usage 3:

- *Calls should, and exit will be compatible with:*
  `String` is a string (a list of character codes). (`basic_props:string/1`)

- *The following properties should hold at call time:*
  Number is a number. (basic_props:num/1)
  Base is an integer. (basic_props:int/1)
- *The following properties hold upon exit:*
  String is a string (a list of character codes). (basic_props:string/1)

Usage 4:
- *The following properties should hold at call time:*
  Number is a free variable. (term_typing:var/1)
  String is a string (a list of character codes). (basic_props:string/1)
  Base is an integer. (basic_props:int/1)
- *The following properties hold upon exit:*
  Number is a number. (basic_props:num/1)

atom_length/2: PREDICATE

`atom_length(Atom,Length)`

Length is the number of characters forming the name of Atom.

Usage 2: ⟨• ISO •⟩
- *Calls should, and exit will be compatible with:*
  Length is an integer. (basic_props:int/1)
- *The following properties should hold at call time:*
  Atom is an atom. (basic_props:atm/1)
- *The following properties hold upon exit:*
  Length is an integer. (basic_props:int/1)

atom_concat/3: PREDICATE

`atom_concat(Atom_1,Atom_2,Atom_12)`

Atom_12 is the result of concatenating Atom_1 followed by Atom_2.

Usage 1: `atom_concat(+atom,+atom,?(atom))` ⟨• ISO •⟩
- *Description:* Concatenate two atoms.

Usage 2: `atom_concat(-(atom),-(atom),+atom)` ⟨• ISO •⟩
- *Description:* Non-deterministically split an atom.

Usage 3: `atom_concat(-(atom),+atom,+atom)` ⟨• ISO •⟩
- *Description:* Take out of an atom a certain suffix (or fail if it cannot be done).

Usage 4: `atom_concat(+atom,-(atom),+atom)` ⟨• ISO •⟩
- *Description:* Take out of an atom a certain prefix (or fail if it cannot be done).

Usage 5: ⟨• ISO •⟩
- *Description:* Concatenate two atoms.
- *Calls should, and exit will be compatible with:*
  Atom_12 is currently instantiated to an atom. (term_typing:atom/1)
- *The following properties should hold at call time:*
  Atom_1 is currently instantiated to an atom. (term_typing:atom/1)
  Atom_2 is currently instantiated to an atom. (term_typing:atom/1)

- *The following properties hold upon exit:*
  `Atom_12` is currently instantiated to an atom. (`term_typing:atom/1`)

Usage 6: ⟨• ISO •⟩

- *Description:* Non-deterministically split an atom.
- *The following properties should hold at call time:*
  `Atom_1` is a free variable. (`term_typing:var/1`)
  `Atom_2` is a free variable. (`term_typing:var/1`)
  `Atom_12` is currently instantiated to an atom. (`term_typing:atom/1`)
- *The following properties hold upon exit:*
  `Atom_1` is currently instantiated to an atom. (`term_typing:atom/1`)
  `Atom_2` is currently instantiated to an atom. (`term_typing:atom/1`)

Usage 7: ⟨• ISO •⟩

- *Description:* Take out of an atom a certain suffix (or fail if it cannot be done).
- *The following properties should hold at call time:*
  `Atom_1` is a free variable. (`term_typing:var/1`)
  `Atom_2` is currently instantiated to an atom. (`term_typing:atom/1`)
  `Atom_12` is currently instantiated to an atom. (`term_typing:atom/1`)
- *The following properties hold upon exit:*
  `Atom_1` is currently instantiated to an atom. (`term_typing:atom/1`)

Usage 8: ⟨• ISO •⟩

- *Description:* Take out of an atom a certain prefix (or fail if it cannot be done).
- *The following properties should hold at call time:*
  `Atom_1` is currently instantiated to an atom. (`term_typing:atom/1`)
  `Atom_2` is a free variable. (`term_typing:var/1`)
  `Atom_12` is currently instantiated to an atom. (`term_typing:atom/1`)
- *The following properties hold upon exit:*
  `Atom_2` is currently instantiated to an atom. (`term_typing:atom/1`)

sub_atom/4: PREDICATE

`sub_atom(Atom,Before,Length,Sub_atom)`

`Sub_atom` is formed with `Length` consecutive characters of `Atom` after the `Before` character.

Usage 2:

- *Calls should, and exit will be compatible with:*
  `Sub_atom` is an atom. (`basic_props:atm/1`)
- *The following properties should hold at call time:*
  `Atom` is an atom. (`basic_props:atm/1`)
  `Before` is currently instantiated to an integer. (`term_typing:integer/1`)
  `Length` is currently instantiated to an integer. (`term_typing:integer/1`)
- *The following properties hold upon exit:*
  `Sub_atom` is an atom. (`basic_props:atm/1`)

20 Arithmetic

Author(s): Daniel Cabeza, Manuel Hermenegildo.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 0.9#18 (1999/3/23, 21:6:13 MET)

Arithmetic is performed by built-in predicates which take as arguments arithmetic expressions (see `arithexpression/1`) and evaluate them. Terms representing arithmetic expressions can be created dynamically, but at the time of evaluation, each variable in an arithmetic expression must be bound to a non-variable expression (the term must be ground). For example, given the code in the first line a possible shell interaction follows:

```
evaluate(Expression, Answer) :- Answer is Expression.

?- _X=24*9, evaluate(_X+6, Ans).

Ans = 222 ?

yes
```

20.1 Usage and interface (arithmetic)

- **Library usage:**
  These predicates are builtin in Ciao, so nothing special has to be done to use them.
- **Exports:**
  - *Predicates:*
    `is/2`, `</2`, `=</2`, `>/2`, `>=/2`, `=:=/2`, `=\=/2`.
  - *Regular Types:*
    `arithexpression/1`.

20.2 Documentation on exports (arithmetic)

is/2: PREDICATE

`Val is Exp`

The arithmetic expression `Exp` is evaluated and the result is unified with `Val`

Usage 2: ⟨• ISO •⟩

- *Calls should, and exit will be compatible with:*
  `Val` is any term. (`basic_props:term/1`)
- *The following properties should hold at call time:*
  `Exp` is an arithmetic expression. (`arithmetic:arithexpression/1`)
- *The following properties hold upon exit:*
  `Val` is any term. (`basic_props:term/1`)

</2: PREDICATE

`Exp1 < Exp2`

The numeric value of `Exp1` is less than the numeric value of `Exp2` when both are evaluated as arithmetic expressions.

Usage 2: ⟨ • ISO • ⟩

– *The following properties should hold at call time:*

`Exp1` is an arithmetic expression. (`arithmetic:arithexpression/1`)

`Exp2` is an arithmetic expression. (`arithmetic:arithexpression/1`)

=</2: PREDICATE

`Exp1 =< Exp2`

The numeric value of `Exp1` is less than or equal to the numeric value of `Exp2` when both are evaluated as arithmetic expressions.

Usage 2: ⟨ • ISO • ⟩

– *The following properties should hold at call time:*

`Exp1` is an arithmetic expression. (`arithmetic:arithexpression/1`)

`Exp2` is an arithmetic expression. (`arithmetic:arithexpression/1`)

>/2: PREDICATE

`Exp1 > Exp2`

The numeric value of `Exp1` is greater than the numeric value of `Exp2` when both are evaluated as arithmetic expressions.

Usage 2: ⟨ • ISO • ⟩

– *The following properties should hold at call time:*

`Exp1` is an arithmetic expression. (`arithmetic:arithexpression/1`)

`Exp2` is an arithmetic expression. (`arithmetic:arithexpression/1`)

>=/2: PREDICATE

`Exp1 >= Exp2`

The numeric value of `Exp1` is greater than or equal to the numeric value of `Exp2` when both are evaluated as arithmetic expressions.

Usage 2: ⟨ • ISO • ⟩

– *The following properties should hold at call time:*

`Exp1` is an arithmetic expression. (`arithmetic:arithexpression/1`)

`Exp2` is an arithmetic expression. (`arithmetic:arithexpression/1`)

=:=/2: PREDICATE

`Exp1 =:= Exp2`

The numeric values of `Exp1` and `Exp2` are equal when both are evaluated as arithmetic expressions.

Usage 2: ⟨ • ISO • ⟩

– *The following properties should hold at call time:*

`Exp1` is an arithmetic expression. (`arithmetic:arithexpression/1`)

`Exp2` is an arithmetic expression. (`arithmetic:arithexpression/1`)

=\=/2: PREDICATE

`Exp1 =\= Exp2`

The numeric values of `Exp1` and `Exp2` are not equal when both are evaluated as arithmetic expressions.

Usage 2: ⟨• ISO •⟩

– *The following properties should hold at call time:*

`Exp1` is an arithmetic expression. (`arithmetic:arithexpression/1`)

`Exp2` is an arithmetic expression. (`arithmetic:arithexpression/1`)

arithexpression/1: REGTYPE

An arithmetic expression is a term built from numbers and evaluable functors that represent arithmetic functions. An arithmetic expression evaluates to a number, which may be an integer (`int/1`) or a float (`flt/1`). The evaluable functors allowed in an arithmetic expression are listed below, together with an indication of the functions they represent. All evaluable functors defined in ISO-Prolog are implemented, as well as some other useful or traditional. Unless stated otherwise, an expression evaluates to a float if any of its arguments is a float, otherwise to an integer.

- `- /1`: sign reversal. ⟨• ISO •⟩
- `+ /1`: identity.
- `-- /1`: decrement by one.
- `++ /1`: increment by one.
- `+ /2`: addition. ⟨• ISO •⟩
- `- /2`: subtraction. ⟨• ISO •⟩
- `* /2`: multiplication. ⟨• ISO •⟩
- `// /2`: integer division. Float arguments are truncated to integers, result always integer. ⟨• ISO •⟩
- `/ /2`: division. Result always float. ⟨• ISO •⟩
- `rem/2`: integer remainder. The result is always an integer, its sign is the sign of the first argument. ⟨• ISO •⟩
- `mod/2`: modulo. The result is always a positive integer. ⟨• ISO •⟩
- `abs/1`: absolute value. ⟨• ISO •⟩
- `sign/1`: sign of. ⟨• ISO •⟩
- `float_integer_part/1`: float integer part. Result always float. ⟨• ISO •⟩
- `float_fractional_part/1`: float fractional part. Result always float. ⟨• ISO •⟩
- `truncate/1`: The result is the integer equal to the integer part of the argument. ⟨• ISO •⟩
- `integer/1`: same as `truncate/1`.
- `float/1`: conversion to float. ⟨• ISO •⟩
- `floor/1`: largest integer not greater than. ⟨• ISO •⟩
- `round/1`: integer nearest to. ⟨• ISO •⟩
- `ceiling/1`: smallest integer not smaller than. ⟨• ISO •⟩
- `** /2`: exponentiation. Result always float. ⟨• ISO •⟩
- `>> /2`: integer bitwise right shift. ⟨• ISO •⟩
- `<< /2`: integer bitwise left shift. ⟨• ISO •⟩
- `/\ /2`: integer bitwise and. ⟨• ISO •⟩

- `\/ /2`: integer bitwise or. ⟨• ISO •⟩
- `\ /1`: integer bitwise complement. ⟨• ISO •⟩
- `# /2`: integer bitwise exclusive or (xor).
- `exp/1`: exponential (e to the power of). Result always float. ⟨• ISO •⟩
- `log/1`: natural logarithm (base e). Result always float. ⟨• ISO •⟩
- `sqrt/1`: square root. Result always float. ⟨• ISO •⟩
- `sin/1`: sine. Result always float. ⟨• ISO •⟩
- `cos/1`: cosine. Result always float. ⟨• ISO •⟩
- `atan/1`: arc tangent. Result always float. ⟨• ISO •⟩
- `gcd/2`: Greatest common divisor. Arguments must evaluate to integers, result always integer.

In addition to these functors, a list of just a number evaluates to this number. Since a quoted string is just a list of integers, this allows a quoted character to be used in place of its ASCII code; e.g. `"A"` behaves within arithmetic expressions as the integer 65. Note that this is not ISO-compliant, and that can be achieved by using the notation `0'A` (which is ISO).

Arithmetic expressions, as described above, are just data structures. If you want one evaluated you must pass it as an argument to one of the arithmetic predicates defined here.

Usage 1: `arithexpression(E)`

– *Description:* `E` is an arithmetic expression.

Usage 2: `arithexpression(E)`

– *Description:* `E` is an arithmetic expression.

21 Basic input/output

Author(s): Daniel Cabeza, Mats Carlsson.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

This module provides predicates for character input/output and for canonical term output. From the ISO-Prolog predicates for character input/output, only the `_code` versions are provided, the rest are given by `library(iso_byte_char)`, using these. Most predicates are provided in two versions: one that specifies the input or output stream as the first argument and a second which omits this argument and uses the current input or output stream.

21.1 Usage and interface (io_basic)

- **Library usage:**
  These predicates are builtin in Ciao, so nothing special has to be done to use them.
- **Exports:**
  - *Predicates:*
    `get_code/2`, `get_code/1`, `get1_code/2`, `get1_code/1`, `peek_code/2`, `peek_code/1`, `skip_code/2`, `skip_code/1`, `put_code/2`, `put_code/1`, `nl/1`, `nl/0`, `tab/2`, `tab/1`, `code_class/2`, `getct/2`, `getct1/2`, `display/2`, `display/1`, `displayq/2`, `displayq/1`.

21.2 Documentation on exports (io_basic)

get_code/2: PREDICATE

`get_code(Stream,Code)`

Reads from `Stream` the next character and unifies `Code` with its character code. At end of stream, unifies `Code` with the integer -1.

Usage 2: ⟨• ISO •⟩

- *Calls should, and exit will be compatible with:*
  `Code` is an integer. (`basic_props:int/1`)
- *The following properties should hold at call time:*
  `Stream` is an open stream. (`streams_basic:stream/1`)
- *The following properties hold upon exit:*
  `Code` is an integer. (`basic_props:int/1`)

get_code/1: PREDICATE

`get_code(Code)`

Behaves like `current_input(S), get_code(S,Code)`.

Usage 2: ⟨• ISO •⟩

- *Calls should, and exit will be compatible with:*
  `Code` is an integer. (`basic_props:int/1`)
- *The following properties hold upon exit:*
  `Code` is an integer. (`basic_props:int/1`)

get1_code/2: PREDICATE

`get1_code(Stream,Code)`

Reads from `Stream` the next non-layout character (see `code_class/2`) and unifies `Code` with its character code. At end of stream, unifies `Code` with the integer -1.

Usage 2:

- *Calls should, and exit will be compatible with:*
  `Code` is an integer. (`basic_props:int/1`)
- *The following properties should hold at call time:*
  `Stream` is an open stream. (`streams_basic:stream/1`)
- *The following properties hold upon exit:*
  `Code` is an integer. (`basic_props:int/1`)

get1_code/1: PREDICATE

`get1_code(Code)`

Behaves like `current_input(S), get1_code(S,Code)`.

Usage 2:

- *Calls should, and exit will be compatible with:*
  `Code` is an integer. (`basic_props:int/1`)
- *The following properties hold upon exit:*
  `Code` is an integer. (`basic_props:int/1`)

peek_code/2: PREDICATE

`peek_code(Stream,Code)`

Unifies `Code` with the character code of the next character of `Stream`, leaving the stream position unaltered. At end of stream, unifies `Code` with the integer -1.

Usage 2: ⟨• ISO •⟩

- *Calls should, and exit will be compatible with:*
  `Code` is an integer. (`basic_props:int/1`)
- *The following properties should hold at call time:*
  `Stream` is an open stream. (`streams_basic:stream/1`)
- *The following properties hold upon exit:*
  `Code` is an integer. (`basic_props:int/1`)

peek_code/1: PREDICATE

`peek_code(Code)`

Behaves like `current_input(S), peek_code(S,Code)`.

Usage 2: ⟨• ISO •⟩

- *Calls should, and exit will be compatible with:*
  `Code` is an integer. (`basic_props:int/1`)
- *The following properties hold upon exit:*
  `Code` is an integer. (`basic_props:int/1`)

skip_code/2: PREDICATE

`skip_code(Stream,Code)`

Skips just past the next character code `Code` from `Stream`.

Usage 2:

– *The following properties should hold at call time:*

`Stream` is an open stream. (`streams_basic:stream/1`)

`Code` is an integer. (`basic_props:int/1`)

skip_code/1: PREDICATE

`skip_code(Code)`

Behaves like `current_input(S), skip_code(S,Code)`.

Usage 2:

– *The following properties should hold at call time:*

`Code` is an integer. (`basic_props:int/1`)

put_code/2: PREDICATE

`put_code(Stream,Code)`

Outputs to `Stream` the character corresponding to character code `Code`.

Usage 2: ⟨• ISO •⟩

– *The following properties should hold at call time:*

`Stream` is an open stream. (`streams_basic:stream/1`)

`Code` is an integer. (`basic_props:int/1`)

put_code/1: PREDICATE

`put_code(Code)`

Behaves like `current_output(S), put_code(S,Code)`.

Usage 2: ⟨• ISO •⟩

– *The following properties should hold at call time:*

`Code` is an integer. (`basic_props:int/1`)

nl/1: PREDICATE

`nl(Stream)`

Outputs a newline character to `Stream`. Equivalent to `put_code(Stream, 0'\n)`.

Usage 2: ⟨• ISO •⟩

– *The following properties should hold at call time:*

`Stream` is an open stream. (`streams_basic:stream/1`)

nl/0: PREDICATE

`nl`

Behaves like `current_output(S), nl(S)`.

tab/2: PREDICATE

`tab(Stream,Num)`

Outputs `Num` spaces to `Stream`.

Usage 2:

- *The following properties should hold at call time:*
  `Stream` is an open stream. (`streams_basic:stream/1`)
  `Num` is an integer. (`basic_props:int/1`)

tab/1: PREDICATE

`tab(Num)`

Behaves like `current_output(S), tab(S,Num)`.

Usage 2:

- *The following properties should hold at call time:*
  `Num` is an integer. (`basic_props:int/1`)

code_class/2: PREDICATE

`code_class(Code,Class)`

Unifies `Class` with an integer corresponding to the lexical class of the character whose code is `Code`, with the following correspondence:

```
0 - layout (includes space, newline, tab)
1 - small letter
2 - capital letter (including '_')
3 - digit
4 - graphic (includes #$&*+-./:<=>?@^\`~ )
5 - punctuation (includes !;"'%(),[]{|} )
```

Note that in ISO-Prolog the back quote ` is a punctuation character, whereas in Ciao it is a graphic character. Thus, if compatibility with ISO-Prolog is desired, the programmer should not use this character in unquoted names.

Usage 2:

- *Calls should, and exit will be compatible with:*
  `Class` is an integer. (`basic_props:int/1`)
- *The following properties should hold at call time:*
  `Code` is an integer. (`basic_props:int/1`)
- *The following properties hold upon exit:*
  `Class` is an integer. (`basic_props:int/1`)

getct/2: PREDICATE

`getct(Code,Type)`

Reads from the current input stream the next character, unifying `Code` with its character code, and `Type` with its lexical class. At end of stream, unifies both `Code` and `Type` with the integer -1. Equivalent to

```
get(Code), (Code = -1 -> Type = -1 ; code_class(Code,Type))
```

Usage 2:

- *Calls should, and exit will be compatible with:*
  `Code` is an integer. (`basic_props:int/1`)
  `Type` is an integer. (`basic_props:int/1`)
- *The following properties hold upon exit:*
  `Code` is an integer. (`basic_props:int/1`)
  `Type` is an integer. (`basic_props:int/1`)

getct1/2: PREDICATE

`getct1(Code,Type)`

Reads from the current input stream the next non-layout character, unifying `Code` with its character code, and `Type` with its lexical class (which will be nonzero). At end of stream, unifies both `Code` and `Type` with the integer -1. Equivalent to

```
get1(Code), (Code = -1 -> Type = -1 ; code_class(Code,Type))
```

Usage 2:

- *Calls should, and exit will be compatible with:*
  `Code` is an integer. (`basic_props:int/1`)
  `Type` is an integer. (`basic_props:int/1`)
- *The following properties hold upon exit:*
  `Code` is an integer. (`basic_props:int/1`)
  `Type` is an integer. (`basic_props:int/1`)

display/2: PREDICATE

`display(Stream,Term)`

Displays `Term` onto `Stream`. Lists are output using list notation, the other compound terms are output in functional notation. Similar to `write_term(Stream, Term, [ignore_ops(ops)])`, except that curly bracketed notation is not used with `{}/1`, and the `write_strings` flag is not honored.

Usage 2:

- *The following properties should hold at call time:*
  `Stream` is an open stream. (`streams_basic:stream/1`)
  `Term` is any term. (`basic_props:term/1`)
- *The following properties hold upon exit:*
  `Term` is any term. (`basic_props:term/1`)
- *The following properties hold globally:*
  `Term` is not further instantiated. (`basic_props:not_further_inst/2`)

display/1: PREDICATE

`display(Term)`

Behaves like `current_output(S), display(S,Term)`.

Usage 2:

– *The following properties should hold at call time:*
  `Term` is any term. (`basic_props:term/1`)
– *The following properties hold upon exit:*
  `Term` is any term. (`basic_props:term/1`)
– *The following properties hold globally:*
  `Term` is not further instantiated. (`basic_props:not_further_inst/2`)

displayq/2: PREDICATE

`displayq(Stream,Term)`

Similar to `display(Stream, Term)`, but atoms and functors that can't be read back by `read_term/3` are quoted. Thus, similar to `write_term(Stream, Term, [quoted(true), ignore_ops(ops)])`, with the same exceptions as `display/2`.

Usage 2:

– *The following properties should hold at call time:*
  `Stream` is an open stream. (`streams_basic:stream/1`)
  `Term` is any term. (`basic_props:term/1`)
– *The following properties hold upon exit:*
  `Term` is any term. (`basic_props:term/1`)
– *The following properties hold globally:*
  `Term` is not further instantiated. (`basic_props:not_further_inst/2`)

displayq/1: PREDICATE

`displayq(Term)`

Behaves like `current_output(S), displayq(S,Term)`.

Usage 2:

– *The following properties should hold at call time:*
  `Term` is any term. (`basic_props:term/1`)
– *The following properties hold upon exit:*
  `Term` is any term. (`basic_props:term/1`)
– *The following properties hold globally:*
  `Term` is not further instantiated. (`basic_props:not_further_inst/2`)

22 Basic file/stream handling

Author(s): Daniel Cabeza, Mats Carlsson.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

This module provides basic predicates for handling files and streams, in order to make input/output on them.

22.1 Usage and interface (`streams_basic`)

- **Library usage:**
  These predicates are builtin in Ciao, so nothing special has to be done to use them.
- **Exports:**
  - *Predicates:*
    `open/3`, `close/1`, `set_input/1`, `current_input/1`, `set_output/1`, `current_output/1`, `character_count/2`, `line_count/2`, `line_position/2`, `flush_output/1`, `flush_output/0`, `clearerr/1`, `current_stream/3`, `stream_code/2`, `absolute_file_name/2`, `absolute_file_name/7`.
  - *Regular Types:*
    `sourcename/1`, `stream/1`, `io_mode/1`.
  - *Multifiles:*
    `file_search_path/2`, `library_directory/1`.

22.2 Documentation on exports (`streams_basic`)

open/3: PREDICATE

`open(File,Mode,Stream)`

Open `File` with mode `Mode` and return in `Stream` the stream associated with the file. No extension is implicit in `File`.

Usage 1: `open(+sourcename,+io_mode,?(stream))` ⟨• ISO •⟩

- *Description:* Normal use.

Usage 2: `open(+int,+io_mode,?(stream))`

- *Description:* In the special case that `File` is an integer, it is assumed to be a file descriptor passed to Prolog from a foreign function call. The file descriptor is connected to a Prolog stream (invoking the UNIX function `fdopen`) which is unified with `Stream`.

Usage 3: ⟨• ISO •⟩

- *Description:* Normal use.
- *Calls should, and exit will be compatible with:*
  `Stream` is an open stream. (`streams_basic:stream/1`)
- *The following properties should hold at call time:*
  `File` is a source name. (`streams_basic:sourcename/1`)
  `Mode` is an opening mode ('read', 'write' or 'append'). (`streams_basic:io_mode/1`)

– *The following properties hold upon exit:*
  `Stream` is an open stream. (`streams_basic:stream/1`)

Usage 4:

– *Description:* In the special case that `File` is an integer, it is assumed to be a file descriptor passed to Prolog from a foreign function call. The file descriptor is connected to a Prolog stream (invoking the UNIX function `fdopen`) which is unified with `Stream`.
– *Calls should, and exit will be compatible with:*
  `Stream` is an open stream. (`streams_basic:stream/1`)
– *The following properties should hold at call time:*
  `File` is an integer. (`basic_props:int/1`)
  `Mode` is an opening mode ('read', 'write' or 'append'). (`streams_basic:io_mode/1`)
– *The following properties hold upon exit:*
  `Stream` is an open stream. (`streams_basic:stream/1`)

close/1: PREDICATE

`close(Stream)`

Close the stream `Stream`.

Usage 2: ⟨• ISO •⟩

– *The following properties should hold at call time:*
  `Stream` is an open stream. (`streams_basic:stream/1`)

set_input/1: PREDICATE

`set_input(Stream)`

Set the current input stream to `Stream`. A notion of *current input stream* is maintained by the system, so that input predicates with no explicit stream operate on the current input stream. Initially it is set to `user_input`.

Usage 2: ⟨• ISO •⟩

– *The following properties should hold at call time:*
  `Stream` is an open stream. (`streams_basic:stream/1`)

current_input/1: PREDICATE

`current_input(Stream)`

Unify `Stream` with the current input stream.

Usage 2: ⟨• ISO •⟩

– *Calls should, and exit will be compatible with:*
  `Stream` is an open stream. (`streams_basic:stream/1`)
– *The following properties hold upon exit:*
  `Stream` is an open stream. (`streams_basic:stream/1`)

set_output/1: PREDICATE

`set_output(Stream)`

Set the current output stream to `Stream`. A notion of *current output stream* is maintained by the system, so that output predicates with no explicit stream operate on the current output stream. Initially it is set to `user_output`.

Usage 2: ⟨• ISO •⟩

– *The following properties should hold at call time:*
  `Stream` is an open stream. (`streams_basic:stream/1`)

current_output/1: PREDICATE

`current_output(Stream)`

Unify `Stream` with the current output stream.

Usage 2: ⟨• ISO •⟩

– *Calls should, and exit will be compatible with:*
  `Stream` is an open stream. (`streams_basic:stream/1`)
– *The following properties hold upon exit:*
  `Stream` is an open stream. (`streams_basic:stream/1`)

character_count/2: PREDICATE

`character_count(Stream,Count)`

`Count` characters have been read from or written to `Stream`.

Usage 2:

– *Calls should, and exit will be compatible with:*
  `Count` is an integer. (`basic_props:int/1`)
– *The following properties should hold at call time:*
  `Stream` is an open stream. (`streams_basic:stream/1`)
– *The following properties hold upon exit:*
  `Count` is an integer. (`basic_props:int/1`)

line_count/2: PREDICATE

`line_count(Stream,Count)`

`Count` lines have been read from or written to `Stream`.

Usage 2:

– *Calls should, and exit will be compatible with:*
  `Count` is an integer. (`basic_props:int/1`)
– *The following properties should hold at call time:*
  `Stream` is an open stream. (`streams_basic:stream/1`)
– *The following properties hold upon exit:*
  `Count` is an integer. (`basic_props:int/1`)

line_position/2: PREDICATE

`line_position(Stream,Count)`

`Count` characters have been read from or written to the current line of `Stream`.

Usage 2:

- *Calls should, and exit will be compatible with:*
  `Count` is an integer. (`basic_props:int/1`)
- *The following properties should hold at call time:*
  `Stream` is an open stream. (`streams_basic:stream/1`)
- *The following properties hold upon exit:*
  `Count` is an integer. (`basic_props:int/1`)

flush_output/1: PREDICATE

`flush_output(Stream)`

Flush any buffered data to output stream `Stream`.

Usage 2: ⟨• ISO •⟩

- *The following properties should hold at call time:*
  `Stream` is an open stream. (`streams_basic:stream/1`)

flush_output/0: PREDICATE

`flush_output`

Behaves like `current_output(S), flush_output(S)`

clearerr/1: PREDICATE

`clearerr(Stream)`

Clear the end-of-file and error indicators for input stream `Stream`.

Usage 2:

- *The following properties should hold at call time:*
  `Stream` is an open stream. (`streams_basic:stream/1`)

current_stream/3: PREDICATE

`current_stream(Filename,Mode,Stream)`

`Stream` is a stream which was opened in mode `Mode` and which is connected to the absolute file name `Filename` (an atom) or to the file descriptor `Filename` (an integer). This predicate can be used for enumerating all currently open streams through backtracking.

Usage 3:

- *Calls should, and exit will be compatible with:*
  `Filename` is an atom. (`basic_props:atm/1`)
  `Mode` is an opening mode ('read', 'write' or 'append'). (`streams_basic:io_mode/1`)
  `Stream` is an open stream. (`streams_basic:stream/1`)

– *The following properties hold upon exit:*
  `Filename` is an atom. (`basic_props:atm/1`)
  `Mode` is an opening mode ('read', 'write' or 'append'). (`streams_basic:io_mode/1`)
  `Stream` is an open stream. (`streams_basic:stream/1`)

Usage 4:

– *Calls should, and exit will be compatible with:*
  `Filename` is an integer. (`basic_props:int/1`)
  `Mode` is an opening mode ('read', 'write' or 'append'). (`streams_basic:io_mode/1`)
  `Stream` is an open stream. (`streams_basic:stream/1`)
– *The following properties hold upon exit:*
  `Filename` is an integer. (`basic_props:int/1`)
  `Mode` is an opening mode ('read', 'write' or 'append'). (`streams_basic:io_mode/1`)
  `Stream` is an open stream. (`streams_basic:stream/1`)

stream_code/2: PREDICATE

`stream_code(Stream,StreamCode)`

`StreamCode` is the file descriptor (an integer) corresponding to the Prolog stream `Stream`.

Usage 3:

– *Calls should, and exit will be compatible with:*
  `StreamCode` is an integer. (`basic_props:int/1`)
– *The following properties should hold at call time:*
  `Stream` is an open stream. (`streams_basic:stream/1`)
– *The following properties hold upon exit:*
  `StreamCode` is an integer. (`basic_props:int/1`)

Usage 4:

– *The following properties should hold at call time:*
  `Stream` is a free variable. (`term_typing:var/1`)
  `StreamCode` is an integer. (`basic_props:int/1`)
– *The following properties hold upon exit:*
  `Stream` is an open stream. (`streams_basic:stream/1`)

absolute_file_name/2: PREDICATE

`absolute_file_name(RelFileSpec,AbsFileSpec)`

If `RelFileSpec` is an absolute pathname then do an absolute lookup. If `RelFileSpec` is a relative pathname then prefix the name with the name of the current directory and do an absolute lookup. If `RelFileSpec` is a path alias, perform the lookup following the path alias rules (see `sourcename/1`). In all cases: if a matching file with suffix `.pl` exists, then `AbsFileSpec` will be unified with this file. Failure to open a file normally causes an exception. The behaviour can be controlled by the `fileerrors` prolog flag.

Usage 1: `absolute_file_name(+RelFileSpec,-(AbsFileSpec))`

– *Description:* `AbsFileSpec` is the absolute name (with full path) of `RelFileSpec`.

- *Calls should, and exit will be compatible with:*
  +RelFileSpec is a source name. (streams_basic:sourcename/1)
  -(AbsFileSpec) is an atom. (basic_props:atm/1)

Usage 2: absolute_file_name(RelFileSpec,AbsFileSpec)

- *Description:* AbsFileSpec is the absolute name (with full path) of RelFileSpec.
- *Calls should, and exit will be compatible with:*
  RelFileSpec is a source name. (streams_basic:sourcename/1)
  AbsFileSpec is an atom. (basic_props:atm/1)
- *The following properties should hold at call time:*
  RelFileSpec is currently a term which is not a free variable. (term_typing:nonvar/1)
  AbsFileSpec is a free variable. (term_typing:var/1)

absolute_file_name/7: PREDICATE

absolute_file_name(Spec,Opt,Suffix,CurrDir,AbsFile,AbsBase,AbsDir)

AbsFile is the absolute name (with full path) of Spec, which has an optional first suffix Opt and an optional second suffix Suffix, when the current directory is CurrDir. AbsBase is the same as AbsFile, but without the second suffix, and AbsDir is the absolute path of the directory where AbsFile is. The Ciao compiler invokes this predicate with Opt='_opt' and Suffix='.pl' when searching source files.

Usage 2:

- *The following properties should hold at call time:*
  Spec is a source name. (streams_basic:sourcename/1)
  Opt is an atom. (basic_props:atm/1)
  Suffix is an atom. (basic_props:atm/1)
  CurrDir is an atom. (basic_props:atm/1)
  AbsFile is a free variable. (term_typing:var/1)
  AbsBase is a free variable. (term_typing:var/1)
  AbsDir is a free variable. (term_typing:var/1)
- *The following properties hold upon exit:*
  AbsFile is an atom. (basic_props:atm/1)
  AbsBase is an atom. (basic_props:atm/1)
  AbsDir is an atom. (basic_props:atm/1)

sourcename/1: REGTYPE

A source name is a flexible way of referring to a concrete file. A source name is either a relative or absolute filename given as:

- an atom, or
- a unary functor (which represents a *path alias*, see below) applied to a *relative* path, the latter being given as an atom.

In all cases certain filename extensions (e.g., .pl) can be implicit. In the first form above, file names can be relative to the current directory. Also, file names beginning with ~ or $ are treated specially. For example,

`'~/ciao/sample.pl'`
: is equivalent to `'/home/staff/herme/ciao/sample.pl'`, if `/home/staff/herme` is the user's home directory. (This is also equivalent to `'$HOME/ciao/sample.pl'` as explained below.)

`'~bardo/prolog/sample.pl'`
: is equivalent to `'/home/bardo/prolog/sample.pl'`, if `/home/bardo` is bardo's home directory.

`'$UTIL/sample.pl'`
: is equivalent to `'/usr/local/src/utilities/sample.pl'`, if `/usr/local/src/utilities` is the value of the environment variable `UTIL`.

The second form allows using path aliases. Such aliases allow refering to files not with absolute file system paths but with paths which are relative to predefined (or user-defined) abstract names. For example, given the path alias `myutils` which has been defined to refer to path `'/home/bardo/utilities'`, if that directory contains the file `stuff.pl` then the term `myutils(stuff)` in a `use_module/1` declaration would refer to the file `'/home/bardo/utilities/stuff.pl'` (the `.pl` extension is implicit in the `use_module/1` declaration). As a special case, if that directory contains a subdirectory named `stuff` which in turn contains the file `stuff.pl`, the same term would refer to the file `'/home/bardo/utilities/stuff/stuff.pl'`. If a path alias is related to several paths, all paths are scanned in sequence until a match is found. For information on predefined path aliases or how to define new path aliases, see `file_search_path/2`.

Usage 1: `sourcename(F)`

- *Description:* `F` is a source name.

Usage 2: `sourcename(F)`

- *Description:* `F` is a source name.

stream/1: REGTYPE

Streams correspond to the file pointers used at the operating system level, and usually represent opened files. There are four special streams which correspond with the operating system standard streams:

`user_input`
: The standard input stream, i.e. the terminal, usually.

`user_output`
: The standard output stream, i.e. the terminal, usually.

`user_error`
: The standard error stream.

`user`
: The standard input or output stream, depending on context.

Usage 1: `stream(S)`

- *Description:* `S` is an open stream.

Usage 2: `stream(S)`

- *Description:* `S` is an open stream.

io_mode/1: REGTYPE

Can have the following values:

`read` Open the file for input.

`write` Open the file for output. The file is created if it does not already exist, the file will otherwise be truncated.

`append` Open the file for output. The file is created if it does not already exist, the file will otherwise be appended to.

Usage 1: `io_mode(M)`

- *Description:* `M` is an opening mode ('read', 'write' or 'append').

Usage 2: `io_mode(M)`

- *Description:* `M` is an opening mode ('read', 'write' or 'append').

22.3 Documentation on multifiles (streams_basic)

file_search_path/2: PREDICATE

`file_search_path(Alias,Path)`

The path alias `Alias` is linked to path `Path`. Both arguments must be atoms. New facts (or clauses) of this predicate can be asserted to define new path aliases. Predefined path aliases in Ciao are:

`library` Initially points to all Ciao library paths. See `library_directory/1`.

`engine` The path of the Ciao engine builtins.

`.` The current path (`'.'`).

The predicate is *multifile*.

The predicate is of type *dynamic*.

library_directory/1: PREDICATE

`library_directory(Path)`

`Path` is a library path (a path represented by the path alias `library`). Predefined library paths in Ciao are `'$CIAOLIB/lib'`, `'$CIAOLIB/library'`, and `'$CIAOLIB/contrib'`, given that `$CIAOLIB` is the path of the root ciao library directory. More library paths can be defined by asserting new facts (or clauses) of this predicate.

The predicate is *multifile*.

The predicate is of type *dynamic*.

23 Exception handling

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

This module includes predicates related to exceptions, which alter the normal flow of Prolog.

23.1 Usage and interface (exceptions)

- **Library usage:**
  These predicates are builtin in Ciao, so nothing special has to be done to use them.
- **Exports:**
  - *Predicates:*
    catch/3, intercept/3, throw/1, halt/0, halt/1.

23.2 Documentation on exports (exceptions)

catch/3: PREDICATE

catch(Goal,Error,Handler)

Executes Goal. If an exception is raised during its execution, Error is unified with the exception, and if the unification succeeds, the entire execution derived from Goal is aborted, and Handler is executed. The execution resumes with the continuation of the catch/3 call. For example, given the code

```
p(X) :- throw(error), display('---').
p(X) :- display(X).
```

the execution of "catch(p(0), E, display(E)), display(.), fail." results in the output "error.".

Meta-predicate with arguments: catch(goal,?,goal).

Usage 2: ⟨• ISO •⟩

- *Calls should, and exit will be compatible with:*
  Error is any term. (basic_props:term/1)
- *The following properties should hold at call time:*
  Goal is a term which represents a goal, i.e., an atom or a structure. (basic_props:callable/1)
  Handler is a term which represents a goal, i.e., an atom or a structure. (basic_props:callable/1)
- *The following properties hold upon exit:*
  Error is any term. (basic_props:term/1)

intercept/3: PREDICATE

intercept(Goal,Error,Handler)

Executes Goal. If an exception is raised during its execution, Error is unified with the exception, and if the unification succeeds, Handler is executed and then the execution resumes after the predicate which produced the exception. Note the difference with builtin `catch/3`, given the same code defined there, the execution of "`intercept(p(0), E, display(E)), display(.), fail.`" results in the output "`error---.0.`".

Meta-predicate with arguments: `intercept(goal,?,goal)`.

Usage 2:

– *Calls should, and exit will be compatible with:*
  `Error` is any term. (`basic_props:term/1`)
– *The following properties should hold at call time:*
  `Goal` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)
  `Handler` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)
– *The following properties hold upon exit:*
  `Error` is any term. (`basic_props:term/1`)

throw/1: PREDICATE

`throw(Ball)`

Raises an error, throwing the exception `Ball`, to be caught by an ancestor `catch/3` or `intercept/3`. The closest matching ancestor is chosen. Exceptions are also thrown by other builtins in case of error.

Usage 1: ⟨• ISO •⟩

– *Calls should, and exit will be compatible with:*
  `Ball` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

Usage 2: ⟨• ISO •⟩

– *Calls should, and exit will be compatible with:*
  `Ball` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

halt/0: PREDICATE

`halt`

Halt the system, exiting to the invoking shell.

halt/1: PREDICATE

`halt(Code)`

Halt the system, exiting to the invoking shell, returning exit code `Code`.

Usage 2: ⟨• ISO •⟩

– *The following properties should hold at call time:*
  `Code` is an integer. (`basic_props:int/1`)

abort/0: PREDICATE

`abort`

Abort the current execution.

24 Fast/concurrent update of facts

Author(s): Daniel Cabeza, Manuel Carro.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Prolog implementations traditionally implement the concept of dynamic predicates: predicates which can be inspected or modified at run-time, adding or deleting individual clauses. The powerfulness of this feature brings also a great disadvantage: as new clause bodies can be arbitrarily added to the program, new unknown predicate calls can arise, thus preventing any global analysis or optimization of the code. But it is the case that most of the time what the programmer wants is simply to store data, with the purpose of sharing it between search branches, predicates, or even execution threads. In Ciao the concept of data predicate serves this purpose: a data predicate is a predicate composed exclusively by facts, which can be inspected, and dynamically added or deleted, at run-time. Using data predicates instead of normal dynamic predicates brings benefits in terms of speed, but above all turns the code automatically analyzable. There is also a special kind of data predicates, called concurrent predicates, which can be used to communicate/synchronize among different execution threads (see Chapter 26 [Low-level concurrency/multithreading primitives], page 137).

24.1 Usage and interface (`data_facts`)

- **Library usage:**
  These predicates are builtin in Ciao, so nothing special has to be done to use them.
- **Exports:**
  - *Predicates:*
    `asserta_fact/1`, `asserta_fact/2`, `assertz_fact/1`, `assertz_fact/2`, `current_fact/1`, `current_fact/2`, `retract_fact/1`, `retractall_fact/1`, `current_fact_nb/1`, `retract_fact_nb/1`, `close_predicate/1`, `open_predicate/1`, `set_fact/1`, `erase/1`.

24.2 Documentation on exports (`data_facts`)

asserta_fact/1: PREDICATE

`asserta_fact(Fact)`

`Fact` is added to the corresponding data predicate. The fact becomes the first clause of the predicate concerned.

Meta-predicate with arguments: `asserta_fact(fact)`.

Usage 2:

- *The following properties should hold at call time:*
  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

asserta_fact/2: PREDICATE

`asserta_fact(Fact,Ref)`

Same as `asserta_fact/1`, instantiating `Ref` to a unique identifier of the asserted fact.

Meta-predicate with arguments: `asserta_fact(fact,?)`.

Usage 2:

- *The following properties should hold at call time:*
  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)
  `Ref` is a free variable. (`term_typing:var/1`)
- *The following properties hold upon exit:*
  `Ref` is a reference of a dynamic or data clause. (`data_facts:reference/1`)

assertz_fact/1: PREDICATE

`assertz_fact(Fact)`

`Fact` is added to the corresponding data predicate. The fact becomes the last clause of the predicate concerned.

Meta-predicate with arguments: `assertz_fact(fact)`.

Usage 2:

- *The following properties should hold at call time:*
  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

assertz_fact/2: PREDICATE

`assertz_fact(Fact,Ref)`

Same as `assertz_fact/1`, instantiating `Ref` to a unique identifier of the asserted fact.

Meta-predicate with arguments: `assertz_fact(fact,?)`.

Usage 2:

- *The following properties should hold at call time:*
  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)
  `Ref` is a free variable. (`term_typing:var/1`)
- *The following properties hold upon exit:*
  `Ref` is a reference of a dynamic or data clause. (`data_facts:reference/1`)

current_fact/1: PREDICATE

`current_fact(Fact)`

Gives on backtracking all the facts defined as data or concurrent which unify with `Fact`. It is faster than calling the predicate explicitly, which do invoke the meta-interpreter. If the `Fact` has been defined as concurrent and has not been closed, `current_fact/1` will wait (instead of failing) for more clauses to appear after the last clause of `Fact` is returned.

Meta-predicate with arguments: `current_fact(fact)`.

Usage 2:

- *The following properties should hold at call time:*
  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

current_fact/2: PREDICATE

`current_fact(Fact,Ref)`

`Fact` is a fact of a data predicate and `Ref` is its reference identifying it uniquely.

Meta-predicate with arguments: `current_fact(fact,?)`.

Usage 1: `current_fact(+callable,-(reference))`

- *Description:* Gives on backtracking all the facts defined as data which unify with `Fact`, instantiating `Ref` to a unique identifier for each fact.

Usage 2: `current_fact(?(callable),+reference)`

- *Description:* Given `Ref`, unifies `Fact` with the fact identified by it.

Usage 3:

- *Description:* Gives on backtracking all the facts defined as data which unify with `Fact`, instantiating `Ref` to a unique identifier for each fact.
- *The following properties should hold at call time:*

  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

  `Ref` is a free variable. (`term_typing:var/1`)
- *The following properties hold upon exit:*

  `Ref` is a reference of a dynamic or data clause. (`data_facts:reference/1`)

Usage 4:

- *Description:* Given `Ref`, unifies `Fact` with the fact identified by it.
- *Calls should, and exit will be compatible with:*

  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)
- *The following properties should hold at call time:*

  `Ref` is a reference of a dynamic or data clause. (`data_facts:reference/1`)
- *The following properties hold upon exit:*

  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

retract_fact/1: PREDICATE

`retract_fact(Fact)`

Unifies `Fact` with the first matching fact of a data predicate, and then erases it. On backtracking successively unifies with and erases new matching facts. If `Fact` is declared as concurrent and is non- closed, `retract_fact/1` will wait for more clauses or for the closing of the predicate after the last matching clause has been removed.

Meta-predicate with arguments: `retract_fact(fact)`.

Usage 2:

- *The following properties should hold at call time:*

  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

retractall_fact/1: PREDICATE

`retractall_fact(Fact)`

Erase all the facts of a data predicate unifying with `Fact`. Even if all facts are removed, the predicate continues to exist.

Meta-predicate with arguments: `retractall_fact(fact)`.

Usage 2:

– *The following properties should hold at call time:*

`Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

current_fact_nb/1: PREDICATE

`current_fact_nb(Fact)`

Behaves as `current_fact/1` but a fact is never waited on even if it is concurrent and non-closed.

Meta-predicate with arguments: `current_fact_nb(fact)`.

Usage 2:

– *The following properties should hold at call time:*

`Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

retract_fact_nb/1: PREDICATE

`retract_fact_nb(Fact)`

Behaves as `retract_fact/1`, but never waits on a fact, even if it has been declared as concurrent and is non- closed.

Meta-predicate with arguments: `retract_fact_nb(fact)`.

Usage 2:

– *The following properties should hold at call time:*

`Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

close_predicate/1: PREDICATE

`close_predicate(Pred)`

Changes the behavior of the predicate `Pred` if it has been declared as a concurrent predicate: calls to this predicate will fail (instead of wait) if no more clauses of `Pred` are available.

Meta-predicate with arguments: `close_predicate(fact)`.

Usage 2:

– *The following properties should hold at call time:*

`Pred` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

open_predicate/1: PREDICATE

`open_predicate(Pred)`

Reverts the behavior of concurrent predicate `Pred` to waiting instead of failing if no more clauses of `Pred` are available.

Meta-predicate with arguments: `open_predicate(fact)`.

Usage 2:

– *The following properties should hold at call time:*
  `Pred` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

set_fact/1: PREDICATE

`set_fact(Fact)`

Sets `Fact` as the unique fact of the corresponding data predicate.

Meta-predicate with arguments: `set_fact(fact)`.

Usage 2:

– *The following properties should hold at call time:*
  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

erase/1: PREDICATE

`erase(Ref)`

Deletes the clause referenced by `Ref`.

Usage 2:

– *The following properties should hold at call time:*
  `Ref` is a reference of a dynamic or data clause. (`data_facts:reference/1`)

24.3 Documentation on internals (`data_facts`)

fact/1: REGTYPE

Usage: `fact(F)`

– *Description:* `F` is a fact (an atom or a structure).

reference/1: REGTYPE

Usage: `reference(R)`

– *Description:* `R` is a reference of a dynamic or data clause.

25 Changing system behaviour and various flags

Author(s): Daniel Cabeza, Mats Carlsson.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Flags define some parameters of the system and control the behavior of system or library predicates. Each flag has a name and an associated predefined value, and except some system flags which are fixed in general their associated value is changeable. Predefined flags in the system are:

`fileerrors`
: If `on`, predicates handling files give errors when a file is inexistent or an operation is not allowed. If `off`, fail in that conditions. Initially `on`.

`unknown`
: Controls action on calls to undefined predicates. The possible states of the flag are:

  `error` An error is thrown with the error term `existence_error(procedure, F/A)`.

  `fail` The call simply fails.

  `warning` A warning is written and the call fails.

  The state is initially `error`. ⟨ • ISO • ⟩

`quiet`
: Controls which messages issued using `io_aux` are actually written. As the system uses that library to report its messages, this flag controls the *verbosity* of the system. Possible states of the flag are:

  `on` No messages are reported.

  `error` Only error messages are reported.

  `warning` Only error and warning messages are reported.

  `off` All messages are reported, except debug messages. This is the default state.

  `debug` All messages, including debug messages, are reported. This is only intended for the system implementators.

`argv`
: Value is a list of atoms representing the program arguments supplied when the current executable was invoked. This is the value to which is instantiated the argument of the `main/1` predicate at executable startup. Unchangeable.

`bounded`
: It is `false`, to denote that the range of integers can be considered infinite (but see `int/1`). Unchangeable. ⟨ • ISO • ⟩

`integer_rounding_function`
: It is `toward_zero`, so that `-1 =:= -3//2` succeeds. Unchangeable. ⟨ • ISO • ⟩

`max_arity`
: It is 255, so that no compound term (or predicate) can have more than this number of arguments. Unchangeable. ⟨ • ISO • ⟩

`gc`
: Controls whether garbage collection is done. May be `on` (default) or `off`.

`gc_margin`
: An integer `Margin`. If less than `Margin` kilobytes are reclaimed in a garbage collection then the size of the garbage collected area should be increased. Also, no garbage collection is attempted unless the garbage collected area has at least `Margin` kilobytes. Initially 500.

`gc_trace`
: Governs garbage collection trace messages. An element off `[on,off,terse,verbose]`. Initially `off`.

25.1 Usage and interface (prolog_flags)

- **Library usage:**
  These predicates are builtin in Ciao, so nothing special has to be done to use them.
- **Exports:**
  - *Predicates:*
    set_prolog_flag/2, current_prolog_flag/2, prolog_flag/3, push_prolog_flag/2, pop_prolog_flag/1, prompt/2, gc/0, nogc/0, fileerrors/0, nofileerrors/0.
  - *Multifiles:*
    define_flag/3.

25.2 Documentation on exports (prolog_flags)

set_prolog_flag/2: PREDICATE

set_prolog_flag(FlagName,Value)

Set existing flag FlagName to Value.

Usage 2: ⟨• ISO •⟩

- *The following properties should hold at call time:*
  FlagName is an atom. (basic_props:atm/1)
  Value is any term. (basic_props:term/1)

current_prolog_flag/2: PREDICATE

current_prolog_flag(FlagName,Value)

FlagName is an existing flag and Value is the value currently associated with it.

Usage 2: ⟨• ISO •⟩

- *Calls should, and exit will be compatible with:*
  FlagName is an atom. (basic_props:atm/1)
  Value is any term. (basic_props:term/1)
- *The following properties hold upon exit:*
  FlagName is an atom. (basic_props:atm/1)
  Value is any term. (basic_props:term/1)

prolog_flag/3: PREDICATE

prolog_flag(FlagName,OldValue,NewValue)

FlagName is an existing flag, unify OldValue with the value associated with it, and set it to new value NewValue.

Usage 2: prolog_flag(?(FlagName),-(OldValue),-(NewValue))

- *Description:* Same as current_prolog_flag(FlagName, OldValue)

– *The following properties should hold at call time:*
  FlagName is an atom. (basic_props:atm/1)
  The terms OldValue and NewValue are strictly identical. (term_compare:== /2)

Usage 3:

– *Calls should, and exit will be compatible with:*
  FlagName is an atom. (basic_props:atm/1)
  OldValue is any term. (basic_props:term/1)
– *The following properties should hold at call time:*
  NewValue is any term. (basic_props:term/1)
– *The following properties hold upon exit:*
  FlagName is an atom. (basic_props:atm/1)
  OldValue is any term. (basic_props:term/1)

Usage 4: prolog_flag(FlagName,OldValue,NewValue)

– *Description:* Same as current_prolog_flag(FlagName, OldValue)
– *The following properties should hold at call time:*
  OldValue is a free variable. (term_typing:var/1)
  NewValue is a free variable. (term_typing:var/1)
  FlagName is an atom. (basic_props:atm/1)
  The terms OldValue and NewValue are strictly identical. (term_compare:== /2)

push_prolog_flag/2: PREDICATE

push_prolog_flag(Flag,NewValue)

Same as set_prolog_flag/2, but storing current value of Flag to restore it with pop_prolog_flag/1.

Usage 2:

– *The following properties should hold at call time:*
  Flag is an atom. (basic_props:atm/1)
  NewValue is any term. (basic_props:term/1)

pop_prolog_flag/1: PREDICATE

pop_prolog_flag(Flag)

Restore the value of Flag previous to the last non-canceled push_prolog_flag/2 on it.

Usage 2:

– *The following properties should hold at call time:*
  Flag is an atom. (basic_props:atm/1)

prompt/2: PREDICATE

prompt(Old,New)

Unify Old with the current prompt for reading, change it to New.

Usage 2: prompt(Old,New)

– *Description:* Unify Old with the current prompt for reading without changing it.

- *The following properties should hold at call time:*
  `Old` is a free variable. (term_typing:var/1)
  `New` is a free variable. (term_typing:var/1)
  The terms `Old` and `New` are strictly identical. (term_compare:== /2)
- *The following properties hold upon exit:*
  `Old` is an atom. (basic_props:atm/1)
  `New` is an atom. (basic_props:atm/1)

Usage 3:
- *Calls should, and exit will be compatible with:*
  `Old` is an atom. (basic_props:atm/1)
- *The following properties should hold at call time:*
  `New` is an atom. (basic_props:atm/1)
- *The following properties hold upon exit:*
  `Old` is an atom. (basic_props:atm/1)

Usage 4: `prompt(Old,New)`
- *Description:* Unify `Old` with the current prompt for reading without changing it.
- *The following properties should hold at call time:*
  `Old` is a free variable. (term_typing:var/1)
  `New` is a free variable. (term_typing:var/1)
  The terms `Old` and `New` are strictly identical. (term_compare:== /2)
- *The following properties hold upon exit:*
  `Old` is an atom. (basic_props:atm/1)
  `New` is an atom. (basic_props:atm/1)

gc/0: PREDICATE

Usage 1:
- *Description:* Enable garbage collection. Equivalent to `set_prolog_flag(gc, on)`

Usage 2:
- *Description:* Enable garbage collection. Equivalent to `set_prolog_flag(gc, on)`

nogc/0: PREDICATE

Usage 1:
- *Description:* Disable garbage collection. Equivalent to `set_prolog_flag(gc, off)`

Usage 2:
- *Description:* Disable garbage collection. Equivalent to `set_prolog_flag(gc, off)`

fileerrors/0: PREDICATE

Usage 1:
- *Description:* Enable reporting of file errors. Equivalent to `set_prolog_flag(fileerrors, on)`

Usage 2:
- *Description:* Enable reporting of file errors. Equivalent to `set_prolog_flag(fileerrors, on)`

nofileerrors/0: PREDICATE

Usage 1:

- *Description:* Disable reporting of file errors. Equivalent to `set_prolog_flag(fileerrors, off)`

Usage 2:

- *Description:* Disable reporting of file errors. Equivalent to `set_prolog_flag(fileerrors, off)`

25.3 Documentation on multifiles (`prolog_flags`)

define_flag/3: PREDICATE

`define_flag(Flag,Values,Default)`

New flags can be defined by writing facts of this predicate. `Flag` is the name of the new flag, `Values` defines the posible values for the flag (see below) and `Default` defines the predefined value associated with the flag (which should be compatible with `Values`).

The predicate is *multifile*.

Usage 1:

- *Description:* Posible values for the flag are atoms.

  Example:

```
:- multifile define_flag/3.
define_flag(tmpdir, atom, '/tmp').
```

- *Call and exit should be compatible with:*

  `Flag` is an atom. (`basic_props:atm/1`)

  Unify `Values` and `atom`. (`term_basic:= /2`)

  `Default` is an atom. (`basic_props:atm/1`)

Usage 2:

- *Description:* Posible values for the flag are integers.

  Example:

```
:- multifile define_flag/3.
define_flag(max_connections, integer, 10).
```

- *Call and exit should be compatible with:*

  `Flag` is an atom. (`basic_props:atm/1`)

  Unify `Values` and `integer`. (`term_basic:= /2`)

  `Default` is an integer. (`basic_props:int/1`)

Usage 3: `define_flag(Flag,Values,Default)`

- *Description:* Posible values for the flag are the elements of `Values`.

  Example:

```
:- multifile define_flag/3.
define_flag(debug, [on,debug,trace,off], off).
```

- *Call and exit should be compatible with:*

  `Flag` is an atom. (`basic_props:atm/1`)

  `Values` is a list. (`basic_props:list/1`)

  `Default` is any term. (`basic_props:term/1`)

- *The following properties should hold at call time:*

  `Default` is an element of `Values`. (`basic_props:member/2`)

26 Low-level concurrency/multithreading primitives

Author(s): Manuel Carro.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.3#118 (1999/11/25, 19:28:6 MET)

This module provides basic mechanisms for using concurrency and implementing multi-goal applications. It provides a means for arbitrary goals to be specified to be run in a separate stack set; in that case, they are assigned a goal identifier with which further accesses (e.g., asking for more solutions) to the goal can be made. Additionally, in some architectures, these goals can be assigned an O.S. thread, separate from the one which made the initial call, thus providing concurrency and, in multiprocessors, parallelism capabilities.

As for now, the memory space of the threads (c.f., stack sets) is separate in the sense that goals are copied to the new stack set, and bindings of variables are not seen among stack sets which allows forward and backward execution to proceed independently in each stack set, at the cost of the initial goal copy. However, the program space (including, specially, the concurrent predicates) are shared and seen by all the goals and threads, and should be used as the primary means of communication and synchronization. Higer level libraries can be built using these basic blocks.

Additionally, a small set of lock primitives are provided. Locks are associated with atom names. Whereas the concurrent database facilities are enough to implement locks, semaphores, messages, etc., the predicates implementing atom-based locks are faster than the ones accessing the concurrent database (but they are less powerful).

26.1 Usage and interface (`concurrency`)

- **Library usage:**
  These predicates are builtin in Ciao, so nothing special has to be done to use them.
- **Exports:**
  - *Predicates:*
    `eng_call/4`, `eng_call/3`, `eng_backtrack/2`, `eng_cut/1`, `eng_release/1`, `eng_wait/1`, `eng_kill/1`, `eng_killothers/0`, `eng_self/1`, `eng_status/0`, `lock_atom/1`, `unlock_atom/1`, `atom_lock_state/2`.

26.2 Documentation on exports (`concurrency`)

eng_call/4: PREDICATE

Meta-predicate with arguments: `eng_call(goal,?,?,?)`.

Usage 1: `eng_call(+Goal,+EngineCreation,+ThreadCreation,-(GoalId))`

- *Description:* Calls `Goal` in a new engine (stack set), possibly using a new thread, and returns a `GoalId` to designate this new goal henceforth. `EngineCreation` can be either `wait` or `create`; the distinction is not yet meaningful. `ThreadCreation` can be one of `self`, `wait`, or `create`. In the first case the creating thread is used to execute `Goal`, and thus it has to wait until its first result or failure. The call will fail if `Goal` fails, and succeed otherwise. However, the call will always suceed when a remote thread is started. The space and identifiers reclaimed for the thread must be explicitly deallocated by calling `eng_release/1`.

- *The following properties should hold at call time:*

  +Goal is a term which represents a goal, i.e., an atom or a structure. (basic_props:callable/1)

  +EngineCreation is an atom. (basic_props:atm/1)

  +ThreadCreation is an atom. (basic_props:atm/1)

  -(GoalId) is an integer. (basic_props:int/1)

Usage 2: eng_call(Goal,EngineCreation,ThreadCreation,GoalId)

- *Description:* Calls Goal in a new engine (stack set), possibly using a new thread, and returns a GoalId to designate this new goal henceforth. EngineCreation can be either wait or create; the distinction is not yet meaningful. ThreadCreation can be one of self, wait, or create. In the first case the creating thread is used to execute Goal, and thus it has to wait until its first result or failure. The call will fail if Goal fails, and succeed otherwise. However, the call will always suceed when a remote thread is started. The space and identifiers reclaimed for the thread must be explicitly deallocated by calling eng_release/1.
- *The following properties should hold at call time:*

  Goal is currently a term which is not a free variable. (term_typing:nonvar/1)

  EngineCreation is currently a term which is not a free variable. (term_typing:nonvar/1)

  ThreadCreation is currently a term which is not a free variable. (term_typing:nonvar/1)

  GoalId is a free variable. (term_typing:var/1)

  Goal is a term which represents a goal, i.e., an atom or a structure. (basic_props:callable/1)

  EngineCreation is an atom. (basic_props:atm/1)

  ThreadCreation is an atom. (basic_props:atm/1)

  GoalId is an integer. (basic_props:int/1)

eng_call/3: PREDICATE

Meta-predicate with arguments: eng_call(goal,?,?).

Usage 1: eng_call(+Goal,+EngineCreation,+ThreadCreation)

- *Description:* Similar to eng_call/4, but the thread (if created) and stack areas are automatically released upon success or failure of the goal. No GoalId is provided for further interaction with the goal.
- *The following properties should hold at call time:*

  +Goal is a term which represents a goal, i.e., an atom or a structure. (basic_props:callable/1)

  +EngineCreation is an atom. (basic_props:atm/1)

  +ThreadCreation is an atom. (basic_props:atm/1)

Usage 2: eng_call(Goal,EngineCreation,ThreadCreation)

- *Description:* Similar to eng_call/4, but the thread (if created) and stack areas are automatically released upon success or failure of the goal. No GoalId is provided for further interaction with the goal.
- *The following properties should hold at call time:*

  Goal is currently a term which is not a free variable. (term_typing:nonvar/1)

EngineCreation is currently a term which is not a free variable. (term_typing:nonvar/1)

ThreadCreation is currently a term which is not a free variable. (term_typing:nonvar/1)

Goal is a term which represents a goal, i.e., an atom or a structure. (basic_props:callable/1)

EngineCreation is an atom. (basic_props:atm/1)

ThreadCreation is an atom. (basic_props:atm/1)

eng_backtrack/2: PREDICATE

Usage 1: `eng_backtrack(+GoalId,+ThreadCreation)`

- *Description:* Performs backtracking on the goal designed by `GoalId`. A new thread can be used to perform backtracking, according to `ThreadCreation` (same as in `eng_call/4`). Fails if the goal is backtracked over by the local thread, and there are no more solutions. Always succeeds if executed by a remote thread. The engine is **not** automatically released up upon failure: `eng_release/1` must be called to that end.
- *The following properties should hold at call time:*

  `+GoalId` is an integer. (basic_props:int/1)

  `+ThreadCreation` is an atom. (basic_props:atm/1)

Usage 2: `eng_backtrack(GoalId,ThreadCreation)`

- *Description:* Performs backtracking on the goal designed by `GoalId`. A new thread can be used to perform backtracking, according to `ThreadCreation` (same as in `eng_call/4`). Fails if the goal is backtracked over by the local thread, and there are no more solutions. Always succeeds if executed by a remote thread. The engine is **not** automatically released up upon failure: `eng_release/1` must be called to that end.
- *The following properties should hold at call time:*

  `GoalId` is currently a term which is not a free variable. (term_typing:nonvar/1)

  `ThreadCreation` is currently a term which is not a free variable. (term_typing:nonvar/1)

  `GoalId` is an integer. (basic_props:int/1)

  `ThreadCreation` is an atom. (basic_props:atm/1)

eng_cut/1: PREDICATE

Usage 1: `eng_cut(+GoalId)`

- *Description:* Performs a *cut* in the execution of the goal `GoalId`. The next call to `eng_backtrack/2` will therefore backtrack all the way and fail.
- *The following properties should hold at call time:*

  `+GoalId` is an integer. (basic_props:int/1)

Usage 2: `eng_cut(GoalId)`

- *Description:* Performs a *cut* in the execution of the goal `GoalId`. The next call to `eng_backtrack/2` will therefore backtrack all the way and fail.
- *The following properties should hold at call time:*

  `GoalId` is currently a term which is not a free variable. (term_typing:nonvar/1)

  `GoalId` is an integer. (basic_props:int/1)

eng_release/1: PREDICATE

Usage 1: `eng_release(+GoalId)`

- *Description:* Cleans up and releases the engine executing the goal designed by `GoalId`. The engine must be idle, i.e., currently not exedcuting any goal. `eng_wait/1` can be used to ensure this.
- *The following properties should hold at call time:*

  `+GoalId` is an integer. (`basic_props:int/1`)

Usage 2: `eng_release(GoalId)`

- *Description:* Cleans up and releases the engine executing the goal designed by `GoalId`. The engine must be idle, i.e., currently not exedcuting any goal. `eng_wait/1` can be used to ensure this.
- *The following properties should hold at call time:*

  `GoalId` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `GoalId` is an integer. (`basic_props:int/1`)

eng_wait/1: PREDICATE

Usage 1: `eng_wait(+GoalId)`

- *Description:* Waits for the engine executing the goal denoted by `GoalId` to finish the computation (i.e., it has finished searching for a solution, either with success or failure).
- *The following properties should hold at call time:*

  `+GoalId` is an integer. (`basic_props:int/1`)

Usage 2: `eng_wait(GoalId)`

- *Description:* Waits for the engine executing the goal denoted by `GoalId` to finish the computation (i.e., it has finished searching for a solution, either with success or failure).
- *The following properties should hold at call time:*

  `GoalId` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `GoalId` is an integer. (`basic_props:int/1`)

eng_kill/1: PREDICATE

Usage 1: `eng_kill(+GoalId)`

- *Description:* Kills the thread executing `GoalId` (if any), and frees the memory used up by the stack set. Usually one should wait (`eng_wait/1`) for a goal, and then release it, but killing the thread explicitly allows recovering from error states. A goal cannot kill itself. This feature should be used with caution, because there are situations where killing a thread might render the system in an unstable state. Threads should cooperate in their killing, but if the killed thread is blocked in a I/O operation, or inside an internal critical region, this cooperation is not possible and the system, although stopped, might very well end up in a incosistent state.
- *The following properties should hold at call time:*

  `+GoalId` is an integer. (`basic_props:int/1`)

Usage 2: `eng_kill(GoalId)`

- *Description:* Kills the thread executing `GoalId` (if any), and frees the memory used up by the stack set. Usually one should wait (`eng_wait/1`) for a goal, and then release it, but killing the thread explicitly allows recovering from error states. A goal cannot kill itself. This feature should be used with caution, because there are situations where killing a thread might render the system in an unstable state. Threads should cooperate in their killing, but if the killed thread is blocked in a I/O operation, or inside an internal critical region, this cooperation is not possible and the system, although stopped, might very well end up in a incosistent state.
- *The following properties should hold at call time:*

  `GoalId` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `GoalId` is an integer. (`basic_props:int/1`)

eng_killothers/0: PREDICATE

Usage 1:

- *Description:* Kills threads and releases stack sets of all active goals, but the one calling `eng_killothers`. Again, a safety measure. The same cautions as with `eng_kill/1` should be taken.

Usage 2:

- *Description:* Kills threads and releases stack sets of all active goals, but the one calling `eng_killothers`. Again, a safety measure. The same cautions as with `eng_kill/1` should be taken.

eng_self/1: PREDICATE

Usage 1: `eng_self(?(GoalId))`

- *Description:* `GoalId` is unified with the identifier of the goal within which `eng_self/1` is executed.
- *The following properties should hold at call time:*

  `?(GoalId)` is an integer. (`basic_props:int/1`)

Usage 2: `eng_self(GoalId)`

- *Description:* `GoalId` is unified with the identifier of the goal within which `eng_self/1` is executed.
- *The following properties should hold at call time:*

  `GoalId` is an integer. (`basic_props:int/1`)

eng_status/0: PREDICATE

Usage 1:

- *Description:* Prints to standard output the current status of the stack sets.

Usage 2:

- *Description:* Prints to standard output the current status of the stack sets.

eng_status1/1: PREDICATE

No further documentation available for this predicate.

lock_atom/1: PREDICATE

Usage 1: `lock_atom(+Atom)`

- *Description:* The semaphore associated to `Atom` is accessed; if its value is nonzero, it is atomically decremented and the execution of this thread proceeds. Otherwise, the goal waits until a nonzero value is reached. The semaphore is then atomically decremented and the execution of this thread proceeds.
- *The following properties should hold at call time:*

  `+Atom` is an atom. (`basic_props:atm/1`)

Usage 2: `lock_atom(Atom)`

- *Description:* The semaphore associated to `Atom` is accessed; if its value is nonzero, it is atomically decremented and the execution of this thread proceeds. Otherwise, the goal waits until a nonzero value is reached. The semaphore is then atomically decremented and the execution of this thread proceeds.
- *The following properties should hold at call time:*

  `Atom` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `Atom` is an atom. (`basic_props:atm/1`)

unlock_atom/1: PREDICATE

Usage 1: `unlock_atom(+Atom)`

- *Description:* The semaphore associated to `Atom` is atomically incremented.
- *The following properties should hold at call time:*

  `+Atom` is an atom. (`basic_props:atm/1`)

Usage 2: `unlock_atom(Atom)`

- *Description:* The semaphore associated to `Atom` is atomically incremented.
- *The following properties should hold at call time:*

  `Atom` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `Atom` is an atom. (`basic_props:atm/1`)

atom_lock_state/2: PREDICATE

Usage 1: `atom_lock_state(+Atom,+Value)`

- *Description:* Sets the semaphore associated to `Atom` to `Value`. This is usually done at the beginning of the execution, but can be executed at any time. If not called, semaphore associated to atoms are by default inited to 1. It should be used with caution: arbitrary use can transform programs using locks in a mess of internal relations. The change of a semaphore value in a place other than the initialization stage of a program is **not** among the allowed operations as defined by Dijkstra [Dij65,BA82].
- *The following properties should hold at call time:*

  `+Atom` is an atom. (`basic_props:atm/1`)

  `+Value` is an integer. (`basic_props:int/1`)

Usage 2: `atom_lock_state(+Atom,-(Value))`

- *Description:* Consults the `Value` of the semaphore associated to `Atom`. Use sparingly and mainly as a medium to check state correctness. Not among the operations on semaphore by Djikstra.

- *The following properties should hold at call time:*
  `+Atom` is an atom. (`basic_props:atm/1`)
  `-(Value)` is an integer. (`basic_props:int/1`)

Usage 3: `atom_lock_state(Atom,Value)`

- *Description:* Sets the semaphore associated to `Atom` to `Value`. This is usually done at the beginning of the execution, but can be executed at any time. If not called, semaphore associated to atoms are by default inited to 1. It should be used with caution: arbitrary use can transform programs using locks in a mess of internal relations. The change of a semaphore value in a place other than the initialization stage of a program is **not** among the allowed operations as defined by Dijkstra [Dij65,BA82].
- *The following properties should hold at call time:*
  `Atom` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Value` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Atom` is an atom. (`basic_props:atm/1`)
  `Value` is an integer. (`basic_props:int/1`)

Usage 4: `atom_lock_state(Atom,Value)`

- *Description:* Consults the `Value` of the semaphore associated to `Atom`. Use sparingly and mainly as a medium to check state correctness. Not among the operations on semaphore by Djikstra.
- *The following properties should hold at call time:*
  `Atom` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Value` is a free variable. (`term_typing:var/1`)
  `Atom` is an atom. (`basic_props:atm/1`)
  `Value` is an integer. (`basic_props:int/1`)

26.3 Known bugs and planned improvements (`concurrency`)

- **This library is being rewritten at the moment. The Prolog interface herein described may not correspond exactly with the existing implementation. This note will be removed when the documentation corresponds to the implementation.**
- Available architectures implementing POSIX threads and in Windows 32 environments.
- Some implementation of threads have a limit on the total number of threads that can be created by a process. Thread creation, in this case, just hangs. A better solution is planned for the future.

27 Extending the syntax

Author(s): Daniel Cabeza.

This chapter documents the builtin directives in Ciao for extending the syntax of source files. Note that the ISO-Prolog directive `char_conversion/2` is not implemented, since Ciao does not (yet) have a character conversion table.

27.1 Usage and interface (`syntax_extensions`)

- **Library usage:**
  These directives are builtin in Ciao, so nothing special has to be done to use them.

27.2 Documentation on internals (`syntax_extensions`)

op/3: DECLARATION

Usage: :- `op(Priority,Op_spec,Operator).` ⟨ • ISO • ⟩

- *Description:* Updates the operator table for reading the terms in the rest of the current text, in the same way as the builtin `op/3` does. Its scope is local to the current text. Usually included in package files
- *The following properties hold at call time:*

  `Priority` is an integer. (`basic_props:int/1`)

  `Op_spec` specifies the type and associativity of an operator. (`basic_props:operator_specifier/1`)

  `Operator` is an atom or a list of atoms. (`basic_props:atm_or_atm_list/1`)

new_declaration/1: DECLARATION

Usage: :- `new_declaration(Predicate).`

- *Description:* Declares `Predicate` to be a valid declaration in the rest of the current text. Such declarations are simply ignored by the compiler or top level, but can be used by other code processing programs such as an automatic documentator. Also, they can easily translated into standard code (a set of facts and/or rules) by defining a suitable expansion (e.g., by `add_sentence_trans/1`, etc.). This is tipically done in package files.

  Equivalent to `new_declaration(Predicate, off)`.
- *The following properties hold at call time:*

  `Predicate` is a Name/Arity structure denoting a predicate name:

```
predname(P/A) :-
        atm(P),
        int(A).
```

  (`basic_props:predname/1`)

new_declaration/2: DECLARATION

Usage: :- `new_declaration(Predicate,In_Itf)`.

- *Description:* Declares `Predicate` to be a valid declaration in the rest of the current text. Such declarations will be included in the interface file for this file if `In_Itf` is 'on', not if it is 'off'. Including such declarations in interface files makes them visible while processing other modules which make use of this one.
- *The following properties hold at call time:*

  `Predicate` is a Name/Arity structure denoting a predicate name:

  ```
  predname(P/A) :-
          atm(P),
          int(A).
  ```

  (`basic_props:predname/1`)

  `In_Itf` is 'on' or 'off' (`syntax_extensions:switch/1`)

load_compilation_module/1: DECLARATION

Usage: :- `load_compilation_module(Module)`.

- *Description:* Loads code defined in `Module` into the compiler, usually including predicates which define translations of terms, for use with the declarations `add_sentence_trans/1` and similar ones. Normally included in package files.
- *The following properties hold at call time:*

  `Module` is a source name. (`streams_basic:sourcename/1`)

add_sentence_trans/1: DECLARATION

Usage: :- `add_sentence_trans(Predicate)`.

- *Description:* Starts a translation, defined by `Predicate`, of the terms read by the compiler in the rest of the current text. For each subsequent term read by the compiler, the translation predicate is called to obtain a new term which will be used by the compiler as if it where the term present in the file. `Predicate` must be exported by a module previously loaded with a `load_compilation_module/1` declaration. Normally included in package files.
- *The following properties hold at call time:*

  `Predicate` is a translation predicate spec (has arity 2 or 3). (`syntax_extensions:translation_predname/1`)

add_term_trans/1: DECLARATION

Usage: :- `add_term_trans(P)`.

- *Description:* Starts a translation, defined by `Predicate`, of the terms and sub-terms read by the compiler in the rest of the current text. This translation is performed after all translations defined by `add_sentence_trans/1` are done. For each subsequent term read by the compiler, and recursively any subterm included, the translation predicate is called to possibly obtain a new term to replace the old one. Care must be taken of not introducing an endless loop of translations. `Predicate` must be exported by a module previously loaded with a `load_compilation_module/1` declaration. Normally included in package files.
- *The following properties hold at call time:*

  `P` is a translation predicate spec (has arity 2 or 3). (`syntax_extensions:translation_predname/1`)

add_goal_trans/1: DECLARATION

Usage: :- `add_goal_trans(Predicate)`.

- *Description:* Declares a translation, defined by `Predicate`, of the goals present in the clauses of the current text. This translation is performed after all translations defined by `add_sentence_trans/1` and `add_term_trans/1` are done. For each clause read by the compiler, the translation predicate is called with each goal present in the clause to possibly obtain other goal to substitute the original one, and the translation is subsequently applied to the resulting goal. Care must be taken of not introducing an endless loop of translations. `Predicate` must be exported by a module previously loaded with a `load_compilation_module/1` declaration. Bear in mind that this type of translation noticeably slows down compilation. Normally included in package files.
- *The following properties hold at call time:*

  `Predicate` is a translation predicate spec (has arity 2 or 3). (`syntax_extensions:translation_predname/1`)

add_clause_trans/1: DECLARATION

Usage: :- `add_clause_trans(Predicate)`.

- *Description:* Declares a translation, defined by `Predicate`, of the clauses of the current text. The translation is performed before `add_goal_trans/1` translations but after `add_sentence_trans/1` and `add_term_trans/1` translations. The usefulness of this translation is that information of the interface of related modules is available when it is performed. For each clause read by the compiler, the translation predicate is called with the first argument instantiated to a structure `clause(Head,Body)`, and the predicate must return in the second argument a similar structure, without changing the functor in `Head`. Before executing the translation predicate with actual clauses it is called with an input of `clause(0,0)` (output is discarded).
- *The following properties hold at call time:*

  `Predicate` is a translation predicate spec (has arity 2 or 3). (`syntax_extensions:translation_predname/1`)

translation_predname/1: REGTYPE

A translation predicate is a predicate of arity 2 or 3 used to make compile-time translations. The compiler invokes a translation predicate instantiating its first argument with the item to be translated, and if the predicate is of arity 3 its third argument with the name of the module where the translation is done. If the call is successful, the second argument is used as if that item were in the place of the original, else the original item is used.

Usage: `translation_predname(P)`

- *Description:* `P` is a translation predicate spec (has arity 2 or 3).

28 Message printing primitives

Author(s): Daniel Cabeza.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

This module provides predicates for printing in a unified way informational messages, and also for printing some terms in a specific way.

28.1 Usage and interface (`io_aux`)

- **Library usage:**
  These predicates are builtin in Ciao, so nothing special has to be done to use them.
- **Exports:**
  - *Predicates:*
    `message/2`, `message_lns/4`, `display_string/1`.

28.2 Documentation on exports (`io_aux`)

message/2: PREDICATE

`message(Type,Message)`

Output to standard error `Message`, which is of type `Type`. The `quiet` *prolog flag* (see Chapter 25 [Changing system behaviour and various flags], page 131) controls which messages are actually output, depending on its type. Also, for 'error', 'warning' and 'note' messages, a prefix is output which denotes the severity of the message. `Message` is an item or a list of items from this list:

`$$(String)`
: `String` is a string, which is output with `display_string/1`.

`''(Term)`
: `Term` is output quoted. If the module `write` is loaded, the term is output with `writeq/1`, else with `displayq/1`.

`~~(Term)`
: `Term` is output unquoted. If the module `write` is loaded, the term is output with `write/1`, else with `display/1`.

`[](Term)`
: `Term` is output as a message, can be an item or a list of items from this list.

`Term`
: Any other term is output with `display/1`.

Usage 1: `message(Type,Message)`

- *The following properties should hold at call time:*
  `Type` is an atom. (`basic_props:atm/1`)
  `Type` is an element of `[error,warning,note,message,debug]`. (`basic_props:member/2`)

Usage 2: `message(Type,Message)`

- *The following properties should hold at call time:*
  `Type` is an atom. (`basic_props:atm/1`)
  `Type` is an element of `[error,warning,note,message,debug]`. (`basic_props:member/2`)

message_lns/4: PREDICATE

`message_lns(Type,L0,L1,Message)`

Output to standard error `Message`, which is of type `Type`, and occurs between lines `L0` and `L1`. This is the same as `message/2`, but printing the lines where the message occurs in a unified way.

Usage 1: `message_lns(Type,L0,L1,Message)`

- *The following properties should hold at call time:*

  `Type` is an atom. (`basic_props:atm/1`)

  `Type` is an element of `[error,warning,note,message,debug]`. (`basic_props:member/2`)

Usage 2: `message_lns(Type,L0,L1,Message)`

- *The following properties should hold at call time:*

  `Type` is an atom. (`basic_props:atm/1`)

  `Type` is an element of `[error,warning,note,message,debug]`. (`basic_props:member/2`)

error/1: PREDICATE

Defined as

```
error(Message) :-
        message(error,Message).
```

.

warning/1: PREDICATE

Defined as

```
warning(Message) :-
        message(warning,Message).
```

.

note/1: PREDICATE

Defined as

```
note(Message) :-
        message(note,Message).
```

.

message/1: PREDICATE

Defined as

```
message(Message) :-
        message(message,Message).
```

.

debug/1: PREDICATE

Defined as

```
debug(Message) :-
        message(debug,Message).
```

.

inform_user/1: PREDICATE

`inform_user(Message)`

Similar to `message/1`, but `Message` is output with `display_list/1`. This predicate is obsolete, and may disappear in future versions.

display_string/1: PREDICATE

`display_string(String)`

Output `String` as the sequence of characters it represents.

Usage 1: `display_string(String)`

- *The following properties should hold at call time:*
  `String` is a string (a list of character codes). (`basic_props:string/1`)

Usage 2: `display_string(String)`

- *The following properties should hold at call time:*
  `String` is a string (a list of character codes). (`basic_props:string/1`)

display_list/1: PREDICATE

`display_list(List)`

Outputs `List`. If `List` is a list, do `display/1` on each of its elements, else do `display/1` on `List`.

display_term/1: PREDICATE

`display_term(Term)`

Output `Term` in a way that a `read/1` will be able to read it back, even if operators change.

28.3 Known bugs and planned improvements (`io_aux`)

- `message/2` assumes that a module with name 'write' is library(write).

29 Attributed variables

Author(s): Christian Holzbaur, Daniel Cabeza, Manuel Carro.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

This library implements *attributed variables*, which provide a mechanism for extensible unification.

29.1 Usage and interface (`attributes`)

- **Library usage:**
  These predicates are builtin in Ciao, so nothing special has to be done to use them.
- **Exports:**
  - *Predicates:*
    `attach_attribute/2`, `get_attribute/2`, `update_attribute/2`, `detach_attribute/1`.

29.2 Documentation on exports (`attributes`)

attach_attribute/2: PREDICATE

Usage 1: `attach_attribute(Var,Attr)`
- *Description:* Attach attribute `Attr` to `Var`.
- *The following properties should hold at call time:*
  `Var` is a free variable. (`term_typing:var/1`)
  `Attr` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

Usage 2: `attach_attribute(Var,Attr)`
- *Description:* Attach attribute `Attr` to `Var`.
- *The following properties should hold at call time:*
  `Var` is a free variable. (`term_typing:var/1`)
  `Attr` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

get_attribute/2: PREDICATE

Usage 1: `get_attribute(Var,Attr)`
- *Description:* Unify `Attr` with the attribute of `Var`, or fail if `Var` has no attribute.
- *The following properties should hold at call time:*
  `Var` is a free variable. (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  `Attr` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

Usage 2: `get_attribute(Var,Attr)`
- *Description:* Unify `Attr` with the attribute of `Var`, or fail if `Var` has no attribute.
- *The following properties should hold at call time:*
  `Var` is a free variable. (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  `Attr` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

update_attribute/2: PREDICATE

Usage 1: `update_attribute(Var,Attr)`

- *Description:* Change the attribute of attributed variable `Var` to `Attr`.
- *The following properties should hold at call time:*

  `Var` is a free variable. (`term_typing:var/1`)

  `Attr` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

Usage 2: `update_attribute(Var,Attr)`

- *Description:* Change the attribute of attributed variable `Var` to `Attr`.
- *The following properties should hold at call time:*

  `Var` is a free variable. (`term_typing:var/1`)

  `Attr` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

detach_attribute/1: PREDICATE

Usage 1: `detach_attribute(Var)`

- *Description:* Take out the attribute from the attributed variable `Var`.
- *The following properties should hold at call time:*

  `Var` is a free variable. (`term_typing:var/1`)

Usage 2: `detach_attribute(Var)`

- *Description:* Take out the attribute from the attributed variable `Var`.
- *The following properties should hold at call time:*

  `Var` is a free variable. (`term_typing:var/1`)

30 Gathering some basic internal info

Author(s): Daniel Cabeza, Manuel Carro.
Version: 1.5#118 (2000/4/19, 18:13:43 CEST)
Version of last change: 1.3#13 (1999/7/2, 18:49:49 MEST)
.... In development ....

30.1 Usage and interface (`system_info`)

- **Library usage:**
  These predicates are builtin in Ciao, so nothing special has to be done to use them.
- **Exports:**
  - *Predicates:*
    `get_arch/1`, `get_os/1`, `this_module/1`, `current_module/1`, `ciaolibdir/1`.

30.2 Documentation on exports (`system_info`)

get_arch/1: PREDICATE

This predicate will describe the computer architecture wich is currently executing the predicate.

Computer architectures are identified by a simple atom. This atom is implementation-defined, and may suffer any change from one Ciao Prolog version to another.

For example,Ciao Prolog running on an Intel-based machine will retrieve:

```
?- get_arch(I).

I = i86 ? ;

no
?-
```

Usage 1: `get_arch(?(ArchDescriptor))`
- *Description:* Unifies `ArchDescriptor` with a simple atom which describes the computer architecture currently executing the predicate.
- *Calls should, and exit will be compatible with:*
  `?(ArchDescriptor)` is an atom. (`basic_props:atm/1`)

Usage 2: `get_arch(ArchDescriptor)`
- *Description:* Unifies `ArchDescriptor` with a simple atom which describes the computer architecture currently executing the predicate.
- *Calls should, and exit will be compatible with:*
  `ArchDescriptor` is an atom. (`basic_props:atm/1`)

get_os/1: PREDICATE

This predicate will describe the Operating System which is running on the machine currently executing the Prolog program.

Operating Systems are identified by a simple atom. This atom is implementation-defined, and may suffer any change from one Ciao Prolog version to another.

For example,Ciao Prolog running on Linux will retrieve:

```
?- get_os(I).

I = 'LINUX' ? ;

no
?-
```

Usage 1: `get_os(?(OsDescriptor))`

- *Description:* Unifies `OsDescriptor` with a simple atom which describes the running Operating System when predicate was called.
- *Calls should, and exit will be compatible with:*
  `?(OsDescriptor)` is an atom. (`basic_props:atm/1`)

Usage 2: `get_os(OsDescriptor)`

- *Description:* Unifies `OsDescriptor` with a simple atom which describes the running Operating System when predicate was called.
- *Calls should, and exit will be compatible with:*
  `OsDescriptor` is an atom. (`basic_props:atm/1`)

this_module/1: PREDICATE

Meta-predicate with arguments: `this_module(addmodule)`.

Usage 1:

- *Description:* `Module` is the internal module identifier for current module.
- *Call and exit should be compatible with:*
  `Arg1` is an internal module identifier (`system_info:internal_module_id/1`)

Usage 2:

- *Description:* `Module` is the internal module identifier for current module.
- *Call and exit should be compatible with:*
  `Arg1` is an internal module identifier (`system_info:internal_module_id/1`)

current_module/1: PREDICATE

This predicate will successively unify its argument with all module names currently loaded. Module names will be simple atoms.

When called using a free variable as argument, it will retrieve on backtracking all modules currently loaded. This is usefull when called from the Ciao `toplevel`.

When called using a module name as argument it will check whether the given module is loaded or not. This is usefull when called from user programs.

Usage 1: `current_module(Module)`

- *Description:* Retrieves (on backtracking) all currently loaded modules into your application.

– *Call and exit should be compatible with:*
  `Module` is an atom. (`basic_props:atm/1`)

Usage 2: `current_module(Module)`

– *Description:* Retrieves (on backtracking) all currently loaded modules into your application.
– *Call and exit should be compatible with:*
  `Module` is an atom. (`basic_props:atm/1`)

ciaolibdir/1: PREDICATE

Usage 1: `ciaolibdir(CiaoPath)`

– *Description:* `CiaoPath` is the path to the root of the Ciao libraries. Inside this directory, there are the directories 'lib', 'library' and 'contrib', which contain library modules.
– *Call and exit should be compatible with:*
  `CiaoPath` is an atom. (`basic_props:atm/1`)

Usage 2: `ciaolibdir(CiaoPath)`

– *Description:* `CiaoPath` is the path to the root of the Ciao libraries. Inside this directory, there are the directories 'lib', 'library' and 'contrib', which contain library modules.
– *Call and exit should be compatible with:*
  `CiaoPath` is an atom. (`basic_props:atm/1`)

30.3 Documentation on internals (`system_info`)

internal_module_id/1: PROPERTY

For a user file it is a term user/1 with an argument different for each user file, for other modules is just the name of the module (as an atom).

Usage: `internal_module_id(M)`

– *Description:* `M` is an internal module identifier

PART III - ISO-Prolog library (iso)

This part documents the *iso* package which provides to Ciao programs (most of) the ISO-Prolog functionality , including the *ISO-Prolog builtins* not covered by the basic library. All these predicates are loaded by default in user files and in modules which use standard module declarations such as:

`:- module(`*modulename*`,`*exports*`).`

which are equivalent to:

`:- module(`*modulename*`,`*exports*`,[iso]).`

or

`:- module(`*modulename*`,`*exports*`).`

`:- use_package([iso]).`

If you do not want these ISO builtins loaded for a given file (in order to make the executable smaller) you can ask for this explicitly using:

`:- module(`*modulename*`,`*exports*`,[]).`

or

`:- module(`*modulename*`,`*exports*`).`

`:- use_package([]).`

See the description of the declarations for declaring modules and using other modules, and the documentation of the `iso` library for details.

31 ISO-Prolog package

This library package allows the use of the ISO-Prolog predicates in Ciao programs. It is included by default in modules starting with a `module/2` declaration or user files without a starting `use_package/1` declaration.

31.1 Usage and interface (`iso`)

- **Library usage:**
  `:- use_package(iso).`
  or
  `:- module(...,...,[iso]).`
- **Other modules used:**
  - *System library modules:*
    `aggregates`, `dynamic`, `iso_misc`, `iso_byte_char`, `iso_incomplete`, `operators`, `read`, `streams`, `write`.

32 Standard operators

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 0.9#7 (1999/3/17, 17:29:23 MET)

Standard operators are defined by this predicate:

```
standard_ops :-
        op(1200,xfx,[:-]),
        op(1200,fx,[:-,?-]),
        op(1100,xfy,[;]),
        op(1050,xfy,[->]),
        op(1000,xfy,[',']),
        op(950,xfy,[&,\&]),
        op(950,xf,[&]),
        op(900,fy,[\+]),
        op(700,xfx,[=,\=,==,\==,@<,@>,@=<,@>=,=..,is,=:=,=\=,<,=<,>,>=]),
        op(550,xfx,[:]),
        op(500,yfx,[+,-,/\,\/,#]),
        op(500,fy,[++,--]),
        op(400,yfx,[*,/,//,rem,mod,<<,>>]),
        op(200,fy,[+,-,\]),
        op(200,xfx,[**]),
        op(200,xfy,[^]),
        op(200,xfy,[`]),
        op(25,fy,[^]).
```

32.1 Usage and interface (operators)

- **Library usage:**
  `:- use_module(library(operators)).`
- **Exports:**
  - *Predicates:*
    `op/3`, `current_op/3`, `current_prefixop/3`, `current_infixop/4`, `current_postfixop/3`.
- **Other modules used:**
  - *System library modules:*
    `assertions/doc_props`.

32.2 Documentation on exports (operators)

op/3: PREDICATE

Usage: `op(+Precedence,+Type,+Name)`

- *Description:* Declares the atom `Name` to be an operator of the stated `Type` and `Precedence`. `Name` may also be a list of atoms in which case all of them are declared to be operators. If `Precedence` is 0 then the operator properties of `Name` (if any) are cancelled.

– *The following properties hold globally:*

  Documentation is still incomplete: `op(+Precedence,+Type,+Name)` may not conform the functionality documented. (`doc_props:doc_incomplete/1`)

current_op/3: PREDICATE

Usage: `current_op(?(Precedence),?(Type),?(Op))`

– *Description:* The atom `Op` is currently an operator of type `Type` and precedence `Precedence`. Neither `Op` nor the other arguments need be instantiated at the time of the call; i.e., this predicate can be used to generate as well as to test.

– *The following properties hold globally:*

  Documentation is still incomplete: `current_op(?(Precedence),?(Type),?(Op))` may not conform the functionality documented. (`doc_props:doc_incomplete/1`)

current_prefixop/3: PREDICATE

No further documentation available for this predicate.

current_infixop/4: PREDICATE

No further documentation available for this predicate.

current_postfixop/3: PREDICATE

No further documentation available for this predicate.

33 All solutions predicates

Author(s): First version by Richard A. O'Keefe and David H.D. Warren. Changes by Mats Carlsson, Daniel Cabeza, and Manuel Hermenegildo.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.5#115 (2000/4/12, 12:17:22 CEST)

This module implements the standard solution aggregation predicates.

When there are many solutions to a problem, and when all those solutions are required to be collected together, this can be achieved by repeatedly backtracking and gradually building up a list of the solutions. The following built-in predicates are provided to automate this process.

33.1 Usage and interface (aggregates)

- **Library usage:**
  `:- use_module(library(aggregates)).`
- **Exports:**
  - *Predicates:*
    `setof/3`, `bagof/3`, `findall/3`, `findall/4`, `findnsols/4`, `findnsols/5`, `^/2`.
- **Other modules used:**
  - *System library modules:*
    `sort`, `lists`.

33.2 Documentation on exports (aggregates)

setof/3: PREDICATE

Meta-predicate with arguments: `setof(?,goal,?)`.

Usage: `setof(@(Template),+Generator,?(Set))` ⟨• ISO •⟩

- *Description:* Finds the `Set` of instances of the `Template` satisfying `Generator`. The set is in ascending order (see Chapter 18 [Comparing terms], page 97 for a definition of this order) without duplicates, and is non-empty. If there are no solutions, `setof` fails. `setof` may succeed in more than one way, binding free variables in `Generator` to different values. This can be avoided by using existential quantifiers on the free variables in front of `Generator`, using `^/2`. For example, given the clauses:

```
father(bill, tom).
father(bill, ann).
father(bill, john).
father(harry, july).
father(harry, daniel).
```

The following query produces two alternative solutions via backtracking:

```
?- setof(X,father(F,X),Sons).

F = bill,
Sons = [ann,john,tom] ? ;

F = harry,
```

```
Sons = [daniel,july] ? ;

no
?-
```

– *The following properties should hold upon exit:*

`Template` is any term. (`basic_props:term/1`)

`Generator` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

`Set` is a list. (`basic_props:list/1`)

bagof/3: PREDICATE

Meta-predicate with arguments: `bagof(?,goal,?)`.

Usage: `bagof(@(Template),+Generator,?(Bag))` ⟨• ISO •⟩

– *Description:* Finds all the instances of the `Template` produced by the `Generator`, and returns them in the `Bag` in the order in which they were found. If the `Generator` contains free variables which are not bound in the `Template`, it assumes that this is like any other Prolog question and that you want bindings for those variables. This can be avoided by using existential quantifiers on the free variables in front of the `Generator`, using `^/2`.

findall/3: PREDICATE

Meta-predicate with arguments: `findall(?,goal,?)`.

Usage: `findall(?(Template),+Generator,?(List))` ⟨• ISO •⟩

– *Description:* A special case of bagof, where all free variables in the `Generator` are taken to be existentially quantified. Faster than the other aggregation predicates.

findall/4: PREDICATE

Meta-predicate with arguments: `findall(?,goal,?,?)`.

Usage: `findall(?(Template),+Generator,?(List),?(Tail))` ⟨• ISO •⟩

– *Description:* As `findall/3`, but returning in `Tail` the tail of `List`.

findnsols/4: PREDICATE

`findnsols(N,Template,Generator,List)`

As `findall/3`, but generating at most `N` solutions of `Generator`. Thus, the length of `List` will not be greater than `N`. If `N`=<0, returns directly an empty list. This predicate is especially useful if `Generator` may have an infinite number of solutions.

Meta-predicate with arguments: `findnsols(?,?,goal,?)`.

findnsols/5: PREDICATE

`findnsols(N,Template,Generator,List,Tail)`

As `findnsols/4`, but returning in `Tail` the tail of `List`.

Meta-predicate with arguments: `findnsols(?,?,goal,?,?)`.

^/2: PREDICATE

Usage: `X ^ P`

- *Description:* Existential quantification: `X` is existentially quantified in `P`. E.g., in `A^p(A,B)`, `A` is existentially quantified. Used only within aggregation predicates. In all other contexts, simply, execute the procedure call `P`.

34 Dynamic predicates

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 0.5#30 (1998/6/30, 14:23:37 MET DST)

This module implements the assert/retract family of predicates to manipulate dynamic predicates.

The predicates defined in this module allow modification of the program as it is actually running. Clauses can be added to the program (*asserted*) or removed from the program (*retracted*). For these predicates, the argument which corresponds to the clause head must be instantiated to an atom or a compound term. The argument corresponding to the clause must be instantiated either to a term `Head :- Body` or, if the body part is empty, to `Head`. An empty body part is represented as `true`.

34.1 Usage and interface (dynamic)

- **Library usage:**
  `:- use_module(library(dynamic)).`
- **Exports:**
  - *Predicates:*
    `asserta/1`, `asserta/2`, `assertz/1`, `assertz/2`, `assert/1`, `assert/2`, `retract/1`, `retractall/1`, `abolish/1`, `clause/2`, `clause/3`, `current_predicate/1`, `data/1`, `wellformed_body/3`.
  - *Multifiles:*
    `do_on_abolish/1`.
- **Other modules used:**
  - *System library modules:*
    `assertions/doc_props`.

34.2 Documentation on exports (dynamic)

asserta/1: PREDICATE

Meta-predicate with arguments: `asserta(clause)`.

Usage: `asserta(+Clause)`

- *Description:* The current instance of `Clause` is interpreted as a clause and is added to the current program. The predicate concerned must be dynamic. The new clause becomes the *first* clause for the predicate concerned. Any uninstantiated variables in the `Clause` will be replaced by new private variables.
- *The following properties should hold globally:*
  Documentation is still incomplete: `asserta(+Clause)` may not conform the functionality documented. (`doc_props:doc_incomplete/1`)

asserta/2: PREDICATE

Meta-predicate with arguments: `asserta(clause,?)`.

Usage: `asserta(+Clause,-(Ref))`

- *Description:* Like `asserta/1`. `Ref` is a unique identifier of the asserted clause.
- *The following properties should hold globally:*

  Documentation is still incomplete: `asserta(+Clause,-(Ref))` may not conform the functionality documented. (`doc_props:doc_incomplete/1`)

assertz/1: PREDICATE

Meta-predicate with arguments: `assertz(clause)`.

Usage: `assertz(+Clause)`

- *Description:* Like `asserta/1`, except that the new clause becomes the *last* clause for the predicate concerned.
- *The following properties should hold globally:*

  Documentation is still incomplete: `assertz(+Clause)` may not conform the functionality documented. (`doc_props:doc_incomplete/1`)

assertz/2: PREDICATE

Meta-predicate with arguments: `assertz(clause,?)`.

Usage: `assertz(+Clause,-(Ref))`

- *Description:* Like `assertz/1`. `Ref` is a unique identifier of the asserted clause.
- *The following properties should hold globally:*

  Documentation is still incomplete: `assertz(+Clause,-(Ref))` may not conform the functionality documented. (`doc_props:doc_incomplete/1`)

assert/1: PREDICATE

Meta-predicate with arguments: `assert(clause)`.

Usage: `assert(+Clause)`

- *Description:* Identical to `assertz/1`. Included for compatibility.
- *The following properties should hold globally:*

  Documentation is still incomplete: `assert(+Clause)` may not conform the functionality documented. (`doc_props:doc_incomplete/1`)

assert/2: PREDICATE

Meta-predicate with arguments: `assert(clause,?)`.

Usage: `assert(+Clause,-(Ref))`

- *Description:* Identical to `assertz/2`. Included for compatibility.
- *The following properties should hold globally:*

  Documentation is still incomplete: `assert(+Clause,-(Ref))` may not conform the functionality documented. (`doc_props:doc_incomplete/1`)

retract/1: PREDICATE

Meta-predicate with arguments: `retract(clause)`.

Usage: `retract(+Clause)`

- *Description:* The first clause in the program that matches `Clause` is erased. The predicate concerned must be dynamic.

  The predicate `retract/1` may be used in a non-determinate fashion, i.e., it will successively retract clauses matching the argument through backtracking. If reactivated by backtracking, invocations of the predicate whose clauses are being retracted will proceed unaffected by the retracts. This is also true for invocations of `clause` for the same predicate. The space occupied by a retracted clause will be recovered when instances of the clause are no longer in use.
- *The following properties should hold globally:*

  Documentation is still incomplete: `retract(+Clause)` may not conform the functionality documented. (`doc_props:doc_incomplete/1`)

retractall/1: PREDICATE

Meta-predicate with arguments: `retractall(fact)`.

Usage: `retractall(+Head)`

- *Description:* Erase all clauses whose head matches `Head`, where `Head` must be instantiated to an atom or a compound term. The predicate concerned must be dynamic. The predicate definition is retained.
- *The following properties should hold globally:*

  Documentation is still incomplete: `retractall(+Head)` may not conform the functionality documented. (`doc_props:doc_incomplete/1`)

abolish/1: PREDICATE

Meta-predicate with arguments: `abolish(spec)`.

Usage: `abolish(+Spec)`

- *Description:* Erase all clauses of the predicate specified by the predicate spec, or list of predicate specs, `Spec`. The predicate definition itself is also erased (the predicate is deemed undefined after execution of the abolish). The predicates concerned must all be user defined.
- *The following properties should hold globally:*

  Documentation is still incomplete: `abolish(+Spec)` may not conform the functionality documented. (`doc_props:doc_incomplete/1`)

clause/2: PREDICATE

Meta-predicate with arguments: `clause(fact,?)`.

Usage: `clause(+Head,?(Body))`

- *Description:* The clause '`Head :- Body`' exists in the current program. The predicate concerned must be dynamic.
- *The following properties should hold globally:*

  Documentation is still incomplete: `clause(+Head,?(Body))` may not conform the functionality documented. (`doc_props:doc_incomplete/1`)

clause/3: PREDICATE

`clause(Head,Body,Ref)`

Like `clause(Head,Body)`, plus the clause is uniquely identified by `Ref`.

Meta-predicate with arguments: `clause(fact,?,?)`.

Usage 1: `clause(+Head,?(Body),?(Ref))`

- *Description:* `Head` must be instantiated to an atom or a compound term.
- *The following properties should hold globally:*
  Documentation is still incomplete: `clause(+Head,?(Body),?(Ref))` may not conform the functionality documented. (`doc_props:doc_incomplete/1`)

Usage 2: `clause(?(Head),?(Body),+Ref)`

- *Description:* `Ref` must be instantiated to a valid identifier.
- *The following properties should hold globally:*
  Documentation is still incomplete: `clause(?(Head),?(Body),+Ref)` may not conform the functionality documented. (`doc_props:doc_incomplete/1`)

current_predicate/1: PREDICATE

Usage: `current_predicate(?(Spec))`

- *Description:* A user defined predicate is named `Spec`.
- *The following properties should hold globally:*
  Documentation is still incomplete: `current_predicate(?(Spec))` may not conform the functionality documented. (`doc_props:doc_incomplete/1`)

dynamic/1: PREDICATE

No further documentation available for this predicate.

data/1: PREDICATE

No further documentation available for this predicate.

wellformed_body/3: PREDICATE

`wellformed_body(BodyIn,Env,BodyOut)`

`BodyIn` is a well-formed clause body. `BodyOut` is its counterpart with no single-variable meta-goals (i.e., with `call(X)` for `X`). `Env` denotes if global cuts are admissible in `BodyIn` (`+` if they are, `-` if they are not).

34.3 Documentation on multifiles (`dynamic`)

do_on_abolish/1: PREDICATE

`do_on_abolish(Head)`

A hook predicate which will be called when the definition of the predicate of `Head` is abolished.

The predicate is *multifile*.

35 iso_byte_char (library)

35.1 Usage and interface (iso_byte_char)

- **Library usage:**
  `:- use_module(library(iso_byte_char)).`
- **Exports:**
  - *Predicates:*
    `char_code/2`, `atom_chars/2`, `number_chars/2`, `get_byte/1`, `get_byte/2`, `peek_byte/1`, `peek_byte/2`, `put_byte/1`, `put_byte/2`, `get_char/1`, `get_char/2`, `peek_char/1`, `peek_char/2`, `put_char/1`, `put_char/2`.

35.2 Documentation on exports (iso_byte_char)

char_code/2: PREDICATE
No further documentation available for this predicate.

atom_chars/2: PREDICATE
No further documentation available for this predicate.

number_chars/2: PREDICATE
No further documentation available for this predicate.

get_byte/1: PREDICATE
No further documentation available for this predicate.

get_byte/2: PREDICATE
No further documentation available for this predicate.

peek_byte/1: PREDICATE
No further documentation available for this predicate.

peek_byte/2: PREDICATE
No further documentation available for this predicate.

put_byte/1: PREDICATE

No further documentation available for this predicate.

put_byte/2: PREDICATE

No further documentation available for this predicate.

get_char/1: PREDICATE

No further documentation available for this predicate.

get_char/2: PREDICATE

No further documentation available for this predicate.

peek_char/1: PREDICATE

No further documentation available for this predicate.

peek_char/2: PREDICATE

No further documentation available for this predicate.

put_char/1: PREDICATE

No further documentation available for this predicate.

put_char/2: PREDICATE

No further documentation available for this predicate.

36 iso_incomplete (library)

36.1 Usage and interface (`iso_incomplete`)

- **Library usage:**
  `:- use_module(library(iso_incomplete)).`
- **Exports:**
  - *Predicates:*
    `open/4`, `close/2`, `stream_property/2`.

36.2 Documentation on exports (`iso_incomplete`)

open/4: PREDICATE

No further documentation available for this predicate.

close/2: PREDICATE

No further documentation available for this predicate.

stream_property/2: PREDICATE

No further documentation available for this predicate.

37 iso_misc (library)

Version: 0.4#5 (1998/2/24)

37.1 Usage and interface (iso_misc)

- **Library usage:**
  `:- use_module(library(iso_misc)).`
- **Exports:**
  - *Predicates:*
    `\=/2`, `once/1`, `compound/1`, `sub_atom/5`, `unify_with_occurs_check/2`.
- **Other modules used:**
  - *System library modules:*
    `between`.

37.2 Documentation on exports (iso_misc)

\=/2: PREDICATE

No further documentation available for this predicate.

once/1: PREDICATE

No further documentation available for this predicate.

Meta-predicate with arguments: `once(goal)`.

compound/1: PREDICATE

No further documentation available for this predicate.

sub_atom/5: PREDICATE

No further documentation available for this predicate.

unify_with_occurs_check/2: PREDICATE

No further documentation available for this predicate.

38 Term input

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)
Version of last change: 1.5#100 (2000/3/30, 18:1:28 CEST)

38.1 Usage and interface (read)

- **Library usage:**
  `:- use_module(library(read)).`
- **Exports:**
  - *Predicates:*
    `read/1`, `read/2`, `read_term/2`, `read_term/3`, `read_top_level/3`, `second_prompt/2`.
- **Other modules used:**
  - *System library modules:*
    `tokenize`, `operators`, `lists`, `assertions/doc_props`.

38.2 Documentation on exports (read)

read/1: PREDICATE

`read(Term)`

Like `read(Stream,Term)` with `Stream` associated to the current standard input.

read/2: PREDICATE

Usage: `read(+Stream,?(Term))`

- *Description:* The next term, delimited by a full-stop (i.e., a `.` followed by either a space or a control character), is read from `Stream` and is unified with `Term`. The syntax of the term must agree with current operator declarations. If the end of `Stream` has been reached, `Term` is unified with the term `end_of_file`. Further calls to `read/2` for the same stream will then cause an error, unless the stream is connected to the terminal (in which case a prompt is opened on the terminal).
- *The following properties should hold globally:*
  Documentation is still incomplete: `read(+Stream,?(Term))` may not conform the functionality documented. (`doc_props:doc_incomplete/1`)

read_term/2: PREDICATE

No further documentation available for this predicate.

read_term/3: PREDICATE

No further documentation available for this predicate.

read_top_level/3: PREDICATE

No further documentation available for this predicate.

second_prompt/2: PREDICATE

No further documentation available for this predicate.

39 Structured stream handling

Version: 0.5#15 (1998/6/9, 16:30:53 MET DST)

39.1 Usage and interface (streams)

- **Library usage:**
  `:- use_module(library(streams)).`
- **Exports:**
  - *Predicates:*
    `open_null_stream/1`, `open_input/2`, `close_input/1`, `open_output/2`, `close_output/1`.

39.2 Documentation on exports (streams)

open_null_stream/1: PREDICATE

No further documentation available for this predicate.

open_input/2: PREDICATE

No further documentation available for this predicate.

close_input/1: PREDICATE

No further documentation available for this predicate.

open_output/2: PREDICATE

No further documentation available for this predicate.

close_output/1: PREDICATE

No further documentation available for this predicate.

40 Term output

Author(s): Adapted from shared code written by Richard A. O'Keefe. Changes by Mats Carlsson, Daniel Cabeza, and Manuel Hermenegildo..

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 0.5#49 (1998/7/14, 20:3:29 MET DST)

This library provides different predicates for term output, additional to the kernel predicates `display/1`- `display/2` and `displayq/1`-`displayq/2`. All the predicates defined in ISO-Prolog are included, plus other traditionally provided by Prolog Implementations. Output predicates are provided in two versions: one that uses the current output stream and other in which the stream is specified explicitly, as an additional first argument.

40.1 Usage and interface (`write`)

- **Library usage:**
  `:- use_module(library(write)).`
- **Exports:**
  - *Predicates:*
    `write_term/3`, `write_term/2`, `write/2`, `write/1`, `write_list1/1`, `writeq/2`, `writeq/1`, `write_canonical/2`, `write_canonical/1`, `print/2`, `print/1`, `portray_clause/2`, `portray_clause/1`, `numbervars/3`, `prettyvars/1`, `printable_char/1`.
  - *Properties:*
    `write_option/1`.
  - *Multifiles:*
    `define_flag/3`, `portray_attribute/2`, `portray/1`.
- **Other modules used:**
  - *System library modules:*
    `operators`, `sort`.

40.2 Documentation on exports (`write`)

write_term/3: PREDICATE

Usage: `write_term(@(Stream),?(Term),+OptList)` ⟨• ISO •⟩

- *Description:* Outputs the term `Term` to the stream `Stream`, with the list of write-options `OptList`. See `write_option/1` type for default options.
- *The following properties should hold upon exit:*
  `@(Stream)` is an open stream. (`streams_basic:stream/1`)
  `?(Term)` is any term. (`basic_props:term/1`)
  `+OptList` is a list of `write_options`. (`basic_props:list/2`)

write_term/2: PREDICATE

Usage: `write_term(?(Term),+OptList)` ⟨• ISO •⟩

- *Description:* Behaves like `current_output(S), write_term(S,Term,OptList)`.

- *The following properties should hold upon exit:*
  `?(Term)` is any term. (`basic_props:term/1`)
  `+OptList` is a list of `write_options`. (`basic_props:list/2`)

write_option/1: PROPERTY

`Opt` is a valid write option which affects the predicate `write_term/3` and similar ones. Possible write_options are:

- **quoted(***bool***):** If *bool* is `true`, atoms and functors that can't be read back by `read_term/3` are quoted, if it is `false`, each atom and functor is written as its name. Default value is `false`.
- **ignore_ops(***flag***):** If *flag* is `true`, each compound term is output in functional notation, if it is `ops`, curly bracketed notation and list notation is enabled when outputing compound terms, if it is `false`, also operator notation is enabled when outputing compound terms. Default value is `false`.
- **numbervars(***bool***):** If *bool* is `true`, a term of the form `'$VAR'(N)` where `N` is an integer, is output as a variable name consisting of a capital letter possibly followed by an integer, a term of the form `'$VAR'(Atom)` where `Atom` is an atom, as this atom (without quotes), and a term of the form `'$VAR'(String)` where `String` is a character string, as the atom corresponding to this character string. See predicates `numbervars/3` and `prettyvars/1`. If *bool* is `false` this cases are not treated in any special way. Default value is `false`.
- **portrayed(***bool***):** If *bool* is `true`, then call multifile predicates `portray/1` and `portray_attribute/1`, to provide the user handlers for pretty printing some terms. `portray_attribute/1` is called whenever an attributed variable is to be printed, `portray/1` is called whenever a non-variable term is to be printed. If either call succeeds, then it is assumed that the term has been output, else it is printed as usual. If *bool* is `false`, these predicates are not called. Default value is `false`. This option is set by the toplevel when writting the final values of variables, and by the debugging package when writting the goals in the tracing messages. Thus you can vary the forms of these messages if you wish.
- **max_depth(***depth***):** *depth* is a positive integer or cero. If it is positive, it denotes the depth limit on printing compound terms. If it is cero, there is no limit. Default value is `0` (no limit).
- **priority(***prio***):** *prio* is an integer between 1 and 1200. If the term to be printed has higher priority than *prio*, it will be printed parenthesized. Default value is 1200 (no term parenthesized).

.

Usage: `write_option(Opt)`

- *Description:* `Opt` is a valid write option.

write/2: PREDICATE

Usage: `write(@(Stream),?(Term))` ⟨• ISO •⟩

- *Description:* Behaves like `write_term(Stream, Term, [numbervars(true)])`.
- *The following properties should hold upon exit:*
  `@(Stream)` is an open stream. (`streams_basic:stream/1`)
  `?(Term)` is any term. (`basic_props:term/1`)

write/1: PREDICATE

Usage: `write(?(Term))` ⟨• ISO •⟩

- *Description:* Behaves like `current_output(S), write(S,Term)`.
- *The following properties should hold upon exit:*
  `?(Term)` is any term. (`basic_props:term/1`)

write_list1/1: PREDICATE

Usage:

- *Description:* Writes a list to current output one element in each line.
- *The following properties should hold at call time:*
  `Arg1` is a list. (`basic_props:list/1`)
- *The following properties should hold upon exit:*
  `Arg1` is a list. (`basic_props:list/1`)

writeq/2: PREDICATE

Usage: `writeq(@(Stream),?(Term))` ⟨• ISO •⟩

- *Description:* Behaves like `write_term(Stream, Term, [quoted(true), numbervars(true)])`.
- *The following properties should hold upon exit:*
  `@(Stream)` is an open stream. (`streams_basic:stream/1`)
  `?(Term)` is any term. (`basic_props:term/1`)

writeq/1: PREDICATE

Usage: `writeq(?(Term))` ⟨• ISO •⟩

- *Description:* Behaves like `current_output(S), writeq(S,Term)`.
- *The following properties should hold upon exit:*
  `?(Term)` is any term. (`basic_props:term/1`)

write_canonical/2: PREDICATE

Usage: `write_canonical(@(Stream),?(Term))` ⟨• ISO •⟩

- *Description:* Behaves like `write_term(Stream, Term, [quoted(true), ignore_ops(true)])`. The output of this predicate can always be parsed by `read_term/2` even if the term contains special characters or if operator declarations have changed.
- *The following properties should hold upon exit:*
  `@(Stream)` is an open stream. (`streams_basic:stream/1`)
  `?(Term)` is any term. (`basic_props:term/1`)

write_canonical/1: PREDICATE

Usage: `write_canonical(?(Term))` ⟨• ISO •⟩

- *Description:* Behaves like `current_output(S), write_canonical(S,Term)`.
- *The following properties should hold upon exit:*
  `?(Term)` is any term. (`basic_props:term/1`)

print/2: PREDICATE

Usage: `print(@(Stream),?(Term))`

- *Description:* Behaves like `write_term(Stream, Term, [numbervars(true), portrayed(true)])`.
- *The following properties should hold upon exit:*

  `@(Stream)` is an open stream. (`streams_basic:stream/1`)

  `?(Term)` is any term. (`basic_props:term/1`)

print/1: PREDICATE

Usage: `print(?(Term))`

- *Description:* Behaves like `current_output(S), print(S,Term)`.
- *The following properties should hold upon exit:*

  `?(Term)` is any term. (`basic_props:term/1`)

portray_clause/2: PREDICATE

Usage: `portray_clause(@(Stream),?(Clause))`

- *Description:* Outputs the clause `Clause` onto `Stream`, pretty printing its variables and using indentation, including a period at the end. This predicate is used by `listing/0`.
- *The following properties should hold upon exit:*

  `@(Stream)` is an open stream. (`streams_basic:stream/1`)

  `?(Clause)` is any term. (`basic_props:term/1`)

portray_clause/1: PREDICATE

Usage: `portray_clause(?(Clause))`

- *Description:* Behaves like `current_output(S), portray_clause(S,Term)`.
- *The following properties should hold upon exit:*

  `?(Clause)` is any term. (`basic_props:term/1`)

numbervars/3: PREDICATE

Usage: `numbervars(?(Term),+N,?(M))`

- *Description:* Unifies each of the variables in term `Term` with a term of the form `'$VAR'(I)` where `I` is an integer from `N` onwards. `M`-1 is unified with the last integer used. If the resulting term is output with a write option `numbervars(true)`, in the place of the variables in the original term will be printed a variable name consisting of a capital letter possibly followed by an integer. When `N` is 0 you will get the variable names A, B, ..., Z, A1, B1, etc.
- *The following properties should hold upon exit:*

  `?(Term)` is any term. (`basic_props:term/1`)

  `+N` is currently instantiated to an integer. (`term_typing:integer/1`)

  `?(M)` is currently instantiated to an integer. (`term_typing:integer/1`)

prettyvars/1: PREDICATE

Usage: `prettyvars(?(Term))`

- *Description:* Similar to `numbervars(Term,0,_)`, except that singleton variables in `Term` are unified with `'$VAR'('_')`, so that when the resulting term is output with a write option `numbervars(true)`, in the place of singleton variables `_` is written. This predicate is used by `portray_clause/2`.
- *The following properties should hold upon exit:*

  `?(Term)` is any term. (`basic_props:term/1`)

printable_char/1: PREDICATE

No further documentation available for this predicate.

40.3 Documentation on multifiles (`write`)

define_flag/3: PREDICATE

Defines flags as follows:

```
define_flag(write_strings,[on,off],off).
```

(See Chapter 25 [Changing system behaviour and various flags], page 131)

The predicate is *multifile*.

portray_attribute/2: PREDICATE

No further documentation available for this predicate.

The predicate is *multifile*.

portray/1: PREDICATE

The predicate is *multifile*.

Usage: `portray(?(Term))`

- *Description: A user defined predicate.* This should either print the `Term` and succeed, or do nothing and fail. In the latter case, the default printer (`write/1`) will print the `Term`.

PART IV - Classic Prolog library (classic)

This part documents some Ciao libraries which provide additional predicates and functionalities that, despite not being in the ISO standard, are present in many popular Prolog systems. This includes definite clause grammars (DCGs), "Quintus-style" internal database, list processing predicates, dictionaries, string processing, DEC-10 Prolog-style input/output, formatted output, dynamic loading of modules, activation of operators at run-time, etc.

41 Classic Prolog package

This library package allows the use of certain Prolog features which have become sort of 'classical' from many Prolog implementations. These include definite clause grammars and some classical predicates like `append/3`. The libraries listed below define these predicates, and the following chapters describe them.

41.1 Usage and interface (`classic`)

- **Library usage:**
  `:- use_package(classic).`
  or
  `:- module(...,...,[classic]).`
- **New operators defined:**
  `-->/2` [1200,xfx].
- **Other modules used:**
  - *System library modules:*
    `operators`, `old_database`, `lists`, `sort`, `dict`, `strings`, `dec10_io`, `format`, `ttyout`, `dynmods`.

42 Definite clause grammars

This library package allows the use of DCGs (Definite Clause Grammars) [Col78,PW80] in a Ciao module/program.

Definite clause grammars are an extension of the well-known context-free grammars. Prolog's grammar rules provide a convenient notation for expressing definite clause grammars. A DCG rule in Prolog takes the general form

```
head --> body.
```

meaning "a possible form for `head` is `body`". Both `body` and `head` are sequences of one or more items linked by the standard Prolog conjunction operator `,`.

Definite clause grammars extend context-free grammars in the following ways:

1. A non-terminal symbol may be any Prolog term (other than a variable or number).
2. A terminal symbol may be any Prolog term. To distinguish terminals from non-terminals, a sequence of one or more terminal symbols is written within a grammar rule as a Prolog list. An empty sequence is written as the empty list `[]`. If the terminal symbols are ASCII character codes, such lists can be written (as elsewhere) as strings. An empty sequence is written as the empty list, `[]` or `""`.
3. Extra conditions, in the form of Prolog procedure calls, may be included in the right-hand side of a grammar rule. Such procedure calls are written enclosed in `{}` brackets.
4. The left-hand side of a grammar rule consists of a non-terminal, optionally followed by a sequence of terminals (again written as a Prolog list).
5. Alternatives may be stated explicitly in the right-hand side of a grammar rule, using the disjunction operator `;` or `|` as in Prolog.
6. The cut symbol may be included in the right-hand side of a grammar rule, as in a Prolog clause. The cut symbol does not need to be enclosed in `{}` brackets.

As an example, here is a simple grammar which parses an arithmetic expression (made up of digits and operators) and computes its value.

```
expr(Z) --> term(X), "+", expr(Y), {Z is X + Y}.
expr(Z) --> term(X), "-", expr(Y), {Z is X - Y}.
expr(X) --> term(X).

term(Z) --> number(X), "*", term(Y), {Z is X * Y}.
term(Z) --> number(X), "/", term(Y), {Z is X / Y}.
term(Z) --> number(Z).

number(C) --> "+", number(C).
number(C) --> "-", number(X), {C is -X}.
number(X) --> [C], {"0"=<C, C=<"9", X is C - "0"}.
```

In the last rule, `C` is the ASCII code of some digit.

The query

```
| ?- expr(Z, "-2+3*5+1", []).
```

will compute `Z`=14. The two extra arguments are explained below.

Now, in fact, grammar rules are merely a convenient "syntactic sugar" for ordinary Prolog clauses. Each grammar rule takes an input string, analyses some initial portion, and produces the remaining portion (possibly enlarged) as output for further analysis. The arguments required for the input and output strings are not written explicitly in a grammar rule, but the syntax implicitly defines them. We now show how to translate grammar rules into ordinary clauses by making explicit the extra arguments.

A rule such as

```
p(X) --> q(X).
```

translates into

```
p(X, S0, S) :- q(X, S0, S).
```

If there is more than one non-terminal on the right-hand side, as in

```
p(X, Y) -->
        q(X),
        r(X, Y),
        s(Y).
```

then corresponding input and output arguments are identified, as in

```
p(X, Y, S0, S) :-
        q(X, S0, S1),
        r(X, Y, S1, S2),
        r(Y, S2, S).
```

Terminals are translated using the built-in predicate `term_basic:'C'/3` (this predicate is not normally useful in itself; it has been given the name `'C'` simply to avoid using up a more useful name). Then, for instance

```
p(X) --> [go,to], q(X), [stop].
```

is translated by

```
p(X, S0, S) :-
        'C'(S0, go, S1),
        'C'(S1, to, S2),
        q(X, S2, S3),
        'C'(S3, stop, S).
```

Extra conditions expressed as explicit procedure calls naturally translate as themselves, e.g.

```
p(X) --> [X], {integer(X), X>0}, q(X).
```

translates to

```
p(X, S0, S) :-
        'C'(S0, X, S1),
        integer(X),
        X>0,
        q(X, S1, S).
```

Similarly, a cut is translated literally.

Terminals on the left-hand side of a rule translate into an explicit list in the output argument of the main non-terminal, e.g.

```
is(N), [not] --> [aint].
```

becomes

```
is(N, S0, [not|S]) :- 'C'(S0, aint, S).
```

Disjunction has a fairly obvious translation, e.g.

```
args(X, Y) -->
        (   dir(X), [to], indir(Y)
        ;   indir(Y), dir(X)
        ).
```

translates to

```
args(X, Y, S0, S) :-
        (   dir(X, S0, S1),
            'C'(S1, to, S2),
            indir(Y, S2, S)
```

```
        ;   indir(Y, S0, S1),
            dir(X, S1, S)
        ).
```

42.1 Usage and interface (dcg)

- **Library usage:**

  `:- use_package(dcg).`

  or

  `:- module(...,...,[dcg]).`

43 Definite clause grammars (expansion)

Version: 0.4#5 (1998/2/24)

43.1 Usage and interface (`dcg_expansion`)

- **Library usage:**
  `:- use_module(library(dcg_expansion)).`
- **Exports:**
  - *Predicates:*
    `phrase/2`, `phrase/3`, `dcg_translation/2`.
- **Other modules used:**
  - *System library modules:*
    `terms`, `assertions/doc_props`.

43.2 Documentation on exports (`dcg_expansion`)

phrase/2: PREDICATE

`phrase(Phrase,List)`

Like `phrase(Phrase,List,[])`.

Meta-predicate with arguments: `phrase(goal,?)`.

phrase/3: PREDICATE

Meta-predicate with arguments: `phrase(goal,?,?)`.

Usage: `phrase(+Phrase,?(List),?(Remainder))`

- *Description:* The list `List` is a phrase of type `Phrase` (according to the current grammar rules), where `Phrase` is either a non-terminal or more generally a grammar rule body. `Remainder` is what remains of the list after a phrase has been found.
- *The following properties should hold globally:*
  Documentation is still incomplete: `phrase(+Phrase,?(List),?(Remainder))` may not conform the functionality documented. (`doc_props:doc_incomplete/1`)

dcg_translation/2: PREDICATE

Performs the code expansion of source clauses that use DCGs.

44 Quintus-like internal database

Version: 0.4#5 (1998/2/24)

The predicates described in this section were introduced in early implementations of Prolog to provide efficient means of performing operations on large quantities of data. The introduction of indexed dynamic predicates have rendered these predicates obsolete, and the sole purpose of providing them is to support existing code. There is no reason whatsoever to use them in new code.

These predicates store arbitrary terms in the database without interfering with the clauses which make up the program. The terms which are stored in this way can subsequently be retrieved via the key on which they were stored. Many terms may be stored on the same key, and they can be individually accessed by pattern matching. Alternatively, access can be achieved via a special identifier which uniquely identifies each recorded term and which is returned when the term is stored.

44.1 Usage and interface (`old_database`)

- **Library usage:**
  `:- use_module(library(old_database)).`
- **Exports:**
  - *Predicates:*
    `recorda/3`, `recordz/3`, `recorded/3`, `current_key/2`.

44.2 Documentation on exports (`old_database`)

recorda/3: PREDICATE

`recorda(+Key,?(Term),-(Ref))`

The term `Term` is recorded in the internal database as the first item for the key `Key`, where `Ref` is its implementation-defined identifier. The key must be given, and only its principal functor is significant. Any uninstantiated variables in the `Term` will be replaced by new private variables, along with copies of any subgoals blocked on these variables.

recordz/3: PREDICATE

`recordz(+Key,?(Term),-(Ref))`

Like `recorda/3`, except that the new term becomes the *last* item for the key `Key`.

recorded/3: PREDICATE

`recorded(?(Key),?(Term),?(Ref))`

The internal database is searched for terms recorded under the key `Key`. These terms are successively unified with `Term` in the order they occur in the database. At the same time, `Ref` is unified with the implementation-defined identifier uniquely identifying the recorded item. If the key is instantiated to a compound term, only its principal functor is significant. If the key is uninstantiated, all terms in the database are successively unified with `Term` in the order they occur.

current_key/2: PREDICATE

`current_key(?(KeyName),?(KeyTerm))`

`KeyTerm` is the most general form of the key for a currently recorded term, and `KeyName` is the name of that key. This predicate can be used to enumerate in undefined order all keys for currently recorded terms through backtracking.

45 List processing

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)
Version of last change: 1.5#41 (2000/2/4, 13:34:24 CET)
This module provides a set of predicates for list processing.

45.1 Usage and interface (`lists`)

- **Library usage:**
  `:- use_module(library(lists)).`
- **Exports:**
  - *Predicates:*
    `nonsingle/1`, `append/3`, `reverse/2`, `reverse/3`, `delete/3`, `select/3`, `length/2`, `nth/3`, `add_after/4`, `add_before/4`, `dlist/3`, `list_concat/2`, `list_insert/2`, `insert_last/3`, `contains_ro/2`, `contains1/2`, `nocontainsx/2`, `last/2`, `list_lookup/3`, `list_lookup/4`, `intset_insert/3`, `intset_delete/3`, `intset_in/2`, `intset_sequence/3`, `intersection/3`, `union/3`, `difference/3`, `equal_lists/2`, `list_to_list_of_lists/2`, `powerset/2`.
  - *Properties:*
    `list1/2`, `sublist/2`, `subordlist/2`.

45.2 Documentation on exports (`lists`)

nonsingle/1: PREDICATE

Usage: `nonsingle(X)`
- *Description:* `X` is not a singleton.

append/3: PREDICATE

Usage: `append(Xs,Ys,Zs)`
- *Description:* `Zs` is `Ys` appended to `Xs`.

reverse/2: PREDICATE

Usage: `reverse(Xs,Ys)`
- *Description:* Reverses the order of elements in `Xs`.
- *The following properties should hold at call time:*
  `Xs` is a list. (`basic_props:list/1`)
  `Ys` is a free variable. (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  `Xs` is a list. (`basic_props:list/1`)
  `Ys` is a list. (`basic_props:list/1`)

reverse/3: PREDICATE

No further documentation available for this predicate.

delete/3: PREDICATE

Usage: `delete(L1,E,L2)`

- *Description:* `L2` is `L1` without the ocurrences of `E`.

select/3: PREDICATE

Usage: `select(X,Xs,Ys)`

- *Description:* `Xs` and `Ys` have the same elements except for one occurrence of `X`.

length/2: PREDICATE

Usage 1: `length(L,N)`

- *Description:* Computes the length of `L`.
- *The following properties should hold at call time:*
  `L` is a list. (`basic_props:list/1`)
  `N` is a free variable. (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  `L` is a list. (`basic_props:list/1`)
  `N` is currently instantiated to an integer. (`term_typing:integer/1`)

Usage 2: `length(L,N)`

- *Description:* Outputs `L` of length `N`.
- *The following properties should hold at call time:*
  `L` is a free variable. (`term_typing:var/1`)
  `N` is currently instantiated to an integer. (`term_typing:integer/1`)
- *The following properties should hold upon exit:*
  `L` is a list. (`basic_props:list/1`)
  `N` is currently instantiated to an integer. (`term_typing:integer/1`)

Usage 3: `length(L,N)`

- *Description:* Checks that `L` is of length `N`.
- *The following properties should hold at call time:*
  `L` is a list. (`basic_props:list/1`)
  `N` is currently instantiated to an integer. (`term_typing:integer/1`)
- *The following properties should hold upon exit:*
  `L` is a list. (`basic_props:list/1`)
  `N` is currently instantiated to an integer. (`term_typing:integer/1`)

nth/3: PREDICATE

`nth(N,List,Elem)`

`N` is the position in `List` of `Elem`. `N` counts from one.

Usage 1: `nth(+int,?(list),?(term))`

– *Description:* Unifies `Elem` and the `N`th element of `List`.

Usage 2: `nth(-(int),?(list),?(term))`

– *Description:* Finds the positions where `Elem` is in `List`. Positions are found in ascending order.

add_after/4: PREDICATE

Usage: `add_after(+L0,+E0,+E,-(L))`

– *Description:* Adds element `E` after element `E0` (or at end) to list `L0` returning in `L` the new list (uses term comparison).

add_before/4: PREDICATE

No further documentation available for this predicate.

list1/2: PROPERTY

Meta-predicate with arguments: `list1(?,pred(1))`.

Usage: `list1(X,Y)`

– *Description:* `X` is a list of `Y`s of at least one element.

dlist/3: PREDICATE

Usage: `dlist(List,DList,Tail)`

– *Description:* `List` is the result of removing `Tail` from the end of `DList` (makes a difference list from a list).

list_concat/2: PREDICATE

Usage: `list_concat(LL,L)`

– *Description:* `L` is the concatenation of all the lists in `LL`.

– *Call and exit should be compatible with:*

`LL` is a list of `lists`. (`basic_props:list/2`)

`L` is a list. (`basic_props:list/1`)

list_insert/2: PREDICATE

Usage: `list_insert(-(List),+Term)`

– *Description:* Adds `Term` to the end of `List` if there is no element in `List` identical to `Term`.

insert_last/3: PREDICATE

Usage: `insert_last(+L0,+E,-(L))`

– *Description:* Adds element `E` at end of list `L0` returning `L`.

contains_ro/2: PREDICATE

Usage:

– *Description:* Impure membership (does not instantiate a variable in its first argument.

contains1/2: PREDICATE

No further documentation available for this predicate.

nocontainsx/2: PREDICATE

Usage: `nocontainsx(L,X)`

– *Description:* `X` is not identical to any element of `L`.

last/2: PREDICATE

Usage: `last(L,X)`

– *Description:* Adds element `X` to incomplete (i.e., ended in a variable) list `L`.

list_lookup/3: PREDICATE

No further documentation available for this predicate.

list_lookup/4: PREDICATE

Usage: `list_lookup(List,Functor,Key,Value)`

– *Description:* Look up `Functor`(`Key`,`Value`) pair in variable ended key-value pair list `L` or else add it at the end.

intset_insert/3: PREDICATE

No further documentation available for this predicate.

intset_delete/3: PREDICATE

No further documentation available for this predicate.

intset_in/2: PREDICATE

No further documentation available for this predicate.

intset_sequence/3: PREDICATE

No further documentation available for this predicate.

intersection/3: PREDICATE

Usage: `intersection(+List1,+List2,-(List))`

- *Description:* `List` has the elements which are both in `List1` and `List2`.

union/3: PREDICATE

Usage: `union(+List1,+List2,-(List))`

- *Description:* `List` has the elements which are in `List1` followed by the elements which are in `List2` but not in `List1`.

difference/3: PREDICATE

Usage: `difference(+List1,+List2,-(List))`

- *Description:* `List` has the elements which are in `List1` but not in `List2`.

sublist/2: PROPERTY

Usage: `sublist(List1,List2)`

- *Description:* `List2` contains all the elements of `List1`.
- *If the following properties should hold at call time:*
  `List2` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

subordlist/2: PROPERTY

Usage: `subordlist(List1,List2)`

- *Description:* `List2` contains all the elements of `List1` in the same order.
- *If the following properties should hold at call time:*
  `List2` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

equal_lists/2: PREDICATE

Usage: `equal_lists(+List1,+List2)`

- *Description:* `List1` has all the elements of `List2`, and vice versa.

list_to_list_of_lists/2: PREDICATE

Usage: `list_to_list_of_lists(+List,-(LList))`

- *Description:* `LList` is the list of one element lists with elements of `List`.

powerset/2: PREDICATE

Usage: `powerset(+List,-(LList))`

- *Description:* `LList` is the powerset of `List`, i.e., the list of all lists which have elements of `List`. If `List` is ordered, `LList` and all its elements are ordered.

46 Sorting lists

Author(s): Richard A. O'Keefe. All changes by UPM CLIP Group..
Version: 0.4#5 (1998/2/24)

46.1 Usage and interface (sort)

- **Library usage:**
  `:- use_module(library(sort)).`
- **Exports:**
  - *Predicates:*
    `sort/2`, `keysort/2`.

46.2 Documentation on exports (sort)

sort/2: PREDICATE

`sort(List1,List2)`

The elements of `List1` are sorted into the standard order (see Chapter 18 [Comparing terms], page 97) and any identical elements are merged, yielding `List2`. The time and space complexity of this operation is at worst `O(N lg N)` where `N` is the length of `List1`.

Usage: `sort(+list,?(list))`

- *Description:* `List2` is the sorted list corresponding to `List1`.

keysort/2: PREDICATE

`keysort(List1,List2)`

`List1` is sorted into order according to the value of the *keys* of its elements, yielding the list `List2`. No merging takes place. This predicate is *stable*, i.e., if an element `A` occurs before another element `B` *with the same key* in the input, then `A` will occur before `B` also in the output. The time and space complexity of this operation is at worst `O(N lg N)` where `N` is the length of `List1`.

Usage: `keysort(+keylist,?(keylist))`

- *Description:* `List2` is the (key-)sorted list corresponding to `List1`.

46.3 Documentation on internals (sort)

keypair/1: REGTYPE

Usage: `keypair(P)`

- *Description:* `P` is a pair of the form `K-_`, where `K` is considered the *key*.

47 Dictionaries

Version: 0.4#5 (1998/2/24)

This module provides predicates for implementing dictionaries. Such dictionaries are currently implemented as ordered binary trees.

47.1 Usage and interface (`dict`)

- **Library usage:**
  `:- use_module(library(dict)).`
- **Exports:**
  - *Predicates:*
    `dic_node/2`, `dic_lookup/3`, `dic_lookup/4`, `dic_get/3`, `dic_replace/4`.

47.2 Documentation on exports (`dict`)

dic_node/2: PREDICATE

No further documentation available for this predicate.

dic_lookup/3: PREDICATE

No further documentation available for this predicate.

dic_lookup/4: PREDICATE

No further documentation available for this predicate.

dic_get/3: PREDICATE

No further documentation available for this predicate.

dic_replace/4: PREDICATE

No further documentation available for this predicate.

48 String processing

Author(s): Daniel Cabeza.

Version: 0.4#5 (1998/2/24)

This module provides predicates for doing input/output with strings (character code lists) and for including in grammars defining strings.

48.1 Usage and interface (strings)

- **Library usage:**
  `:- use_module(library(strings)).`
- **Exports:**
  - *Predicates:*
    `get_line/2`, `get_line/1`, `write_string/2`, `write_string/1`, `whitespace/2`, `whitespace0/2`, `string/3`.

48.2 Documentation on exports (strings)

get_line/2: PREDICATE

`get_line(Stream,Line)`

Reads from `Stream` a line of text and unifies `Line` with it. The end of the line can have UNIX [10] or MS-DOS [13 10] termination, which is not included in `Line`. At EOF, the term end_of_file is returned.

get_line/1: PREDICATE

`get_line(Line)`

Behaves like `current_input(S), get_line(S,Line)`.

write_string/2: PREDICATE

`write_string(Stream,String)`

Writes `String` onto `Stream`.

write_string/1: PREDICATE

`write_string(String)`

Behaves like `current_input(S), write_string(S, String)`.

whitespace/2: PREDICATE

`whitespace(String,Rest)`

In a grammar rule, as `whitespace/0`, represents whitespace (a positive number of space (32), tab (9), newline (10) or return (13) characters). Thus, `Rest` is a proper suffix of `String` with one or more whitespace characters removed. An example of use would be:

```
attrs([]) --> ""
attrs([N|Ns]) -->
    whitespace,
    attr(N),
    attrs(Ns).
```

whitespace0/2: PREDICATE

`whitespace0(String,Rest)`

In a grammar rule, as `whitespace0/0`, represents possible whitespace (any number of space (32), tab (9), newline (10) or return (13) characters). Thus, `Rest` is `String` or a proper suffix of `String` with one or more whitespace characters removed. An example of use would be:

```
assignment(N,V) -->
    variable_name(N), whitespace0, "=", whitespace0, value(V).
```

string/3: PREDICATE

`string(String,Head,Tail)`

In a grammar rule, as `string/1`, represents literally `String`. An example of use would be:

```
double(A) -->
        string(A),
        string(A).
```

48.3 Documentation on internals (strings)

line/1: PROPERTY

A property, defined as follows:

```
line(L) :-
        string(L).
line(end_of_file).
```

49 DEC-10 Prolog file IO

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)
Version of last change: 0.4#5 (1998/2/24)
This module implements the support for DEC-10 Prolog style file I/O.

49.1 Usage and interface (`dec10_io`)

- **Library usage:**
  `:- use_module(library(dec10_io)).`
- **Exports:**
  - *Predicates:*
    `see/1`, `seeing/1`, `seen/0`, `tell/1`, `telling/1`, `told/0`, `close_file/1`.
- **Other modules used:**
  - *System library modules:*
    `streams`.

49.2 Documentation on exports (`dec10_io`)

see/1: PREDICATE
No further documentation available for this predicate.

seeing/1: PREDICATE
No further documentation available for this predicate.

seen/0: PREDICATE
No further documentation available for this predicate.

tell/1: PREDICATE
No further documentation available for this predicate.

telling/1: PREDICATE
No further documentation available for this predicate.

told/0: PREDICATE
No further documentation available for this predicate.

close_file/1: PREDICATE
No further documentation available for this predicate.

50 Formatted output

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.3#27 (1999/7/9, 20:25:50 MEST)

The `format` family of predicates is due to Quintus Prolog. They act as a Prolog interface to the C `stdio` function `printf()`, allowing formatted output.

Output is formatted according to an output pattern which can have either a format control sequence or any other character, which will appear verbatim in the output. Control sequences act as place-holders for the actual terms that will be output. Thus

```
| ?- format("Hello ~q!",world).
```

will print `Hello world!`.

If there is only one item to print it may be supplied alone. If there are more they have to be given as a list. If there are none then an empty list should be supplied. There has to be as many items as control characters.

The character ~ introduces a control sequence. To print a ~ verbatim just repeat it:

```
| ?- format("Hello ~~world!", []).
```

will result in `Hello ~world!`.

A format may be spread over several lines. The control sequence followed by a ⟨LFD⟩ will translate to the empty string:

```
| ?- format("Hello world!", []).
```

will result in `Hello world!`.

50.1 Usage and interface (`format`)

- **Library usage:**
  `:- use_module(library(format)).`
- **Exports:**
  - *Predicates:*
    `format/2`, `format/3`.
  - *Regular Types:*
    `format_control/1`.
- **Other modules used:**
  - *System library modules:*
    `write`, `assertions/doc_props`.

50.2 Documentation on exports (`format`)

format/2: PREDICATE

Usage: `format(Format,Arguments)`

- *Description:* Print `Arguments` onto current output stream according to format `Format`.

- *Call and exit should be compatible with:*
  `Format` is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with `name/2`. (`format:format_control/1`)

format/3: PREDICATE

Usage: `format(+Stream,Format,Arguments)`

- *Description:* Print `Arguments` onto `Stream` according to format `Format`.
- *Call and exit should be compatible with:*
  `Format` is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with `name/2`. (`format:format_control/1`)

format_control/1: REGTYPE

The general format of a control sequence is `~NC`. The character `C` determines the type of the control sequence. `N` is an optional numeric argument. An alternative form of `N` is `*`. `*` implies that the next argument in `Arguments` should be used as a numeric argument in the control sequence. Example:

```
| ?- format("Hello~4cworld!", [0'x]).
```

and

```
| ?- format("Hello~*cworld!", [4,0'x]).
```

both produce

```
Helloxxxxworld!
```

The following control sequences are available.

- ~a The argument is an atom. The atom is printed without quoting.
- ~Nc (Print character.) The argument is a number that will be interpreted as an ASCII code. `N` defaults to one and is interpreted as the number of times to print the character.
- ~Ne
- ~NE
- ~Nf
- ~Ng
- ~NG (Print float). The argument is a float. The float and `N` will be passed to the C `printf()` function as

  ```
  printf("%.Ne", Arg)
  printf("%.NE", Arg)
  printf("%.Nf", Arg)
  printf("%.Ng", Arg)
  printf("%.NG", Arg)
  ```

  If `N` is not supplied the action defaults to

  ```
  printf("%e", Arg)
  printf("%E", Arg)
  printf("%f", Arg)
  printf("%g", Arg)
  printf("%G", Arg)
  ```

- ~Nd (Print decimal.) The argument is an integer. `N` is interpreted as the number of digits after the decimal point. If `N` is 0 or missing, no decimal point will be printed. Example:

```
| ?- format("Hello ~1d world!", [42]).
| ?- format("Hello ~d world!", [42]).
```

will print as

```
Hello 4.2 world!
Hello 42 world!
```

respectively.

- ~ND (Print decimal.) The argument is an integer. Identical to `~Nd` except that `,` will separate groups of three digits to the left of the decimal point. Example:

  ```
  | ?- format("Hello ~1D world!", [12345]).
  ```

  will print as

  ```
  Hello 1,234.5 world!
  ```

- ~Nr (Print radix.) The argument is an integer. `N` is interpreted as a radix. `N` should be `>=` 2 and `<=` 36. If `N` is missing the radix defaults to 8. The letters `a-z` will denote digits larger than 9. Example:

  ```
  | ?- format("Hello ~2r world!", [15]).
  | ?- format("Hello ~16r world!", [15]).
  ```

  will print as

  ```
  Hello 1111 world!
  Hello f world!
  ```

  respectively.

- ~NR (Print radix.) The argument is an integer. Identical to `~Nr` except that the letters `A-Z` will denote digits larger than 9. Example:

  ```
  | ?- format("Hello ~16R world!", [15]).
  ```

  will print as

  ```
  Hello F world!
  ```

- ~Ns (Print string.) The argument is a list of ASCII codes. Exactly `N` characters will be printed. `N` defaults to the length of the string. Example:

  ```
  | ?- format("Hello ~4s ~4s!", ["new","world"]).
  | ?- format("Hello ~s world!", ["new"]).
  ```

  will print as

  ```
  Hello new  worl!
  Hello new world!
  ```

  respectively.

- ~i (Ignore argument.) The argument may be of any type. The argument will be ignored. Example:

  ```
  | ?- format("Hello ~i~s world!", ["old","new"]).
  ```

  will print as

  ```
  Hello new world!
  ```

- ~k (Print canonical.) The argument may be of any type. The argument will be passed to `write_canonical/2` (Chapter 40 [Term output], page 183). Example:

  ```
  | ?- format("Hello ~k world!", [[a,b,c]]).
  ```

  will print as

  ```
  Hello .(a,.(b,.(c,[]))) world!
  ```

- ~p (print.) The argument may be of any type. The argument will be passed to `print/2` (Chapter 40 [Term output], page 183). Example:

```
| ?- assert((portray([X|Y]) :- print(cons(X,Y)))).
| ?- format("Hello ~p world!", [[a,b,c]]).
```

will print as

```
Hello cons(a,cons(b,cons(c,[]))) world!
```

- ~q (Print quoted.) The argument may be of any type. The argument will be passed to `writeq/2` (Chapter 40 [Term output], page 183). Example:

  ```
  | ?- format("Hello ~q world!", [['A','B']]).
  ```

  will print as

  ```
  Hello ['A','B'] world!
  ```

- ~w (write.) The argument may be of any type. The argument will be passed to `write/2` (Chapter 40 [Term output], page 183). Example:

  ```
  | ?- format("Hello ~w world!", [['A','B']]).
  ```

  will print as

  ```
  Hello [A,B] world!
  ```

- ~Nn (Print newline.) Print `N` newlines. `N` defaults to 1. Example:

  ```
  | ?- format("Hello ~n world!", []).
  ```

  will print as

  ```
  Hello
   world!
  ```

- ~N (Fresh line.) Print a newline, if not already at the beginning of a line.

The following control sequences are also available for compatibility, but do not perform any useful functions.

- `~N|` (Set tab.) Set a tab stop at position `N`, where `N` defaults to the current position, and advance the current position there.
- `~N+` (Advance tab.) Set a tab stop at `N` positions past the current position, where `N` defaults to 8, and advance the current position there.
- `~Nt` (Set fill character.) Set the fill character to be used in the next position movement to `N`, where `N` defaults to ⟨SPC⟩.

Usage: `format_control(C)`

- *Description:* `C` is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with `name/2`.
- *The following properties should hold globally:*

  Documentation is still incomplete: `format_control(C)` may not conform the functionality documented. (`doc_props:doc_incomplete/1`)

51 ttyout (library)

Version: 0.4#5 (1998/2/24)

51.1 Usage and interface (ttyout)

- **Library usage:**
  `:- use_module(library(ttyout)).`
- **Exports:**
  - *Predicates:*
    `ttyget/1`, `ttyget1/1`, `ttynl/0`, `ttyput/1`, `ttyskip/1`, `ttytab/1`, `ttyflush/0`, `ttydisplay/1`, `ttydisplayq/1`, `ttyskipeol/0`, `ttydisplay_string/1`.

51.2 Documentation on exports (ttyout)

ttyget/1: PREDICATE
No further documentation available for this predicate.

ttyget1/1: PREDICATE
No further documentation available for this predicate.

ttynl/0: PREDICATE
No further documentation available for this predicate.

ttyput/1: PREDICATE
No further documentation available for this predicate.

ttyskip/1: PREDICATE
No further documentation available for this predicate.

ttytab/1: PREDICATE
No further documentation available for this predicate.

ttyflush/0: PREDICATE
No further documentation available for this predicate.

ttydisplay/1: PREDICATE

No further documentation available for this predicate.

ttydisplayq/1: PREDICATE

No further documentation available for this predicate.

ttyskipeol/0: PREDICATE

No further documentation available for this predicate.

ttydisplay_string/1: PREDICATE

No further documentation available for this predicate.

52 dynmods (library)

52.1 Usage and interface (dynmods)

- **Library usage:**
  `:- use_module(library(dynmods)).`
- **Exports:**
  - *Predicates:*
    `use_module/1`.
- **Other modules used:**
  - *System library modules:*
    `aggregates`, `dynamic`, `iso_misc`, `iso_byte_char`, `iso_incomplete`, `operators`, `read`, `streams`, `write`, `compiler/compiler`.

52.2 Documentation on exports (dynmods)

use_module/1: PREDICATE

No further documentation available for this predicate.

Meta-predicate with arguments: `use_module(addmodule)`.

53 runtime_ops (library)

53.1 Usage and interface (runtime_ops)

- **Library usage:**

  `:- use_package(runtime_ops).`

  or

  `:- module(...,...,[runtime_ops]).`
- **Other modules used:**
  - *System library modules:*

    `operators`, `aggregates`, `dynamic`, `iso_misc`, `iso_byte_char`, `iso_incomplete`, `read`, `streams`, `write`.

PART V - Annotated Prolog library (assertions)

Ciao allows *annotating* the program code with *assertions*. Such assertions include type and instantiation mode declarations, but also more general properties as well as comments in the style of the *literate programming*. These assertions document predicates (and modules and whole applications) and can be used by the Ciao preprocessor/compiler while debugging and optimizing the program or library, and by the Ciao documenter to build the program or library reference manual.

54 The Ciao assertion package

Author(s): Manuel Hermenegildo, Francisco Bueno, German Puebla.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.5#8 (1999/12/9, 21:1:11 MET)

The `assertions` package adds a number of new declaration definitions and new operator definitions which allow including program assertions in user programs. Such assertions can be used to describe predicates, properties, modules, applications, etc. These descriptions can be formal specifications (such as preconditions and post-conditions) or machine-readable textual comments.

This module is part of the `assertions` library. It defines the basic code-related assertions, i.e., those intended to be used mainly by compilation-related tools, such as the static analyzer or the run-time test generator.

Giving specifications for predicates and other program elements is the main functionality documented here. The exact syntax of comments is described in the autodocumenter (`lpdoc` [Knu84,Her99a]) manual, although some support for adding machine-readable comments in assertions is also mentioned here.

There are two kinds of assertions: predicate assertions and program point assertions. All predicate assertions are currently placed as directives in the source code, i.e., preceded by "`:-`". Program point assertions are placed as goals in clause bodies.

54.1 More info

The facilities provided by the library are documented in the description of its component modules. This documentation is intended to provide information only at a "reference manual" level. For a more tutorial introduction to the subject and some more examples please see the document "An Assertion Language for Debugging of Constraint Logic Programs (Technical Report CLIP2/97.1)". The assertion language implemented in this library is modeled after this design document, although, due to implementation issues, it may differ in some details. The purpose of this manual is to document precisely what the implementation of the library supports at any given point in time.

54.2 Some attention points

- **Formatting commands within text strings:** many of the predicates defined in these modules include arguments intended for providing textual information. This includes titles, descriptions, comments, etc. The type of this argument is a character string. In order for the automatic generation of documentation to work correctly, this character string should adhere to certain conventions. See the description of the `docstring/1` type/grammar for details.
- **Referring to variables:** In order for the automatic documentation system to work correctly, variable names (for example, when referring to arguments in the head patterns of *pred* declarations) must be surrounded by an `@var` command. For example, `@var{VariableName}` should be used for referring to the variable "VariableName", which will appear then formatted as follows: `VariableName`. See the description of the `docstring/1` type/grammar for details.

54.3 Usage and interface (assertions)

- **Library usage:**
  The recommended procedure in order to make use of assertions in user programs is to include the `assertions` syntax library, using one of the following declarations, as appropriate:
  ```
  :- module(...,...,[assertions]).
  :- include(library(assertions)).
  :- use_package([assertions]).
  ```
- **Exports:**
  - *Predicates:*
    `check/1`, `trust/1`, `true/1`, `false/1`.
- **New operators defined:**
  `=>/2` [975,xfx], `::/2` [978,xfx], `decl/1` [1150,fx], `decl/2` [1150,xfx], `pred/1` [1150,fx], `pred/2` [1150,xfx], `prop/1` [1150,fx], `prop/2` [1150,xfx], `modedef/1` [1150,fx], `calls/1` [1150,fx], `calls/2` [1150,xfx], `success/1` [1150,fx], `success/2` [1150,xfx], `comp/1` [1150,fx], `comp/2` [1150,xfx], `entry/1` [1150,fx].
- **New declarations defined:**
  `pred/1`, `pred/2`, `calls/1`, `calls/2`, `success/1`, `success/2`, `comp/1`, `comp/2`, `prop/1`, `prop/2`, `entry/1`, `modedef/1`, `decl/1`, `decl/2`, `comment/2`.
- **Other modules used:**
  - *System library modules:*
    `assertions/assertions_props`.

54.4 Documentation on new declarations (assertions)

pred/1: DECLARATION

This assertion provides information on a predicate. The body of the assertion (its only argument) contains properties or comments in the formats defined by `assrt_body/1`.

More than one of these assertions may appear per predicate, in which case each one represents a possible " mode" of use (usage) of the predicate. The exact scope of the usage is defined by the properties given for calls in the body of each assertion (which should thus distinguish the different usages intended). All of them together cover all possible modes of usage.

For example, the following assertions describe (all the and the only) modes of usage of predicate `length/2` (see `lists`):

```
:- pred length(L,N) : list * var => list * integer
# "Computes the length of L.".
:- pred length(L,N) : var * integer => list * integer
# "Outputs L of length N.".
:- pred length(L,N) : list * integer => list * integer
# "Checks that L is of length N.".
```

Usage: :- `pred(AssertionBody)`.

- *The following properties should hold at call time:*
  `AssertionBody` is an assertion body. (`assertions_props:assrt_body/1`)

pred/2: DECLARATION

This assertion is similar to a `pred/1` assertion but it is explicitely qualified. Non-qualified `pred/1` assertions are assumed the qualifier `check`.

Usage: :- `pred(AssertionStatus,AssertionBody)`.

– *The following properties should hold at call time:*

`AssertionStatus` is an acceptable status for an assertion. (`assertions_props:assrt_status/1`)

`AssertionBody` is an assertion body. (`assertions_props:assrt_body/1`)

calls/1: DECLARATION

This assertion is similar to a `pred/1` assertion but it only provides information about the calls to a predicate. If one or several calls assertions are given they are understood to describe all possible calls to the predicate.

For example, the following assertion describes all possible calls to predicate `is/2` (see `arithmetic`):

```
:- calls is(term,arithexpression).
```

Usage: :- `calls(AssertionBody)`.

– *The following properties should hold at call time:*

`AssertionBody` is a call assertion body. (`assertions_props:c_assrt_body/1`)

calls/2: DECLARATION

This assertion is similar to a `calls/1` assertion but it is explicitely qualified. Non-qualified `calls/1` assertions are assumed the qualifier `check`.

Usage: :- `calls(AssertionStatus,AssertionBody)`.

– *The following properties should hold at call time:*

`AssertionStatus` is an acceptable status for an assertion. (`assertions_props:assrt_status/1`)

`AssertionBody` is a call assertion body. (`assertions_props:c_assrt_body/1`)

success/1: DECLARATION

This assertion is similar to a `pred/1` assertion but it only provides information about the answers to a predicate. The described answers might be conditioned to a particular way of calling the predicate.

For example, the following assertion specifies the answers of the `length/2` predicate *if* it is called as in the first mode of usage above (note that the previous pred assertion already conveys such information, however it also compelled the predicate calls, while the success assertion does not):

```
:- success length(L,N) : list * var => list * integer.
```

Usage: :- `success(AssertionBody)`.

– *The following properties should hold at call time:*

`AssertionBody` is a predicate assertion body. (`assertions_props:s_assrt_body/1`)

success/2: DECLARATION

This assertion is similar to a `success/1` assertion but it is explicitely qualified. Non-qualified `success/1` assertions are assumed the qualifier `check`.

Usage: :- `success(AssertionStatus,AssertionBody)`.

– *The following properties should hold at call time:*

`AssertionStatus` is an acceptable status for an assertion. (`assertions_props:assrt_status/1`)

`AssertionBody` is a predicate assertion body. (`assertions_props:s_assrt_body/1`)

comp/1: DECLARATION

This assertion is similar to a `pred/1` assertion but it only provides information about the global execution properties of a predicate (note that such kind of information is also conveyed by pred assertions). The described properties might be conditioned to a particular way of calling the predicate.

For example, the following assertion specifies that the computation of `append/3` (see `lists`) will not fail *if* it is called as described (but does not compel the predicate to be called that way):

```
:- comp append(Xs,Ys,Zs) : var * var * var + not_fail.
```

Usage: :- `comp(AssertionBody)`.

– *The following properties should hold at call time:*

`AssertionBody` is a comp assertion body. (`assertions_props:g_assrt_body/1`)

comp/2: DECLARATION

This assertion is similar to a `comp/1` assertion but it is explicitely qualified. Non-qualified `comp/1` assertions are assumed the qualifier `check`.

Usage: :- `comp(AssertionStatus,AssertionBody)`.

– *The following properties should hold at call time:*

`AssertionStatus` is an acceptable status for an assertion. (`assertions_props:assrt_status/1`)

`AssertionBody` is a comp assertion body. (`assertions_props:g_assrt_body/1`)

prop/1: DECLARATION

This assertion is similar to a `pred/1` assertion but it flags that the predicate being documented is also a " property."

Properties are standard predicates, but which are *guaranteed to terminate for any possible instantiation state of their argument(s)*, do not perform side-effects which may interfere with the program behaviour, and do not further instantiate their arguments or add new constraints.

Provided the above holds, properties can thus be safely used as run-time checks. The program transformation used in `ciaopp` for run-time checking guarantees the third requirement. It also performs some basic checks on properties which in most cases are enough for the second requirement. However, it is the user's responsibility to guarantee termination of the properties defined. (See also Chapter 56 [Declaring regular types], page 241 for some considerations applicable to writing properties.)

The set of properties is thus a strict subset of the set of predicates. Note that properties can be used to describe characteristics of arguments in assertions and they can also be executed (called) as any other predicates.

Usage: :- `prop(AssertionBody)`.

- *The following properties should hold at call time:*
  `AssertionBody` is an assertion body. (`assertions_props:assrt_body/1`)

prop/2: DECLARATION

This assertion is similar to a `prop/1` assertion but it is explicitely qualified. Non-qualified `prop/1` assertions are assumed the qualifier `check`.

Usage: :- `prop(AssertionStatus,AssertionBody)`.

- *The following properties should hold at call time:*
  `AssertionStatus` is an acceptable status for an assertion. (`assertions_props:assrt_status/1`)
  `AssertionBody` is an assertion body. (`assertions_props:assrt_body/1`)

entry/1: DECLARATION

This assertion provides information about the *external* calls to a predicate. It is identical syntactically to a `calls/1` assertion. However, they describe only external calls, i.e., calls to the exported predicates of a module from outside the module, or calls to the predicates in a non-modular file from other files (or the user).

These assertions are *trusted* by the compiler. As a result, if their descriptions are erroneous they can introduce bugs in programs. Thus, `entry/1` assertions should be written with care.

An important use of these assertions is in providing information to the compiler which it may not be able to infer from the program. The main use is in providing information on the ways in which exported predicates of a module will be called from outside the module. This will greatly improve the precision of the analyzer, which otherwise has to assume that the arguments that exported predicates receive are any arbitrary term.

Usage: :- `entry(AssertionBody)`.

- *The following properties should hold at call time:*
  `AssertionBody` is a call assertion body. (`assertions_props:c_assrt_body/1`)

modedef/1: DECLARATION

This assertion is used to define modes. A mode defines in a compact way a set of call and success properties. Once defined, modes can be applied to predicate arguments in assertions. The meaning of this application is that the call and success properties defined by the mode hold for the argument to which the mode is applied. Thus, a mode is conceptually a "property macro".

The syntax of mode definitions is similar to that of pred declarations. For example, the following set of assertions:

```
:- modedef +A : nonvar(A) # "A is bound upon predicate entry.".

:- pred p(+A,B) : integer(A) =>  ground(B).
```

is equivalent to:

```
:- pred p(A,B) : (nonvar(A),integer(A)) =>  ground(B)
   # "A is bound upon predicate entry.".
```

Usage: :- `modedef(AssertionBody)`.

- *The following properties should hold at call time:*
  `AssertionBody` is an assertion body. (`assertions_props:assrt_body/1`)

decl/1: DECLARATION

This assertion is similar to a `pred/1` assertion but it is used for declarations instead than for predicates.

Usage: :- `decl(AssertionBody)`.

- *The following properties should hold at call time:*
  `AssertionBody` is an assertion body. (`assertions_props:assrt_body/1`)

decl/2: DECLARATION

This assertion is similar to a `decl/1` assertion but it is explicitely qualified. Non-qualified `decl/1` assertions are assumed the qualifier `check`.

Usage: :- `decl(AssertionStatus,AssertionBody)`.

- *The following properties should hold at call time:*
  `AssertionStatus` is an acceptable status for an assertion. (`assertions_props:assrt_status/1`)
  `AssertionBody` is an assertion body. (`assertions_props:assrt_body/1`)

comment/2: DECLARATION

Usage: :- `comment(Pred,Comment)`.

- *Description:* This assertion gives a text `Comment` for a given predicate `Pred`.
- *The following properties should hold at call time:*
  `Pred` is a head pattern. (`assertions_props:head_pattern/1`)
  `Comment` is a text comment with admissible documentation commands. The usual formatting commands that are applicable in comment strings are defined by `stringcommand/1`. See the `lpdoc` manual for documentation on comments. (`assertions_props:docstring/1`)

54.5 Documentation on exports (assertions)

check/1: PREDICATE

Usage: `check(PropertyConjunction)`

- *Description:* This assertion provides information on a clause program point (position in the body of a clause). Calls to a `check/1` assertion can appear in the body of a clause in any place where a literal can normally appear. The property defined by `PropertyConjunction` should hold in all the run-time stores corresponding to that program point. See also Chapter 61 [Run-time checking of assertions], page 259.

– *The following properties should hold at call time:*

  `PropertyConjunction` is either a term or a *conjunction* of terms. The main functor and arity of each of those terms corresponds to the definition of a property. The first argument of each such term is a variable which appears as a head argument. (`assertions_props:property_conjunction/1`)

trust/1: PREDICATE

Usage: `trust(PropertyConjunction)`

– *Description:* This assertion also provides information on a clause program point. It is identical syntactically to a `check/1` assertion. However, the properties stated are not taken as something to be checked but are instead *trusted* by the compiler. While the compiler may in some cases detect an inconsistency between a `trust/1` assertion and the program, in all other cases the information given in the assertion will be taken to be true. As a result, if these assertions are erroneous they can introduce bugs in programs. Thus, `trust/1` assertions should be written with care.

  An important use of these assertions is in providing information to the compiler which it may not be able to infer from the program (either because the information is not present or because the analyzer being used is not precise enough). In particular, providing information on external predicates which may not be accessible at the time of compiling the module can greatly improve the precision of the analyzer. This can be easily done with trust assertion.

– *The following properties should hold at call time:*

  `PropertyConjunction` is either a term or a *conjunction* of terms. The main functor and arity of each of those terms corresponds to the definition of a property. The first argument of each such term is a variable which appears as a head argument. (`assertions_props:property_conjunction/1`)

true/1: PREDICATE

Usage: `true(PropertyConjunction)`

– *Description:* This assertion is identical syntactically to a `check/1` assertion. However, the properties stated have been proved to hold by the analyzer. Thus, these assertions often represent the analyzer output.

– *The following properties should hold at call time:*

  `PropertyConjunction` is either a term or a *conjunction* of terms. The main functor and arity of each of those terms corresponds to the definition of a property. The first argument of each such term is a variable which appears as a head argument. (`assertions_props:property_conjunction/1`)

false/1: PREDICATE

Usage: `false(PropertyConjunction)`

– *Description:* This assertion is identical syntactically to a `check/1` assertion. However, the properties stated have been proved not to hold by the analyzer. Thus, these assertions often represent the analyzer output.

– *The following properties should hold at call time:*

  `PropertyConjunction` is either a term or a *conjunction* of terms. The main functor and arity of each of those terms corresponds to the definition of a property. The first argument of each such term is a variable which appears as a head argument. (`assertions_props:property_conjunction/1`)

55 Types and properties related to assertions

Author(s): Manuel Hermenegildo.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.5#25 (1999/12/29, 12:46:8 CET)

This module is part of the `assertions` library. It defines types and properties related to assertions.

55.1 Usage and interface (`assertions_props`)

- **Library usage:**
  `:- use_module(library(assertions_props)).`
- **Exports:**
  - *Properties:*
    `head_pattern/1`, `nabody/1`, `docstring/1`.
  - *Regular Types:*
    `assrt_body/1`, `complex_arg_property/1`, `property_conjunction/1`, `property_starterm/1`, `complex_goal_property/1`, `dictionary/1`, `c_assrt_body/1`, `s_assrt_body/1`, `g_assrt_body/1`, `assrt_status/1`, `assrt_type/1`, `predfunctor/1`, `propfunctor/1`.
- **Other modules used:**
  - *System library modules:*
    `dcg_expansion`.

55.2 Documentation on exports (`assertions_props`)

assrt_body/1: REGTYPE

This predicate defines the different types of syntax admissible in the bodies of `pred/1`, `func/1`, etc. assertions. Such a body is of the form:

```
Pr [:: DP] [: CP] [=> AP] [+ GP] [# CO]
```

where (fields between [...] are optional):

- `Pr` is a head pattern (`head_pattern/1`) which describes the predicate or property and possibly gives some implicit call/answer information.
- `DP` is a (possibly empty) complex argument property (`complex_arg_property/1`) which expresses properties which are compatible with the predicate, i.e., instantiations made by the predicate are *compatible* with the properties in the sense that applying the property at any point to would not make it fail.
- `CP` is a (possibly empty) complex argument property (`complex_arg_property/1`) which applies to the *calls* to the predicate.
- `AP` is a (possibly empty) complex argument property (`complex_arg_property/1`) which applies to the *answers* to the predicate (if the predicate succeeds). These only apply if the (possibly empty) properties given for calls in the assertion hold.

- `GP` is a (possibly empty) complex goal property (`complex_goal_property/1`) which applies to the *whole execution* of a call to the predicate. These only apply if the (possibly empty) properties given for calls in the assertion hold.
- `CO` is a comment string (`docstring/1`). This comment only applies if the (possibly empty) properties given for calls in the assertion hold. The usual formatting commands that are applicable in comment strings can be used (see `stringcommand/1`).

  See the `lpdoc` manual for documentation on assertion comments.

Usage: `assrt_body(X)`

- *Description:* `X` is an assertion body.

head_pattern/1: PROPERTY

A head pattern can be a predicate name (functor/arity) (`predname/1`) or a term. Thus, both `p/3` and `p(A,B,C)` are valid head patterns. In the case in which the head pattern is a term, each argument of such a term can be:

- A variable. This is useful in order to be able to refer to the corresponding argument positions by name within properties and in comments. Thus, `p(Input,Parameter,Output)` is a valid head pattern.
- A ground term. In this case this term determines a property of the corresponding argument. The actual property referred to is that given by the term but with one more argument added at the beginning, which is a new variable which, in a rewriting of the head pattern, appears at the argument position occupied by the term. Unless otherwise stated (see below), the property built this way is understood to hold for both calls and answers. For example, the head pattern `p(Input,list(integer),Output)` is valid and equivalent for example to having the head pattern `p(Input,A,Output)` and stating that the property `list(A,integer)` holds for the calls and successes of the predicate.
- Finally, it can also be a variable or a ground term, as above, but preceded by a " mode." This mode determines in a compact way certain call or answer properties. For example, the head pattern `p(Input,+list(integer),Output)` is valid, as long as `+/1` is declared as a mode.

  Acceptable modes are documented in `library(modes)`. User defined modes are documented in `modedef/1`.

Usage: `head_pattern(Pr)`

- *Description:* `Pr` is a head pattern.

complex_arg_property/1: REGTYPE

`complex_arg_property(Props)`

`Props` is a (possibly empty) complex argument property. Such properties can appear in two formats, which are defined by `property_conjunction/1` and `property_starterm/1` respectively. The two formats can be mixed provided they are not in the same field of an assertion. I.e., the following is a valid assertion:

`:- pred foo(X,Y) : nonvar * var => (ground(X),ground(Y)).`

Usage: `complex_arg_property(Props)`

- *Description:* `Props` is a (possibly empty) complex argument property

property_conjunction/1: REGTYPE

This type defines the first, unabridged format in which properties can be expressed in the bodies of assertions. It is essentially a conjunction of properties which refer to variables. The following is an example of a complex property in this format:

- `(integer(X),list(Y,integer))`: `X` has the property `integer/1` and `Y` has the property `list/2`, with second argument `integer`.

Usage: `property_conjunction(Props)`

- *Description:* `Props` is either a term or a *conjunction* of terms. The main functor and arity of each of those terms corresponds to the definition of a property. The first argument of each such term is a variable which appears as a head argument.

property_starterm/1: REGTYPE

This type defines a second, compact format in which properties can be expressed in the bodies of assertions. A `property_starterm/1` is a term whose main functor is `*/2` and, when it appears in an assertion, the number of terms joined by `*/2` is exactly the arity of the predicate it refers to. A similar series of properties as in `property_conjunction/1` appears, but the arity of each property is one less: the argument position to which they refer (first argument) is left out and determined by the position of the property in the `property_starterm/1`. The idea is that each element of the `*/2` term corresponds to a head argument position. Several properties can be assigned to each argument position by grouping them in curly brackets. The following is an example of a complex property in this format:

- `integer * list(integer)`: the first argument of the procedure (or function, or ...) has the property `integer/1` and the second one has the property `list/2`, with second argument `integer`.
- `{integer,var} * list(integer)`: the first argument of the procedure (or function, or ...) has the properties `integer/1` and `var/1` and the second one has the property `list/2`, with second argument `integer`.

Usage: `property_starterm(Props)`

- *Description:* `Props` is either a term or several terms separated by `*/2`. The main functor of each of those terms corresponds to that of the definition of a property, and the arity should be one less than in the definition of such property. All arguments of each such term are ground.

complex_goal_property/1: REGTYPE

`complex_goal_property(Props)`

`Props` is a (possibly empty) complex goal property. Such properties can be either a term or a *conjunction* of terms. The main functor and arity of each of those terms corresponds to the definition of a property. Such properties apply to all executions of all goals of the predicate which comply with the assertion in which the `Props` appear.

The arguments of the terms in `Props` are implicitely augmented with a first argument which corresponds to a goal of the predicate of the assertion in which the `Props` appear. For example, the assertion

```
:- comp var(A) + not_further_inst(A).
```

has property `not_further_inst/1` as goal property, and establishes that in all executions of `var(A)` it should hold that `not_further_inst(var(A),A)`.

Usage: `complex_goal_property(Props)`

– *Description:* `Props` is either a term or a *conjunction* of terms. The main functor and arity of each of those terms corresponds to the definition of a property. A first implicit argument in such terms identifies goals to which the properties apply.

nabody/1: PROPERTY

Usage: `nabody(ABody)`

– *Description:* `ABody` is a normalized assertion body.

dictionary/1: REGTYPE

Usage: `dictionary(D)`

– *Description:* `D` is a dictionary of variable names.

c_assrt_body/1: REGTYPE

This predicate defines the different types of syntax admissible in the bodies of `call/1`, `entry/1`, etc. assertions. The following are admissible:

```
Pr : CP [# CO]
```

where (fields between [...] are optional):

- `CP` is a (possibly empty) complex argument property (`complex_arg_property/1`) which applies to the *calls* to the predicate.
- `CO` is a comment string (`docstring/1`). This comment only applies if the (possibly empty) properties given for calls in the assertion hold. The usual formatting commands that are applicable in comment strings can be used (see `stringcommand/1`).

The format of the different parts of the assertion body are given by `n_assrt_body/5` and its auxiliary types.

Usage: `c_assrt_body(X)`

– *Description:* `X` is a call assertion body.

s_assrt_body/1: REGTYPE

This predicate defines the different types of syntax admissible in the bodies of `pred/1`, `func/1`, etc. assertions. The following are admissible:

```
Pr : CP => AP # CO
Pr : CP => AP
Pr => AP # CO
Pr => AP
```

where:

- `Pr` is a head pattern (`head_pattern/1`) which describes the predicate or property and possibly gives some implicit call/answer information.
- `CP` is a (possibly empty) complex argument property (`complex_arg_property/1`) which applies to the *calls* to the predicate.
- `AP` is a (possibly empty) complex argument property (`complex_arg_property/1`) which applies to the *answers* to the predicate (if the predicate succeeds). These only apply if the (possibly empty) properties given for calls in the assertion hold.

- CO is a comment string (docstring/1). This comment only applies if the (possibly empty) properties given for calls in the assertion hold. The usual formatting commands that are applicable in comment strings can be used (see stringcommand/1).

The format of the different parts of the assertion body are given by n_assrt_body/5 and its auxiliary types.

Usage: s_assrt_body(X)

- *Description:* X is a predicate assertion body.

g_assrt_body/1: REGTYPE

This predicate defines the different types of syntax admissible in the bodies of comp/1 assertions. The following are admissible:

```
Pr : CP + GP # CO
Pr : CP + GP
Pr + GP # CO
Pr + GP
```

where:

- Pr is a head pattern (head_pattern/1) which describes the predicate or property and possibly gives some implicit call/answer information.
- CP is a (possibly empty) complex argument property (complex_arg_property/1) which applies to the *calls* to the predicate.
- GP contains (possibly empty) complex goal property (complex_goal_property/1) which applies to the *whole execution* of a call to the predicate. These only apply if the (possibly empty) properties given for calls in the assertion hold.
- CO is a comment string (docstring/1). This comment only applies if the (possibly empty) properties given for calls in the assertion hold. The usual formatting commands that are applicable in comment strings can be used (see stringcommand/1).

The format of the different parts of the assertion body are given by n_assrt_body/5 and its auxiliary types.

Usage: g_assrt_body(X)

- *Description:* X is a comp assertion body.

assrt_status/1: REGTYPE

The types of assertion status. They have the same meaning as the program-point assertions, and are as follows:

```
assrt_status(true).
assrt_status(false).
assrt_status(check).
assrt_status(checked).
assrt_status(trust).
```

Usage: assrt_status(X)

- *Description:* X is an acceptable status for an assertion.

assrt_type/1: REGTYPE

The admissible kinds of assertions:

```
assrt_type(pred).
assrt_type(prop).
assrt_type(decl).
assrt_type(func).
assrt_type(calls).
assrt_type(success).
assrt_type(comp).
assrt_type(entry).
assrt_type(modedef).
```

Usage: `assrt_type(X)`

- *Description:* `X` is an admissible kind of assertion.

predfunctor/1: REGTYPE

Usage: `predfunctor(X)`

- *Description:* `X` is a type of assertion which defines a predicate.

propfunctor/1: REGTYPE

Usage: `propfunctor(X)`

- *Description:* `X` is a type of assertion which defines a *property*.

docstring/1: PROPERTY

Usage: `docstring(String)`

- *Description:* `String` is a text comment with admissible documentation commands. The usual formatting commands that are applicable in comment strings are defined by `stringcommand/1`. See the `lpdoc` manual for documentation on comments.

56 Declaring regular types

Author(s): Manuel Hermenegildo, Pedro Lopez, Francisco Bueno.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.5#9 (1999/12/9, 21:57:42 MET)

This library package adds some new declaration definitions and new operator definitions to user programs. These new declarations and operators provide some very simple syntactic sugar to support regular type definitions in source code. Regular types are just properties which have the additional characteristic of being regular types (`basic_props:regtype/1`).

For example, this library package allows writing:

```
:- regtype tree(X) # "X is a tree.".
```

instead of the more combersome:

```
:- prop tree(X) + regtype # "X is a tree.".
```

Regular types can be used as properties to describe predicates and play an essential role in program debugging (see the Ciao Prolog preprocessor (`ciaopp`) manual).

In this chapter we explain some general considerations worth taking into account when writing properties in general, not just regular types. The exact syntax of regular types is also described.

56.1 Defining properties

Given the classes of assertions in the Ciao assertion language, there are two fundamental classes of properties. Properties used in assertions which refer to execution states (i.e., `calls/1`, `success/1`, and the like) are called *properties of execution states*. Properties used in assertions related to computations (i.e., `comp/1`) are called *properties of computations*. Different considerations apply when writing a property of the former or of the later kind.

Consider a definition of the predicate `string_concat/3` which concatenates two character strings (represented as lists of ASCII codes):

```
string_concat([],L,L).
string_concat([X|Xs],L,[X|NL]):- string_concat(Xs,L,NL).
```

Assume that we would like to state in an assertion that each argument "is a list of integers." However, we must decide which one of the following two possibilities we mean exactly: "the argument is *instantiated* to a list of integers" (let us call this property `instantiated_to_intlist/1`), or "if any part of the argument is instantiated, this instantiation must be compatible with the argument being a list of integers" (we will call this property `compatible_with_intlist/1`). For example, `instantiated_to_intlist/1` should be true for the terms `[]` and `[1,2]`, but should not for `X`, `[a,2]`, and `[X,2]`. In turn, `compatible_with_intlist/1` should be true for `[]`, `X`, `[1,2]`, and `[X,2]`, but should not be for `[X|1]`, `[a,2]`, and `1`. We refer to properties such as `instantiated_to_intlist/1` above as *instantiation properties* and to those such as `compatible_with_intlist/1` as *compatibility properties* (corresponding to the traditional notions of "instantiation types" and "compatibility types").

It turns out that both of these notions are quite useful in practice. In the example above, we probably would like to use `compatible_with_intlist/1` to state that on success of `string_concat/3` all three argument must be compatible with lists of integers in an assertion like:

```
:- success string_concat(A,B,C) => ( compatible_with_intlist(A),
                                     compatible_with_intlist(B),
                                     compatible_with_intlist(C) ).
```

With this assertion, no error will be flagged for a call to `string_concat/3` such as `string_concat([20],L,R)`, which on success produces the resulting atom `string_concat([20],L,[20|L])`, but a call `string_concat([],a,R)` would indeed flag an error.

On the other hand, and assuming that we are running on a Prolog system, we would probably like to use `instantiated_to_intlist/1` for `sumlist/2` as follows:

```
:- calls sumlist(L,N) : instantiated_to_intlist(L).

sumlist([],0).
sumlist([X|R],S) :- sumlist(R,PS), S is PS+X.
```

to describe the type of calls for which the program has been designed, i.e., those in which the first argument of `sumlist/2` is indeed a list of integers.

The property `instantiated_to_intlist/1` might be written as in the following (Prolog) definition:

```
:- prop instantiated_to_intlist/1.

instantiated_to_intlist(X) :-
       nonvar(X), instantiated_to_intlist_aux(X).

instantiated_to_intlist_aux([]).
instantiated_to_intlist_aux([X|T]) :-
       integer(X), instantiated_to_intlist(T).
```

(Recall that the Prolog builtin `integer/1` itself implements an instantiation check, failing if called with a variable as the argument.)

The property `compatible_with_intlist/1` might in turn be written as follows (also in Prolog):

```
:- prop compatible_with_intlist/1.

compatible_with_intlist(X) :- var(X).
compatible_with_intlist(X) :-
       nonvar(X), compatible_with_intlist_aux(X).

compatible_with_intlist_aux([]).
compatible_with_intlist_aux([X|T]) :-
       int_compat(X), compatible_with_intlist(T).

int_compat(X) :- var(X).
int_compat(X) :- nonvar(X), integer(X).
```

Note that these predicates meet the criteria for being properties and thus the `prop/1` declaration is correct.

Ensuring that a property meets the criteria for "not affecting the computation" can sometimes make its coding somewhat tedious. In some ways, one would like to be able to write simply:

```
intlist([]).
intlist([X|R]) :- int(X), intlist(R).
```

(Incidentally, note that the above definition, provided that it suits the requirements for being a property and that `int/1` is a regular type, meets the criteria for being a regular type. Thus, it could be declared `:- regtype intlist/1`.)

But note that (independently of the definition of `int/1`) the definition above is not the correct instantiation check, since it would succeed for a call such as `intlist(X)`. In fact, it is not strictly correct as a compatibility property either, because, while it would fail or succeed as expected, it would perform instantiations (e.g., if called with `intlist(X)` it would bind `X` to `[]`). In practice, it is convenient to provide some run-time support to aid in this task.

The run-time support of the Ciao system (see Chapter 61 [Run-time checking of assertions], page 259) ensures that the execution of properties is performed in such a way that properties written as above can be used directly as instantiation checks. Thus, writing:

```
:- calls sumlist(L,N) : intlist(L).
```

has the desired effect. Also, the same properties can often be used as compatibility checks by writing them in the assertions as `compat(Property)` (`basic_props:compat/1`). Thus, writing:

```
:- success string_concat(A,B,C) => ( compat(intlist(A)),
                                      compat(intlist(B)),
                                      compat(intlist(C)) ).
```

also has the desired effect.

As a general rule, the properties that can be used directly for checking for compatibility should be *downwards closed*, i.e., once they hold they will keep on holding in every state accessible in forwards execution. There are certain predicates which are inherently *instantiation* checks and should not be used as *compatibility* properties nor appear in the definition of a property that is to be used with `compat`. Examples of such predicates (for Prolog) are `==`, `ground`, `nonvar`, `integer`, `atom`, `>`, etc. as they require a certain instantiation degree of their arguments in order to succeed.

In contrast with properties of execution states, *properties of computations* refer to the entire execution of the call(s) that the assertion relates to. One such property is, for example, `not_fail/1` (note that although it has been used as in `:- comp append(Xs,Ys,Zs) + not_fail`, it is in fact read as `not_fail(append(Xs,Ys,Zs))`; see `assertions_props:complex_goal_property/1`). For this property, which should be interpreted as "execution of the predicate either succeeds at least once or loops," we can use the following predicate `not_fail/1` for run-time checking:

```
not_fail(Goal):-
      if( call(Goal),
          true,             %% then
          warning(Goal) ). %% else
```

where the `warning/1` (library) predicate simply prints a warning message.

In this simple case, implementation of the predicate is not very difficult using the (non-standard) `if/3` builtin predicate present in many Prolog systems.

However, it is not so easy to code predicates which check other properties of the computation and we may in general need to program a meta-interpreter for this purpose.

56.2 Usage and interface (regtypes)

- **Library usage:**
  `:- use_package(regtypes).`
  or
  `:- module(...,...,[regtypes]).`
- **New operators defined:**
  `regtype/1` [1150,fx], `regtype/2` [1150,xfx].
- **New declarations defined:**
  `regtype/1`, `regtype/2`.
- **Other modules used:**
  – *System library modules:*
    `assertions/assertions_props`.

56.3 Documentation on new declarations (regtypes)

regtype/1: DECLARATION

This assertion is similar to a pred assertion but it flags that the predicate being documented is also a " regular type." This allows for example checking whether it is in the class of types supported by the type checking and inference modules. Currently, types are properties whose definitions are *regular programs.*

A regular program is defined by a set of clauses, each of the form:

```
p(x, v_1, ..., v_n)  :- body_1, ..., body_k.
```

where:

1. `x` is a term whose variables (which are called *term variables*) are unique, i.e., it is not allowed to introduce equality constraints between the variables of `x`.
   For example, `p(f(X, Y)) :- ...` is valid, but `p(f(X, X)) :- ...` is not.
2. in all clauses defining `p/n+1` the terms `x` do not unify except maybe for one single clause in which `x` is a variable.
3. `n` >= 0 and `p/n` is a *parametric type functor* (whereas the predicate defined by the clauses is `p/n+1`).
4. `v_1`, ..., `v_n` are unique variables, which are called *parametric variables.*
5. Each `body_i` is of the form:
   1. `regtype(z, t)` where `z` is one of the *term variables* and `t` is a *regular type expression*;
   2. `q(y, t_1, ..., t_m)` where `m` >= 0, `q/m` is a *parametric type functor*, not in the set of functors `=/2`, `^/2`, `./3` `regtype/2`.
      `t_1, ..., t_m` are *regular type expressions*, and `y` is a *term variable.*
6. Each term variable occurs at most once in the clause's body (and should be as the first argument of a literal).

A *regular type expression* is either a parametric variable or a parametric type functor applied to some of the parametric variables.

A parametric type functor is a regular type, defined by a regular program, or a basic type. Basic types are defined in the CIAO `engine(basic_props)` manual.

The set of types is thus a well defined subset of the set of properties. Note that types can be used to describe characteristics of arguments in assertions and they can also be executed (called) as any other predicates.

Usage: :- `regtype(AssertionBody)`.

– *The following properties should hold at call time:*

`AssertionBody` is an assertion body. (`assertions_props:assrt_body/1`)

regtype/2: DECLARATION

This assertion is similar to a `regtype/1` assertion but it is explicitely qualified. Non-qualified `regtype/1` assertions are assumed the qualifier `check`. Note that checking regular type definitions should be done with the `ciaopp` preprocessor.

Usage: :- `regtype(AssertionStatus,AssertionBody)`.

– *The following properties should hold at call time:*

`AssertionStatus` is an acceptable status for an assertion. (`assertions_props:assrt_status/1`)

`AssertionBody` is an assertion body. (`assertions_props:assrt_body/1`)

57 Properties which are native to analyzers

Author(s): Francisco Bueno, Manuel Hermenegildo, Pedro Lopez.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.5#1 (1999/11/29, 17:12:34 MET)

This library contains a set of properties which are natively understood by the different program analyzers of `ciaopp`. They are used by `ciaopp` on output and they can also be used as properties in assertions.

57.1 Usage and interface (`native_props`)

- **Library usage:**
  `:- use_module(library('assertions/native_props'))`
  or also as a package `:- use_package(nativeprops)`.
  Note the different names of the library and the package.
- **Exports:**
  - *Properties:*
    `linear/1`, `mshare/1`, `fails/1`, `not_fails/1`, `possibly_fails/1`, `covered/1`, `not_covered/1`, `is_det/1`, `possibly_nondet/1`, `mut_exclusive/1`, `not_mut_exclusive/1`, `size_lb/2`, `size_ub/2`, `steps_lb/2`, `steps_ub/2`, `sideff_pure/1`, `sideff_soft/1`, `sideff_hard/1`.
- **Other modules used:**
  - *System library modules:*
    `andprolog/andprolog`, `metaterms`, `sort`, `lists`.

57.2 Documentation on exports (`native_props`)

linear/1: PROPERTY

`linear(X)`

`X` is bound to a term which is linear, i.e., if it contains any variables, such variables appear only once in the term. For example, `[1,2,3]` and `f(A,B)` are linear terms, while `f(A,A)` is not.

Usage: `linear(X)`

- *Description:* `X` is instantiated to a linear term.

mshare/1: PROPERTY

`mshare(X)`

`X` contains all *sharing sets* [JL88,MH89] which specify the possible variable occurrences in the terms to which the variables involved in the clause may be bound. Sharing sets are a compact way of representing groundness of variables and dependencies between variables. This representation is however generally difficult to read for humans. For this reason, this information is often translated to `ground/1`, `indep/1` and `indep/2` properties, which are easier to read.

Usage: `mshare(X)`

– *Description:* The sharing pattern is `X`.

fails/1: PROPERTY

`fails(X)`

Calls of the form `X` fail.

Usage: `fails(X)`

– *Description:* Calls of the form `X` fail.

not_fails/1: PROPERTY

`not_fails(X)`

Calls of the form `X` produce at least one solution, or not terminate [DLGH97].

Usage: `not_fails(X)`

– *Description:* All the calls of the form `X` do not fail.

possibly_fails/1: PROPERTY

`possibly_fails(X)`

Non-failure is not ensured for any call of the form `X` [DLGH97]. In other words, nothing can be ensured about non-failure nor termination of such calls.

Usage: `possibly_fails(X)`

– *Description:* Non-failure is not ensured for calls of the form `X`.

covered/1: PROPERTY

`covered(X)`

For any call of the form `X` there is at least one clause whose test succeeds (i.e. all the calls of the form `X` are covered.) [DLGH97].

Usage: `covered(X)`

– *Description:* All the calls of the form `X` are covered.

not_covered/1: PROPERTY

`not_covered(X)`

There is some call of the form `X` for which there is not any clause whose test succeeds [DLGH97].

Usage: `not_covered(X)`

– *Description:* Not all of the calls of the form `X` are covered.

is_det/1: PROPERTY

`is_det(X)`

All calls of the form `X` are deterministic, i.e. produce at most one solution, or not terminate.

Usage: `is_det(X)`

– *Description:* All calls of the form `X` are deterministic.

possibly_nondet/1: PROPERTY

`possibly_nondet(X)`

Non-determinism is not ensured for all calls of the form `X`. In other words, nothing can be ensured about determinacy nor termination of such calls.

Usage: `possibly_nondet(X)`

- *Description:* Non-determinism is not ensured for calls of the form `X`.

mut_exclusive/1: PROPERTY

`mut_exclusive(X)`

For any call of the form `X` at most one clause succeeds, i.e. clauses are pairwise exclusive.

Usage: `mut_exclusive(X)`

- *Description:* For any call of the form `X` at most one clause succeeds.

not_mut_exclusive/1: PROPERTY

`not_mut_exclusive(X)`

Not for all calls of the form `X` at most one clause succeeds. I.e. clauses are not disjoint for some call.

Usage: `not_mut_exclusive(X)`

- *Description:* Not for all calls of the form `X` at most one clause succeeds.

size_lb/2: PROPERTY

`size_lb(X,Y)`

The minimum size of the terms to which the argument `Y` is bound to is given by the expression `Y`. Various measures can be used to determine the size of an argument, e.g., list-length, term-size, term-depth, integer-value, etc. [DL93].

Usage: `size_lb(X,Y)`

- *Description:* `Y` is a lower bound on the size of argument `X`.

size_ub/2: PROPERTY

`size_ub(X,Y)`

The maximum size of the terms to which the argument `Y` is bound to is given by the expression `Y`. Various measures can be used to determine the size of an argument, e.g., list-length, term-size, term-depth, integer-value, etc. [DL93].

Usage: `size_ub(X,Y)`

- *Description:* `Y` is a upper bound on the size of argument `X`.

steps_lb/2: PROPERTY

`steps_lb(X,Y)`

The minimum computation time (in resolution steps) spent by any call of the form `X` is given by the expression `Y` [DLGHL97,LGHD96]

Usage: `steps_lb(X,Y)`

- *Description:* `Y` is a lower bound on the cost of any call of the form `X`.

steps_ub/2: PROPERTY

`steps_ub(X,Y)`

The maximum computation time (in resolution steps) spent by any call of the form `X` is given by the expression `Y` [DL93,LGHD96]

Usage: `steps_ub(X,Y)`

- *Description:* `Y` is a upper bound on the cost of any call of the form `X`.

sideff_pure/1: PROPERTY

Usage: `sideff_pure(X)`

- *Description:* `X` is pure, i.e., has no side-effects.

sideff_soft/1: PROPERTY

Usage: `sideff_soft(X)`

- *Description:* `X` has *soft side-effects*, i.e., those not affecting program execution (e.g., input/output).

sideff_hard/1: PROPERTY

Usage: `sideff_hard(X)`

- *Description:* `X` has *hard side-effects*, i.e., those that might affect program execution (e.g., assert/retract).

indep/1: PROPERTY

Usage: `indep(X)`

- *Description:* The variables in pairs in `X` are pairwise independent.

indep/2: PROPERTY

Usage: `indep(X,Y)`

- *Description:* `X` and `Y` do not have variables in common.

58 Meta-properties

Author(s): Francisco Bueno.

This library allows the use of some meta-constructs which provide for specifying properties of terms which are unknown at the time of the specification, or expressed with a shorthand for the property definition, i.e., without really defining it.

An example of such use is an assertion which specifies that any property holding upon call will also hold upon exit:

```
:- pred p(X) : Prop(X) => Prop(X).
```

Another example is using shorthands for properties when documenting:

```
:- pred p(X) : regtype(X,(^(list;list);list)).
```

(See below for an explanation of such a regular type.)

58.1 Usage and interface (meta_props)

- **Library usage:**
  `:- use_module(library('assertions/meta_props'))`
  or also as a package `:- use_package(metaprops)`.
  Note the different names of the library and the package.
- **Exports:**
  - *Properties:*
    `call/2`, `prop/2`, `regtype/2`.
  - *Multifiles:*
    `callme/2`.

58.2 Documentation on exports (meta_props)

call/2: PROPERTY

`call(P,A)`

`A` has property `P` (provided that `P` is a property). Equivalent to `P(A)`.

Usage: `call(P,A)`

- *Description:* `A` has property `P`.
- *If the following properties hold at call time:*
  `P` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

prop/2: PROPERTY

Usage: `prop(A,P)`

- *Description:* `A` has property `P`.
- *If the following properties hold at call time:*
  `P` has property `^ (callable;prop_abs)`. (`meta_props:prop/2`)

regtype/2: PROPERTY

Usage: `regtype(A,T)`

- *Description:* `A` is of type `T`.
- *If the following properties hold at call time:*
  `T` has property `^ (regtype;prop_abs)`. (`meta_props:prop/2`)

58.3 Documentation on multifiles (`meta_props`)

callme/2: PREDICATE

(User defined.) A hook predicate you have to define as `callme(P,X):- P(X), !.` in the program that uses this library. This is done automatically if the package is used instead of the library module (but then you *should not* define `callme/2` in your program).

The predicate is *multifile*.

58.4 Documentation on internals (`meta_props`)

prop_abs/1: PROPERTY

`prop_abs(Prop)`

`Prop` is a *property abstraction*, i.e., a *parametric property*, or a term formed of property abstractions, where the functors used in the term are escaped by `^`.

One particular case of property abstractions are *parametric regular type abstractions*, i.e., a parametric type functor or a `^`-escaped term formed of regular type abstractions.

Such abstractions are a short-hand for a corresponding regular type (correspondingly, property). For example, the following abstraction:

```
^(list;list);list
```

denotes terms of the form `(X;Y)` where `list(X)` and `list(Y)` hold and also terms `T` such that `list(T)` holds. It is equivalent to the regular type:

```
abstract_type((X;Y)):- list(X), list(Y).
abstract_type(T):- list(T).
```

Usage: `prop_abs(Prop)`

- *Description:* `Prop` is a property abstraction.

58.5 Known bugs and planned improvements (`meta_props`)

- Using a hook predicate is not very elegant. Need something else.
- The cut in the hook prevents backtracking (enough for most uses of properties but not quite ok).

59 ISO-Prolog modes

Author(s): Daniel Cabeza, Manuel Hermenegildo.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.3#112 (1999/11/23, 1:8:50 MET)

This file defines the " modes" used in the documentation of the ISO-Prolog standard. See also Chapter 60 [Classical Prolog modes], page 255 for an alternative set of modes.

59.1 Usage and interface (`isomodes`)

- **Library usage:**
  `:- use_package([assertions,isomodes]).`
- **New operators defined:**
  `?/1` [200,fy], `@/1` [200,fy].
- **New modes defined:**
  `+/1`, `@/1`, `-/1`, `?/1`, `*/1`, `+/2`, `@/2`, `-/2`, `?/2`, `*/2`.
- **Other modules used:**
  - *System library modules:*
    `assertions/meta_props`.

59.2 Documentation on new modes (`isomodes`)

+/1: MODE

Usage: `+ A`

- *The following properties are added at call time:*
  `A` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

@/1: MODE

Usage: `@(A)`

- *The following properties are added globally:*
  `A` is not further instantiated. (`basic_props:not_further_inst/2`)

-/1: MODE

Usage: `- A`

- *The following properties are added at call time:*
  `A` is a free variable. (`term_typing:var/1`)

?/1: MODE

Unspecified argument.

***/1:** MODE

Unspecified argument.

+/2: MODE

Usage: `A + X`

- *The following properties are added at call time:*
  `A` has property `X`. (`meta_props:call/2`)

@/2: MODE

Usage: `@(A,X)`

- *The following properties are added at call time:*
  `A` has property `X`. (`meta_props:call/2`)
- *The following properties are added upon exit:*
  `A` has property `X`. (`meta_props:call/2`)
- *The following properties are added globally:*
  `A` is not further instantiated. (`basic_props:not_further_inst/2`)

-/2: MODE

Usage: `A - X`

- *The following properties are added at call time:*
  `A` is a free variable. (`term_typing:var/1`)
- *The following properties are added upon exit:*
  `A` has property `X`. (`meta_props:call/2`)

?/2: MODE

Usage: `?(A,X)`

- *Call and exit are compatible with:*
  `A` has property `X`. (`meta_props:call/2`)
- *The following properties are added upon exit:*
  `A` has property `X`. (`meta_props:call/2`)

***/2:** MODE

Usage: `A * X`

- *Call and exit are compatible with:*
  `A` has property `X`. (`meta_props:call/2`)

60 Classical Prolog modes

Author(s): Manuel Hermenegildo.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 0.8#43 (1999/3/6, 18:39:38 CET)

This file defines a number of very simple " modes" which are frequently useful in programs. These correspond to the modes used in classical Prolog texts with some simple addtions. Note that some of these modes use the same symbol as one of the ISO-modes (see Chapter 59 [ISO-Prolog modes], page 253) but with subtly different meaning.

60.1 Usage and interface (`basicmodes`)

- **Library usage:**
  :- use_package([assertions,basicmodes]).
- **New operators defined:**
  `?/1` [500,fx], `@/1` [500,fx].
- **New modes defined:**
  `+/1`, `-/1`, `?/1`, `@/1`, `in/1`, `out/1`, `go/1`, `+/2`, `-/2`, `?/2`, `@/2`, `in/2`, `out/2`, `go/2`.
- **Other modules used:**
  - *System library modules:*
    `assertions/meta_props`.

60.2 Documentation on new modes (`basicmodes`)

+/1: MODE

Input value in argument.

Usage: `+ A`

- *The following properties are added at call time:*
  `A` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

-/1: MODE

No input value in argument.

Usage: `- A`

- *The following properties are added at call time:*
  `A` is a free variable. (`term_typing:var/1`)

?/1: MODE

Unspecified argument.

@/1: MODE

No output value in argument.

Usage: `@(A)`

- *The following properties are added globally:*
  `A` is not further instantiated. (`basic_props:not_further_inst/2`)

in/1: MODE

Input argument.

Usage: `in(A)`

- *The following properties are added at call time:*
  `A` is currently ground (it contains no variables). (`term_typing:ground/1`)
- *The following properties are added upon exit:*
  `A` is currently ground (it contains no variables). (`term_typing:ground/1`)

out/1: MODE

Output argument.

Usage: `out(A)`

- *The following properties are added at call time:*
  `A` is a free variable. (`term_typing:var/1`)
- *The following properties are added upon exit:*
  `A` is currently ground (it contains no variables). (`term_typing:ground/1`)

go/1: MODE

Ground output (input/output argument).

Usage: `go(A)`

- *The following properties are added upon exit:*
  `A` is currently ground (it contains no variables). (`term_typing:ground/1`)

+/2: MODE

Usage: `A + X`

- *Call and exit are compatible with:*
  `A` has property `X`. (`meta_props:call/2`)
- *The following properties are added at call time:*
  `A` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

-/2: MODE

Usage: `A - X`

- *Call and exit are compatible with:*
  `A` has property `X`. (`meta_props:call/2`)
- *The following properties are added at call time:*
  `A` is a free variable. (`term_typing:var/1`)

?/2: MODE

Usage: `?(A,X)`

- *Call and exit are compatible with:*
  `A` has property `X`. (`meta_props:call/2`)

@/2: MODE

Usage: `@(A,X)`

- *Call and exit are compatible with:*
  `A` has property `X`. (`meta_props:call/2`)
- *The following properties are added globally:*
  `A` is not further instantiated. (`basic_props:not_further_inst/2`)

in/2: MODE

Usage: `in(A,X)`

- *Call and exit are compatible with:*
  `A` has property `X`. (`meta_props:call/2`)
- *The following properties are added at call time:*
  `A` is currently ground (it contains no variables). (`term_typing:ground/1`)
- *The following properties are added upon exit:*
  `A` is currently ground (it contains no variables). (`term_typing:ground/1`)

out/2: MODE

Usage: `out(A,X)`

- *Call and exit are compatible with:*
  `A` has property `X`. (`meta_props:call/2`)
- *The following properties are added at call time:*
  `A` is a free variable. (`term_typing:var/1`)
- *The following properties are added upon exit:*
  `A` is currently ground (it contains no variables). (`term_typing:ground/1`)

go/2: MODE

Usage: `go(A,X)`

- *Call and exit are compatible with:*
  `A` has property `X`. (`meta_props:call/2`)
- *The following properties are added upon exit:*
  `A` is currently ground (it contains no variables). (`term_typing:ground/1`)

61 Run-time checking of assertions

Author(s): German Puebla.

This library package allows the use of run-time checks for the assertions introduced in a program.

The recommended way of performing *run-time checks* of predicate assertions in a program is via the Ciao preprocessor (see `ciaopp` manual), which performs the required program transformation. However, this package can also be used to perform checking of program-point assertions.

61.1 Usage and interface (`rtchecks`)

- **Library usage:**
  `:- use_package(rtchecks).`
  or
  `:- module(...,...,[rtchecks]).`
- **Other modules used:**
  - *System library modules:*
    `assertions/meta_props`, `rtchecks/rtchecks_sys`.

61.2 Documentation on multifiles (`rtchecks`)

callme/2: PREDICATE

No further documentation available for this predicate.

The predicate is *multifile*.

callme/1: PREDICATE

No further documentation available for this predicate.

The predicate is *multifile*.

proves/2: PREDICATE

No further documentation available for this predicate.

The predicate is *multifile*.

disproves/2: PREDICATE

No further documentation available for this predicate.

The predicate is *multifile*.

61.3 Known bugs and planned improvements (`rtchecks`)

- All the code in this package is included in the user program when it is used, ant there is a lot of it! A module should be used instead.
- `check/1` uses lists instead of "proper" properties.

PART VI - Ciao Prolog library miscellanea

This part documents several Ciao libraries which provide different useful additional functionalities. Such functionalities include performing operating system calls, gathering statistics from the Prolog engine, file and file name manipulation, error and exception handling, fast reading and writing of terms (marshalling and unmarshalling), file locking, program reporting messages, pretty-printing programs and assertions, a browser of the system libraries, additional expansion utilities, concurrent aggregates, graph visualization, etc.

62 Operating system utilities

Author(s): Daniel Cabeza, Manuel Carro.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.3#50 (1999/9/8, 22:35:15 MEST)

This module contains predicates for invoking services which are typically provided by the operating system. Note that the predicates which take names of files or directories as arguments in this module expect atoms, not path aliases. I.e., generally these predicates will not call `absolute_file_name/2` on names of files or directories taken as arguments.

62.1 Usage and interface (`system`)

- **Library usage:**
  `:- use_module(library(system)).`
- **Exports:**
  - *Predicates:*
    `pause/1`, `time/1`, `datime/1`, `datime/9`, `getenvstr/2`, `extract_paths/2`, `get_pid/1`, `current_host/1`, `current_executable/1`, `umask/2`, `working_directory/2`, `cd/1`, `shell/0`, `shell/1`, `shell/2`, `system/1`, `system/2`, `popen/3`, `exec/4`, `exec/3`, `directory_files/2`, `mktemp/2`, `file_exists/1`, `file_exists/2`, `file_property/2`, `file_properties/6`, `modif_time/2`, `modif_time0/2`, `fmode/2`, `chmod/2`, `chmod/3`, `delete_file/1`.
  - *Regular Types:*
    `datime_struct/1`, `popen_mode/1`.

62.2 Documentation on exports (`system`)

pause/1: PREDICATE

`pause(Seconds)`

Make this thread sleep for some `Seconds`.

time/1: PREDICATE

`time(Time)`

`Time` is unified with the number of seconds elapsed since January, 1, 1970 (UTC).

datime/1: PREDICATE

`datime(Datime)`

`Datime` is unified with a term of the form `datime(Year,Month,Day,Hour,Minute,Second)` which contains the current date and time.

datime/9: PREDICATE

`datime(Time,Year,Month,Day,Hour,Min,Sec,WeekDay,YearDay)`

`Time` is as in `time/1`. If given, the rest of the arguments are unified with the date and time to which the `Time` argument refers. If `Time` is unbound, it is bound to current time and the rest of the arguments refer to current time. `WeekDay` is the number of days since Sunday, in the range 0 to 6. `YearDay` is the number of days since January 1, in the range 0 to 365.

datime_struct/1: REGTYPE

A regular type, defined as follows:

```
datime_struct(datime(Year,Month,Day,Hour,Min,Sec)) :-
        int(Year),
        int(Month),
        int(Day),
        int(Hour),
        int(Min),
        int(Sec).
```

getenvstr/2: PREDICATE

`getenvstr(Name,Value)`

The environment variable `Name` has `Value`. Fails if variable `Name` is not defined.

extract_paths/2: PREDICATE

`extract_paths(String,Paths)`

Interpret `String` as the value of a UNIX environment variable holding a list of paths and return in `Paths` the list of the paths. Paths in `String` are separated by colons, and an empty path is considered a shorthand for '.' (current path). The most typical environment variable with this format is PATH. For example, this is a typical use:

```
?- set_prolog_flag(write_strings, on).

yes
?- getenvstr('PATH', PATH), extract_paths(PATH, Paths).

PATH = ":/home/bardo/bin:/home/clip/bin:/opt/bin/:/bin",
Paths = [".","/home/bardo/bin","/home/clip/bin","/opt/bin/","/bin"] ?

yes
?-
```

get_pid/1: PREDICATE

`get_pid(Pid)`

Unifies `Pid` with the process identificator of the current process or thread.

current_host/1: PREDICATE

`current_host(Hostname)`

`Hostname` is unified with the fully qualified name of the host.

current_executable/1: PREDICATE

`current_executable(Path)`

Unifies `Path` with the path to the current executable.

umask/2: PREDICATE

`umask(OldMask,NewMask)`

The process file creation mask was `OldMask`, and it is changed to `NewMask`.

Usage 2: `umask(OldMask,NewMask)`

- *Description:* Gets the process file creation mask without changing it.
- *The following properties should hold at call time:*
  `OldMask` is a free variable. (`term_typing:var/1`)
  `NewMask` is a free variable. (`term_typing:var/1`)
  The terms `OldMask` and `NewMask` are strictly identical. (`term_compare:== /2`)
- *The following properties hold upon exit:*
  `OldMask` is an integer. (`basic_props:int/1`)
  `NewMask` is an integer. (`basic_props:int/1`)

working_directory/2: PREDICATE

`working_directory(OldDir,NewDir)`

Unifies current working directory with `OldDir`, and then changes the working directory to `NewDir`. Calling `working_directory(Dir,Dir)` simply unifies `Dir` with the current working directory without changing anything else.

Usage 2: `working_directory(OldDir,NewDir)`

- *Description:* Gets current working directory.
- *The following properties should hold at call time:*
  `OldDir` is a free variable. (`term_typing:var/1`)
  `NewDir` is a free variable. (`term_typing:var/1`)
  The terms `OldDir` and `NewDir` are strictly identical. (`term_compare:== /2`)
- *The following properties hold upon exit:*
  `OldDir` is an atom. (`basic_props:atm/1`)
  `NewDir` is an atom. (`basic_props:atm/1`)

cd/1: PREDICATE

`cd(Path)`

Changes working directory to `Path`.

shell/0: PREDICATE

Usage:

– *Description:* Execs the shell specified by the environment variable `SHELL`. When the shell process terminates, control is returned to Prolog.

shell/1: PREDICATE

`shell(Command)`

`Command` is executed in the shell specified by the environment variable `SHELL`. It succeeds if the exit code is zero and fails otherwise.

shell/2: PREDICATE

`shell(Command,ReturnCode)`

Executes `Command` in the shell specified by the environment variable `SHELL` and stores the exit code in `ReturnCode`.

system/1: PREDICATE

`system(Command)`

Executes `Command` using the shell `/bin/sh`.

system/2: PREDICATE

`system(Command,ReturnCode)`

Executes `Command` in the `/bin/sh` shell and stores the exit code in `ReturnCode`.

popen/3: PREDICATE

`popen(Command,Mode,Stream)`

Open a pipe to process `Command` in a new shell with a given `Mode` and return a communication `Stream` (as in UNIX `popen(3)`). If `Mode` is `read` the output from the process is sent to `Stream`. If `Mode` is `write`, `Stream` is sent as input to the process. `Stream` may be read from or written into using the ordinary stream I/O predicates. `Stream` must be closed explicitly using `close/1`, i.e., it is not closed automatically when the process dies.

popen_mode/1: REGTYPE

Usage: `popen_mode(M)`

– *Description:* `M` is 'read' or 'write'.

exec/4: PREDICATE

`exec(Command,StdIn,StdOut,StdErr)`

Starts the process `Command` and returns the I/O streams of the process in `StdIn`, `StdOut`, and `StdErr`.

exec/3: PREDICATE

`exec(Command,StdIn,StdOut)`

Starts the process `Command` and returns the I/O streams of the process in `StdIn` and `StdOut`. `Standard error` is connected to whichever the parent process had it connected to.

directory_files/2: PREDICATE

`directory_files(Directory,FileList)`

`FileList` is the unordered list of entries (files, directories, etc.) in `Directory`.

mktemp/2: PREDICATE

`mktemp(Template,Filename)`

Returns a unique `Filename` based on `Template`: `Template` must be a valid file name with six trailing X, which are substituted to create a new file name.

file_exists/1: PREDICATE

`file_exists(File)`

Succeeds if `File` (a file or directory) exists (and is accessible).

file_exists/2: PREDICATE

`file_exists(File,Mode)`

`File` (a file or directory) exists and it is accessible with `Mode`, as in the Unix call `access(2)`. Typically, `Mode` is 4 for read permission, 2 for write permission and 1 for execute permission.

file_property/2: PREDICATE

`file_property(File,Property)`

`File` has the property `Property`. The possible properties are:

type(`Type`)
: `Type` is one of `regular`, `directory`, `symlink`, `fifo`, `socket` or `unknown`.

linkto(`Linkto`)
: If `File` is a symbolic link, `Linkto` is the file pointed to by the link (and the other properties come from that file, not from the link itself).

mod_time(`ModTime`)
: `ModTime` is the time of last modification (seconds since January, 1, 1970).

mode(`Protection`)
: `Protection` is the protection mode.

size(`Size`)
: `Size` is the size.

If `Property` is uninstantiated, the predicate will enumerate the properties on backtracking.

file_properties/6: PREDICATE

`file_properties(Path,Type,Linkto,Time,Protection,Size)`

The file `Path` has the following properties:

- File type `Type` (one of `regular`, `directory`, `symlink`, `fifo`, `socket` or `unknown`).
- If `Path` is a symbolic link, `Linkto` is the file pointed to. All other properties come from the file pointed, not the link. `Linkto` is '' if `Path` is not a symbolic link.
- Time of last modification `Time` (seconds since January, 1, 1970).
- Protection mode `Protection`.
- Size in bytes `Size`.

modif_time/2: PREDICATE

`modif_time(File,Time)`

The file `File` was last modified at `Time`, which is in seconds since January, 1, 1970. Fails if `File` does not exist.

modif_time0/2: PREDICATE

`modif_time0(File,Time)`

If `File` exists, `Time` is its latest modification time, as in `modif_time/2`. Otherwise, if `File` does not exist, `Time` is zero.

fmode/2: PREDICATE

`fmode(File,Mode)`

The file `File` has protection mode `Mode`.

chmod/2: PREDICATE

`chmod(File,NewMode)`

Change the protection mode of file `File` to `NewMode`.

chmod/3: PREDICATE

`chmod(File,OldMode,NewMode)`

The file `File` has protection mode `OldMode` and it is changed to `NewMode`.

Usage 2: `chmod(File,OldMode,NewMode)`

- *Description:* Equivalent to fmode(`File`,`OldMode`)
- *The following properties should hold at call time:*
  `File` is an atom. (`basic_props:atm/1`)
  `OldMode` is a free variable. (`term_typing:var/1`)
  `NewMode` is a free variable. (`term_typing:var/1`)
  The terms `OldMode` and `NewMode` are strictly identical. (`term_compare:== /2`)
- *The following properties hold upon exit:*
  `File` is an atom. (`basic_props:atm/1`)
  `OldMode` is an atom. (`basic_props:atm/1`)
  `NewMode` is an atom. (`basic_props:atm/1`)

delete_file/1: PREDICATE

`delete_file(File)`

Delete the file `File`.

63 Prolog system internal predicates

Author(s): Manuel Carro, Daniel Cabeza, Mats Carlsson.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.5#27 (1999/12/29, 15:9:45 CET)

This module implements some miscellaneous predicates which provide access to some internal statistics, special properties of the predicates, etc.

63.1 Usage and interface (prolog_sys)

- **Library usage:**
  `:- use_module(library(prolog_sys)).`
- **Exports:**
  - *Predicates:*
    `statistics/0`, `statistics/2`, `predicate_property/2`, `current_atom/1`, `current_predicate/2`, `garbage_collect/0`, `new_atom/1`.

63.2 Documentation on exports (prolog_sys)

statistics/0: PREDICATE

No further documentation available for this predicate.

statistics/2: PREDICATE

Usage 1: `statistics(Time_option,Time_result)`

- *Description:* Gather information about time (either process time or wall time) since last consult or since start of program. Results are returned in milliseconds.
- *The following properties should hold at call time:*
  `Time_option` is any term. (`basic_props:term/1`)
  `Time_result` is any term. (`basic_props:term/1`)
- *The following properties hold upon exit:*
  Options to get information about execution time. `Time_option` must be one of `runtime`, `walltime`. (`prolog_sys:time_option/1`)
  `Time_result` is a two-element list of integers. The first integer is the time since the start of the execution; the second integer is the time since the previous consult to time. (`prolog_sys:time_result/1`)

Usage 2: `statistics(Memory_option,Memory_result)`

- *Description:* Gather information about memory consumption.
- *The following properties should hold at call time:*
  `Memory_option` is any term. (`basic_props:term/1`)
  `Memory_result` is any term. (`basic_props:term/1`)

- *The following properties hold upon exit:*

  Options to get information about memory usage. (`prolog_sys:memory_option/1`)

  Result is a two-element list of integers. The first element is the space taken up by the option selected, measured in bytes; the second integer is zero for program space (which grows as ncessary), and the amount of free space otherwise. (`prolog_sys:memory_result/1`)

Usage 3: `statistics(Garbage_collection_option,Gc_result)`

- *Description:* Gather information about garbage collection.
- *The following properties should hold at call time:*

  `Garbage_collection_option` is any term. (`basic_props:term/1`)

  `Gc_result` is any term. (`basic_props:term/1`)
- *The following properties hold upon exit:*

  Options to get information about garbage collection. (`prolog_sys:garbage_collection_option/1`)

  `Gc_result` is a tree-element list of integers, related to garbage collection and memory management. When `stack_shifts` is selected, the first one is the number of shifts (reallocations) of the local stack; the second is the number of shifts of the trail, and the third is the time spent in these shifts. When `garbage_collection` is selected, the numbers are, respectively, the number of garbage collections performed, the number of bytes freed, and the time spent in garbage collection. (`prolog_sys:gc_result/1`)

Usage 4: `statistics(Symbol_option,Symbol_result)`

- *Description:* Gather information about number of symbols and predicates.
- *The following properties should hold at call time:*

  `Symbol_option` is any term. (`basic_props:term/1`)

  `Symbol_result` is any term. (`basic_props:term/1`)
- *The following properties hold upon exit:*

  Option to get information about the number of symbols in the program. (`prolog_sys:symbol_option/1`)

  `Symbol_result` is a two-element list of integers. The first one is the number of atom, functor, and predicate names in the symbol table. The second is the number of predicates known to be defined (although maybe without clauses). (`prolog_sys:symbol_result/1`)

predicate_property/2: PREDICATE

Usage: `predicate_property(Head,Property)`

- *Description:* The predicate with clause `Head` is `Property`.
- *The following properties should hold at call time:*

  `Head` is any term. (`basic_props:term/1`)

  `Property` is any term. (`basic_props:term/1`)
- *The following properties hold upon exit:*

  `Head` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

  `Property` is an atom. (`basic_props:atm/1`)

current_atom/1: PREDICATE

No further documentation available for this predicate.

current_predicate/2: PREDICATE

No further documentation available for this predicate.

garbage_collect/0: PREDICATE

No further documentation available for this predicate.

new_atom/1: PREDICATE

Usage: `new_atom(Atom)`

- *Description:* Returns, on success, a new atom, not existing before in the system. The entry argument must be a variable. The idea behind this atom generation is to provide a fast source of identifiers for new objects, concurrent predicates, etc. on the fly.
- *The following properties should hold at call time:*
  `Atom` is a free variable. (`term_typing:var/1`)
- *The following properties hold upon exit:*
  `Atom` is an atom. (`basic_props:atm/1`)

63.3 Documentation on internals (`prolog_sys`)

time_option/1: REGTYPE

Usage: `time_option(M)`

- *Description:* Options to get information about execution time. `M` must be one of `runtime`, `walltime`.

memory_option/1: REGTYPE

Usage: `memory_option(M)`

- *Description:* Options to get information about memory usage.

garbage_collection_option/1: REGTYPE

Usage: `garbage_collection_option(M)`

- *Description:* Options to get information about garbage collection.

symbol_option/1: REGTYPE

Usage: `symbol_option(M)`

- *Description:* Option to get information about the number of symbols in the program.

time_result/1: REGTYPE

Usage: `time_result(Result)`

- *Description:* `Result` is a two-element list of integers. The first integer is the time since the start of the execution; the second integer is the time since the previous consult to time.

memory_result/1: REGTYPE

Usage: `memory_result(Result)`

- *Description:* Result is a two-element list of integers. The first element is the space taken up by the option selected, measured in bytes; the second integer is zero for program space (which grows as ncessary), and the amount of free space otherwise.

gc_result/1: REGTYPE

Usage: `gc_result(Result)`

- *Description:* `Result` is a tree-element list of integers, related to garbage collection and memory management. When `stack_shifts` is selected, the first one is the number of shifts (reallocations) of the local stack; the second is the number of shifts of the trail, and the third is the time spent in these shifts. When `garbage_collection` is selected, the numbers are, respectively, the number of garbage collections performed, the number of bytes freed, and the time spent in garbage collection.

symbol_result/1: REGTYPE

Usage: `symbol_result(Result)`

- *Description:* `Result` is a two-element list of integers. The first one is the number of atom, functor, and predicate names in the symbol table. The second is the number of predicates known to be defined (although maybe without clauses).

63.4 Known bugs and planned improvements (`prolog_sys`)

- The space used by the process is not measured here: process data, code, and stack also take up memory. The memory reported for atoms is not what is actually used, but the space used up by the hash table (which is enlarged as needed).

64 ctrlcclean (library)

Version: 0.4#5 (1998/2/24)

64.1 Usage and interface (ctrlcclean)

- **Library usage:**
  `:- use_module(library(ctrlcclean)).`
- **Exports:**
  - *Predicates:*
    `ctrlc_clean/1`, `delete_on_ctrlc/2`, `ctrlcclean/0`.
- **Other modules used:**
  - *System library modules:*
    `system`.

64.2 Documentation on exports (ctrlcclean)

ctrlc_clean/1: PREDICATE

No further documentation available for this predicate.

Meta-predicate with arguments: `ctrlc_clean(goal)`.

delete_on_ctrlc/2: PREDICATE

No further documentation available for this predicate.

ctrlcclean/0: PREDICATE

No further documentation available for this predicate.

65 errhandle (library)

Version: 0.4#5 (1998/2/24)

65.1 Usage and interface (`errhandle`)

- **Library usage:**
  `:- use_module(library(errhandle)).`
- **Exports:**
  - *Predicates:*
    `error_protect/1`, `handle_error/2`.

65.2 Documentation on exports (`errhandle`)

error_protect/1: PREDICATE

No further documentation available for this predicate.

Meta-predicate with arguments: `error_protect(goal)`.

handle_error/2: PREDICATE

No further documentation available for this predicate.

66 Fast Reading and Writing of Terms

Author(s): Daniel Cabeza, Oscar Portela Arjona.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.5#81 (2000/3/23, 16:37:17 CET)

This library provides predicates to support reading / writing of terms on a format designed to be handled on read faster than standart representation.

66.1 Usage and interface (`fastrw`)

- **Library usage:**
  `:- use_module(library(fastrw)).`
- **Exports:**
  - *Predicates:*
    `fast_read/1`, `fast_write/1`, `fast_write_to_string/3`.
- **Other modules used:**
  - *System library modules:*
    `dict`.

66.2 Documentation on exports (`fastrw`)

fast_read/1: PREDICATE

`fast_read(Term)`

The next term is read from current standart input and is unified with `Term`. The syntax of the term must agree with fast_read / fast_write format. If the end of the input has been reached, `Term` is unified with the term 'end_of_file'. Further calls to `fast_read/1` will then cause an error.

fast_write/1: PREDICATE

`fast_write(Term)`

Output `Term` in a way that `fast_read/1` will be able to read it back.

fast_write_to_string/3: PREDICATE

No further documentation available for this predicate.

67 File name manipulation

Author(s): Daniel Cabeza, Angel Fernandez Pineda.
Version: 1.5#118 (2000/4/19, 18:13:43 CEST)
Version of last change: 1.3#51 (1999/9/9, 16:28:44 MEST)
This library provides some small utilities to handle file name syntax.

67.1 Usage and interface (`filenames`)

- **Library usage:**
  `:- use_module(library(filenames)).`
- **Exports:**
  - *Predicates:*
    `no_path_file_name/2`, `file_name_extension/3`, `basename/2`, `extension/2`.
- **Other modules used:**
  - *System library modules:*
    `lists`.

67.2 Documentation on exports (`filenames`)

no_path_file_name/2: PREDICATE

This predicate will extract the last item (usually the file name) from a given path.

The first argument must be instantiated to a string or atom. Whenever the first argument is an atom, the second argument will be an atom. Whenever the first argument is a string, the second argument will be a string.

This predicate will fail under any of the following conditions:

- First argument is not an atom, nor a string.
- Second argument is not the last given path item (given path is the first argument).

Those are the most usual usages of no_path_file_name/2:

```
?- no_path_file_name_("/home/nexusV/somefile.txt",K).

K = "somefile.txt" ?

yes
?- no_path_file_name('/home/nexusV/somefile.txt',K).

K = 'somefile.txt' ?

yes
?-
```

Usage: `no_path_file_name(Path,FileName)`

- *Description:* `FileName` is the file corresponding to the given `Path`.
- *Call and exit should be compatible with:*
  `Path` is an atom or a string (`filenames:atom_or_str/1`)
  `FileName` is an atom or a string (`filenames:atom_or_str/1`)

file_name_extension/3: PREDICATE

This predicate may be used in two ways:

- To create a file name from its components: name and extension. For instance:

```
?- file_name_extension(File,mywork,'.txt').

File = 'mywork.txt' ?

yes
?-
```

- To split a file name into its name and extension. For Instance:

```
?- file_name_extension('mywork.txt',A,B).

A = mywork,
B = '.txt' ?

yes
?-
```

Any other usage of file_name_extension/3 will cause the predicate to fail. Notice that valid arguments are accepted both as atoms or strings.

Usage: `file_name_extension(FileName,BaseName,Extension)`

- *Description:* Splits a `FileName` into its `BaseName` and `Extension`.
- *Call and exit should be compatible with:*

  `FileName` is an atom or a string (`filenames:atom_or_str/1`)
  `BaseName` is an atom or a string (`filenames:atom_or_str/1`)
  `Extension` is an atom or a string (`filenames:atom_or_str/1`)

basename/2: PREDICATE

`basename(FileName,BaseName)`

`BaseName` is `FileName` without extension. Equivalent to `file_name_extension(FileName,BaseName,_)`. Useful to extract the base name of a file using functional syntax.

Usage:

- *Calls should, and exit will be compatible with:*

  `FileName` is an atom or a string (`filenames:atom_or_str/1`)
  `BaseName` is an atom or a string (`filenames:atom_or_str/1`)

extension/2: PREDICATE

`extension(FileName,Extension)`

`Extension` is the extension (suffix) of `FileName`. Equivalent to `file_name_extension(FileName,_,Extension)`. Useful to extract the extension of a file using functional syntax.

Usage:

- *Calls should, and exit will be compatible with:*

  `FileName` is an atom or a string (`filenames:atom_or_str/1`)
  `Extension` is an atom or a string (`filenames:atom_or_str/1`)

68 file_utils (library)

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)
Version of last change: 1.5#55 (2000/2/11, 21:19:43 CET)

68.1 Usage and interface (file_utils)

- **Library usage:**
  `:- use_module(library(file_utils)).`
- **Exports:**
  - *Predicates:*
    `file_terms/2`, `copy_stdout/1`, `file_to_string/2`, `stream_to_string/2`.
- **Other modules used:**
  - *System library modules:*
    `read`, `streams`.

68.2 Documentation on exports (file_utils)

file_terms/2: PREDICATE

Usage 1: `file_terms(@(File),?(Terms))`
- *Description:* Transform a file `File` to/from a list of terms `Terms`.
- *The following properties should hold upon exit:*
  `@(File)` is a source name. (`streams_basic:sourcename/1`)
  `?(Terms)` is a list. (`basic_props:list/1`)

Usage 2: `file_terms(File,Terms)`
- *Description:* Unifies `Terms` with the list of all terms in `File`.
- *The following properties should hold at call time:*
  `File` is a source name. (`streams_basic:sourcename/1`)
  `Terms` is a free variable. (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  `File` is a source name. (`streams_basic:sourcename/1`)
  `Terms` is a list. (`basic_props:list/1`)

Usage 3: `file_terms(File,Terms)`
- *Description:* Writes the terms in list `Terms` (including the ending '.') onto file `File`.
- *The following properties should hold at call time:*
  `File` is a source name. (`streams_basic:sourcename/1`)
  `Terms` is a list. (`basic_props:list/1`)
- *The following properties should hold upon exit:*
  `File` is a source name. (`streams_basic:sourcename/1`)
  `Terms` is a list. (`basic_props:list/1`)

copy_stdout/1: PREDICATE

Usage: `copy_stdout(+File)`

- *Description:* Copies file `File` to standard output.
- *The following properties should hold upon exit:*
  `+File` is a source name. (`streams_basic:sourcename/1`)

file_to_string/2: PREDICATE

Usage: `file_to_string(+FileName,-(String))`

- *Description:* Reads all the characters from the file `FileName` and returns them in `String`.
- *Call and exit should be compatible with:*
  `+FileName` is a source name. (`streams_basic:sourcename/1`)
  `-(String)` is a string (a list of character codes). (`basic_props:string/1`)

stream_to_string/2: PREDICATE

Usage: `stream_to_string(+Stream,-(String))`

- *Description:* Reads all the characters from `Stream` and returns them in `String`.
- *Call and exit should be compatible with:*
  `+Stream` is an open stream. (`streams_basic:stream/1`)
  `-(String)` is a string (a list of character codes). (`basic_props:string/1`)

69 file_locks (library)

69.1 Usage and interface (`file_locks`)

- **Library usage:**
  `:- use_module(library(file_locks)).`
- **Exports:**
  - *Predicates:*
    `lock_file/3`, `unlock_file/2`.

69.2 Documentation on exports (`file_locks`)

lock_file/3: PREDICATE

No further documentation available for this predicate.

unlock_file/2: PREDICATE

No further documentation available for this predicate.

70 terms (library)

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)
Version of last change: 1.3#124 (1999/11/27, 4:4:5 MET)

70.1 Usage and interface (terms)

- **Library usage:**
  `:- use_module(library(terms)).`
- **Exports:**
  - *Predicates:*
    `copy_args/3`, `arg/2`, `atom_concat/2`.

70.2 Documentation on exports (terms)

copy_args/3: PREDICATE
No further documentation available for this predicate.

arg/2: PREDICATE
No further documentation available for this predicate.

atom_concat/2: PREDICATE
`atom_concat(Atms,Atm)`

`Atm` is the atom resulting from concatenating all atoms in the list `Atms` in the order in which they appear.

71 tokenize (library)

Version: 0.4#5 (1998/2/24)

71.1 Usage and interface (`tokenize`)

- **Library usage:**
  `:- use_module(library(tokenize)).`
- **Exports:**
  - *Predicates:*
    `read_tokens/2`.
  - *Multifiles:*
    `define_flag/3`.
- **Other modules used:**
  - *System library modules:*
    `read`, `dict`.

71.2 Documentation on exports (`tokenize`)

read_tokens/2: PREDICATE

No further documentation available for this predicate.

71.3 Documentation on multifiles (`tokenize`)

define_flag/3: PREDICATE

No further documentation available for this predicate.

The predicate is *multifile*.

72 Printing status and error messages

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.3#108 (1999/11/18, 13:48:3 MET)

This is a very simple library for printing status and error messages to the console.

72.1 Usage and interface (messages)

- **Library usage:**
  `:- use_module(library(messages)).`
- **Exports:**
  - *Predicates:*
    `error_message/1`, `error_message/2`, `error_message/3`, `warning_message/1`, `warning_message/2`, `warning_message/3`, `note_message/1`, `note_message/2`, `note_message/3`, `simple_message/1`, `simple_message/2`, `optional_message/2`, `optional_message/3`, `debug_message/1`, `debug_message/2`, `debug_goal/2`, `debug_goal/3`.
  - *Multifiles:*
    `callme/2`, `issue_debug_messages/1`.
- **Other modules used:**
  - *System library modules:*
    `assertions/meta_props`, `format`, `lists`, `filenames`.

72.2 Documentation on exports (messages)

error_message/1: PREDICATE

Meta-predicate with arguments: `error_message(addmodule)`.

Usage: `error_message(Text)`

- *Description:* The text provided in `Text` is printed as an ERROR message.
- *The following properties should hold at call time:*
  `Text` is a string (a list of character codes). (`basic_props:string/1`)

error_message/2: PREDICATE

Meta-predicate with arguments: `error_message(?,addmodule)`.

Usage: `error_message(Text,ArgList)`

- *Description:* The text provided in `Text` is printed as an ERROR message, using the arguments in `ArgList` to interpret any variable-related formatting commands embedded in `Text`.
- *The following properties should hold at call time:*
  `Text` is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with `name/2`. (`format:format_control/1`)
  `ArgList` is a list. (`basic_props:list/1`)

error_message/3: PREDICATE

Meta-predicate with arguments: `error_message(?,?,addmodule)`.

Usage: `error_message(Lc,Text,ArgList)`

- *Description:* The text provided in `Text` is printed as an ERROR message, using the arguments in `ArgList` to interpret any variable-related formatting commands embedded in `Text`, and reporting error location `Lc` (file and line numbers).
- *The following properties should hold at call time:*

  `Lc` is of type `^loc(atm,int,int)`. (`meta_props:regtype/2`)

  `Text` is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with `name/2`. (`format:format_control/1`)

  `ArgList` is a list. (`basic_props:list/1`)

warning_message/1: PREDICATE

Meta-predicate with arguments: `warning_message(addmodule)`.

Usage: `warning_message(Text)`

- *Description:* The text provided in `Text` is printed as a WARNING message.
- *The following properties should hold at call time:*

  `Text` is a string (a list of character codes). (`basic_props:string/1`)

warning_message/2: PREDICATE

Meta-predicate with arguments: `warning_message(?,addmodule)`.

Usage: `warning_message(Text,ArgList)`

- *Description:* The text provided in `Text` is printed as a WARNING message, using the arguments in `ArgList` to interpret any variable-related formatting commands embedded in `Text`.
- *The following properties should hold at call time:*

  `Text` is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with `name/2`. (`format:format_control/1`)

  `ArgList` is a list. (`basic_props:list/1`)

warning_message/3: PREDICATE

Meta-predicate with arguments: `warning_message(?,?,addmodule)`.

Usage: `warning_message(Lc,Text,ArgList)`

- *Description:* The text provided in `Text` is printed as a WARNING message, using the arguments in `ArgList` to interpret any variable-related formatting commands embedded in `Text`, and reporting error location `Lc` (file and line numbers).
- *The following properties should hold at call time:*

  `Lc` is of type `^loc(atm,int,int)`. (`meta_props:regtype/2`)

  `Text` is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with `name/2`. (`format:format_control/1`)

  `ArgList` is a list. (`basic_props:list/1`)

note_message/1: PREDICATE

Meta-predicate with arguments: `note_message(addmodule)`.

Usage: `note_message(Text)`

- *Description:* The text provided in `Text` is printed as a NOTE.
- *The following properties should hold at call time:*

  `Text` is a string (a list of character codes). (`basic_props:string/1`)

note_message/2: PREDICATE

Meta-predicate with arguments: `note_message(?,addmodule)`.

Usage: `note_message(Text,ArgList)`

- *Description:* The text provided in `Text` is printed as a NOTE, using the arguments in `ArgList` to interpret any variable-related formatting commands embedded in `Text`.
- *The following properties should hold at call time:*

  `Text` is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with `name/2`. (`format:format_control/1`)

  `ArgList` is a list. (`basic_props:list/1`)

note_message/3: PREDICATE

Meta-predicate with arguments: `note_message(?,?,addmodule)`.

Usage: `note_message(Lc,Text,ArgList)`

- *Description:* The text provided in `Text` is printed as a NOTE, using the arguments in `ArgList` to interpret any variable-related formatting commands embedded in `Text`, and reporting error location `Lc` (file and line numbers).
- *The following properties should hold at call time:*

  `Lc` is of type `^loc(atm,int,int)`. (`meta_props:regtype/2`)

  `Text` is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with `name/2`. (`format:format_control/1`)

  `ArgList` is a list. (`basic_props:list/1`)

simple_message/1: PREDICATE

Usage: `simple_message(Text)`

- *Description:* The text provided in `Text` is printed.
- *The following properties should hold at call time:*

  `Text` is a string (a list of character codes). (`basic_props:string/1`)

simple_message/2: PREDICATE

Usage: `simple_message(Text,ArgList)`

- *Description:* The text provided in `Text` is printed as a message, using the arguments in `ArgList`.
- *The following properties should hold at call time:*

  `Text` is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with `name/2`. (`format:format_control/1`)

  `ArgList` is a list. (`basic_props:list/1`)

optional_message/2: PREDICATE

Usage: `optional_message(Text,Opts)`

- *Description:* The text provided in `Text` is printed as a message, but only if the atom `-v` is a member of `Opts`. These predicates are meant to be used for optional messages, which are only to be printed when *verbose* output is requested explicitly.
- *The following properties should hold at call time:*

  `Text` is a string (a list of character codes). (`basic_props:string/1`)

  `Opts` is a list of `atm`s. (`basic_props:list/2`)

optional_message/3: PREDICATE

Usage: `optional_message(Text,ArgList,Opts)`

- *Description:* The text provided in `Text` is printed as a message, using the arguments in `ArgList`, but only if the atom `-v` is a member of `Opts`. These predicates are meant to be used for optional messages, which are only to be printed when *verbose* output is requested explicitly.
- *The following properties should hold at call time:*

  `Text` is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with `name/2`. (`format:format_control/1`)

  `ArgList` is a list. (`basic_props:list/1`)

  `Opts` is a list of `atm`s. (`basic_props:list/2`)

debug_message/1: PREDICATE

Meta-predicate with arguments: `debug_message(addmodule)`.

Usage: `debug_message(Text)`

- *Description:* The text provided in `Text` is printed as a debugging message. These messages are turned `on` by defining a fact of `issue_debug_messages/1` with the module name as argument.
- *The following properties should hold at call time:*

  `Text` is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with `name/2`. (`format:format_control/1`)

debug_message/2: PREDICATE

Meta-predicate with arguments: `debug_message(?,addmodule)`.

Usage: `debug_message(Text,ArgList)`

- *Description:* The text provided in `Text` is printed as a debugging message, using the arguments in `ArgList` to interpret any variable-related formatting commands embedded in `Text`. These messages are turned `on` by defining a fact of `issue_debug_messages/1` which the module name as argument.
- *The following properties should hold at call time:*

  `Text` is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with `name/2`. (`format:format_control/1`)

  `ArgList` is a list. (`basic_props:list/1`)

debug_goal/2: PREDICATE

Meta-predicate with arguments: `debug_goal(goal,addmodule)`.

Usage: `debug_goal(Goal,Text)`

– *Description:* `Goal` is called. The text provided in `Text` is then printed as a debugging message. The whole process (including running `Goal`) is turned `on` by defining a fact of `issue_debug_messages/1` with the module name as argument.

debug_goal/3: PREDICATE

Meta-predicate with arguments: `debug_goal(goal,?,addmodule)`.

Usage: `debug_goal(Goal,Text,ArgList)`

– *Description:* `Goal` is called. The text provided in `Text` is then printed as a debugging message, using the arguments in `ArgList` to interpret any variable-related formatting commands embedded in `Text`. Note that the variables in `ArgList` can be computed by `Goal`. The whole process (including running `Goal`) is turned `on` by defining a fact of `issue_debug_messages/1` with the module name as argument.

72.3 Documentation on multifiles (`messages`)

callme/2: PREDICATE

No further documentation available for this predicate.

The predicate is *multifile*.

issue_debug_messages/1: PREDICATE

The predicate is *multifile*.

The predicate is of type *data*.

Usage: `issue_debug_messages(Module)`

– *Description:* Printing of debugging messages is enabled for module `Module`.

– *The following properties should hold upon exit:*

`Module` is currently instantiated to an atom. (`term_typing:atom/1`)

72.4 Known bugs and planned improvements (`messages`)

- Debug message switching should really be done with an expansion, for performance.

73 A simple pretty-printer for Ciao programs

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 0.9#100 (1999/5/26, 12:36:46 MEST)

This library module writes out to standard output a clause or a list of clauses.

73.1 Usage and interface (pretty_print)

- **Library usage:**
  `:- use_module(library(pretty_print)).`
- **Exports:**
  - *Predicates:*
    `pretty_print/2`, `pretty_print/3`.
- **Other modules used:**
  - *System library modules:*
    `vndict`, `write`.

73.2 Documentation on exports (pretty_print)

pretty_print/2: PREDICATE

Usage: `pretty_print(Cls,Flags)`

- *Description:* Prints each clause in the list `Cls` after numbering its variables.
- *The following properties should hold at call time:*
  `pretty_print:clauses(Cls)` (`pretty_print:clauses/1`)
  `Flags` is a list of `flags`. (`basic_props:list/2`)

pretty_print/3: PREDICATE

Usage: `pretty_print(Cls,Flags,Ds)`

- *Description:* Prints each clause in the list `Cls` after using the corresponding variable names dictionary in `Ds` to name its variables.
- *The following properties should hold at call time:*
  `pretty_print:clauses(Cls)` (`pretty_print:clauses/1`)
  `Flags` is a list of `flags`. (`basic_props:list/2`)
  `Ds` is a dictionary of variable names. (`vndict:varnamedict/1`)

73.3 Documentation on internals (pretty_print)

clauses/1: REGTYPE

A regular type, defined as follows:

```
clauses([]).
clauses([_1|_2]) :-
        clause(_1),
        clauses(_2).
clauses(_1) :-
        clause(_1).
```

clause/1: REGTYPE

A regular type, defined as follows:

```
clause(_1) :-
        clterm(_1).
clause((_1,_2)) :-
        clterm(_1),
        term(_2).
```

clterm/1: REGTYPE

A regular type, defined as follows:

```
clterm(clause(_1,_2)) :-
        callable(_1),
        body(_2).
clterm(directive(_1)) :-
        body(_1).
clterm((_1:-_2)) :-
        callable(_1),
        body(_2).
clterm(_1) :-
        callable(_1).
```

body/1: REGTYPE

A well formed body, including cge expressions and &-concurrent expressions. The atomic goals may or may not have a key in the form `^(goal:any)`, and may or may not be module qualified, but if they are it has to be in the form `^(^(moddesc:goal):any)`.

Usage: `body(X)`

– *Description:* `X` is a printable body.

flag/1: REGTYPE

A keyword `ask/1` flags whether to output *asks* or *whens* and `nl/1` whether to separate clauses with a blank line or not.

Usage: `flag(X)`

– *Description:* `X` is a flag for the pretty-printer.

74 Pretty-printing assertions

Author(s): Francisco Bueno Carrillo.
Version: 1.5#118 (2000/4/19, 18:13:43 CEST)
Version of last change: 1.5#2 (1999/11/29, 18:2:53 MET)

This module defines some predicates which are useful for writing assertions in a readable form.

74.1 Usage and interface (assrt_write)

- **Library usage:**
  `:- use_module(library(assrt_write)).`
- **Exports:**
  - *Predicates:*
    `write_assertion/6`, `write_assertion_as_comment/6`.
- **Other modules used:**
  - *System library modules:*
    `format`, `assertions/assrt_lib`, `messages`, `assertions/assertions_props`.

74.2 Documentation on exports (assrt_write)

write_assertion/6: PREDICATE

Usage: `write_assertion(Goal,Status,Type,Body,Dict,Flag)`

- *Description:* Writes the (normalized) assertion to current output.
- *Call and exit should be compatible with:*
  `Status` is an acceptable status for an assertion. (`assertions_props:assrt_status/1`)
  `Type` is an admissible kind of assertion. (`assertions_props:assrt_type/1`)
  `Body` is a normalized assertion body. (`assertions_props:nabody/1`)
  `Dict` is a dictionary of variable names. (`assertions_props:dictionary/1`)
  `Flag` is `status` or `nostatus`. (`assrt_write:status_flag/1`)

write_assertion_as_comment/6: PREDICATE

Usage: `write_assertion_as_comment(Goal,Status,Type,Body,Dict,Flag)`

- *Description:* Writes the (normalized) assertion to current output as a Prolog comment.
- *Call and exit should be compatible with:*
  `Status` is an acceptable status for an assertion. (`assertions_props:assrt_status/1`)
  `Type` is an admissible kind of assertion. (`assertions_props:assrt_type/1`)
  `Body` is a normalized assertion body. (`assertions_props:nabody/1`)
  `Dict` is a dictionary of variable names. (`assertions_props:dictionary/1`)
  `Flag` is `status` or `nostatus`. (`assrt_write:status_flag/1`)

75 The Ciao library browser

Author(s): Angel Fernandez Pineda.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.3#39 (1999/7/21, 12:8:4 MEST)

`librowser` library provides a set of predicates wich enables the user to interactively find Ciao/Prolog libraries and/or any predicate exported by them.

This is a simple example:

```
?- apropos('*find*').
persdbrt_sql: dbfindall/4
persdbrtsql: dbfindall/4
conc_aggregates: findall/3
linda: rd_findall/3
vndict: find_name/4
internals: $find_file/8
aggregates: findall/4,findall/3

yes
?-
```

Librowser is specially usefull when using inside GNU Emacs, just place the cursor over a librowser response and press C-cTAB in order to get help on the related predicate. Refer to the **"Using Ciao inside GNU Emacs"** chapter for further information.

75.1 Usage and interface (`librowser`)

- **Library usage:**

  It is not necesary to use this library at user programs. It was designed to be used at the Ciao *toplevel* shell: `ciaosh`. In order to do so, just make use of `use_module/1` as follows:

  `use_module(library(librowser)).`

  Then, the library interface must be read. This is automatically done when calling any predicate at librowser, and the entire process will take a little moment.So, you should want to perform such a process after loading the Ciao toplevel:

  ```
  Ciao 0.9 #75: Fri Apr 30 19:04:24 MEST 1999
  ?- use_module(library(librowser)).

  yes
  ?- update.
  ```

  Whether you want this process to be automatically performed when loading `ciaosh`, you may include those lines in your *.ciaorc* personal initialization file.
- **Exports:**
  - *Predicates:*

    `update/0`, `browse/2`, `where/1`, `describe/1`, `system_lib/1`, `apropos/1`.
- **Other modules used:**
  - *System library modules:*

    `filenames`, `read`, `system`, `streams`, `patterns`, `lists`.

75.2 Documentation on exports (`librowser`)

update/0: PREDICATE

This predicate will scan the Ciao system libraries for predicate definitions. This may be done once time before calling any other predicate at this library.

update/0 will also be automatically called (once) when calling any other predicate at librowser.

Usage:

- *Description:* Creates an internal database of modules at Ciao system libraries.

browse/2: PREDICATE

This predicate is fully reversible, and is provided to inspect concrete predicate specifications. For example:

```
?- browse(M,findall/A).

A = 3,
M = conc_aggregates ? ;

A = 4,
M = aggregates ? ;

A = 3,
M = aggregates ? ;

no
?-
```

Usage: `browse(Module,Spec)`

- *Description:* Asocciates the given `Spec` predicate specification with the `Module` which exports it.
- *The following properties should hold at call time:*
  `Module` is a module name (an atom) (`librowser:module_name/1`)
  `Spec` is a **Functor/Arity** predicate specification (`librowser:pred_spec/1`)

where/1: PREDICATE

This predicate will print at the screen the module needed in order to import a given predicate specification. For example:

```
?- where(findall/A).
findall/3 exported at module conc_aggregates
findall/4 exported at module aggregates
findall/3 exported at module aggregates

yes
?-
```

Usage: `where(Spec)`

- *Description:* Display what module to load in order to import the given `Spec`.
- *The following properties should hold at call time:*
  `Spec` is a **Functor/Arity** predicate specification (`librowser:pred_spec/1`)

describe/1: PREDICATE

This one is used to find out which predicates were exported by a given module. Very usefull when you know the library, but not the concrete predicate. For example:

```
?- describe(librowser).
Predicates at library librowser :

apropos/1
system_lib/1
describe/1
where/1
browse/2
update/0

yes
?-
```

Usage: `describe(Module)`

- *Description:* Display a list of exported predicates at the given `Module`
- *The following properties should hold at call time:*
  `Module` is a module name (an atom) (`librowser:module_name/1`)

system_lib/1: PREDICATE

It retrieves on backtracking all Ciao system libraries stored in the internal database. Certainly, those which were scanned at `update/0` calling.

Usage: `system_lib(Module)`

- *Description:* `Module` variable will be successively instantiated to the system libaries stored in the internal database.
- *The following properties should hold at call time:*
  `Module` is a module name (an atom) (`librowser:module_name/1`)

apropos/1: PREDICATE

This tool makes use of regular expresions in order to find predicate specifications. It is very usefull whether you can't renember the full name of a predicate. Regular expresions take the same format as described in library `patterns`. Example:

```
?- apropos('atom_*').

metaterms: atom_concat/2
concurrency: atom_lock_state/2
atomic_basic: atom_concat/3,atom_length/2,atom_codes/2
iso_byte_char: atom_chars/2

yes
?-
```

Usage: `apropos(RegSpec)`

- *Description:* This will search any predicate specification `Spec` which matches the given `RegSpec` incomplete predicate specification.
- *The following properties should hold at call time:*
  `RegSpec` is a Pattern/Arity specification. (`librowser:apropos_spec/1`)

75.3 Documentation on internals (`librowser`)

apropos_spec/1: REGTYPE

Defined as:

```
apropos_spec(_1).
apropos_spec(Pattern/Arity) :-
        pattern(Pattern),
        int(Arity).
```

Usage: `apropos_spec(S)`

– *Description:* `S` is a Pattern/Arity specification.

76 Code translation utilities

Author(s): Angel Fernandez Pineda.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.3#110 (1999/11/22, 11:52:54 MET)

This library offers a general way to perform clause body expansions. Goal, fact and spec translation predicates are authomatically called when needed, while this utility navigates through the meta-argument specification of the body itself. All predicates within this library must be called at *second-pass expansions*, since it uses information stored at `c_itf` library.

76.1 Usage and interface (`expansion_tools`)

- **Library usage:**
  This library is provided as a tool for those modules which performs source-to-source code translation, usually known as *code expanders*. It may be loaded as other modules using a `use_module/1`. Nothing special needs to be done.
- **Exports:**
  - *Predicates:*
    `imports_meta_pred/3`, `body_expander/6`, `arg_expander/6`.
- **Other modules used:**
  - *System library modules:*
    `compiler/c_itf`.

76.2 Documentation on exports (`expansion_tools`)

imports_meta_pred/3: PREDICATE

Macro provided in order to know meta-predicate specifications accessible from a module.

Usage: `imports_meta_pred(Module,MetaSpec,AccessibleAt)`

- *Description:* Tells whether `MetaSpec` meta-predicate specification is accessible from `Module`. `AccessibleAt` will be binded to '-' whether meta-predicate is a builtin one. If not, it will be unified with the module which defines the meta-predicate.
- *The following properties should hold at call time:*
  `Module` is an atom. (`basic_props:atm/1`)
  `MetaSpec` is any term. (`basic_props:term/1`)
  `AccessibleAt` is a free variable. (`term_typing:var/1`)

body_expander/6: PREDICATE

This predicate is the main translation tool. It navigates through a clause body, when a single *goal* appears, user-code is called in order to perform a translation. Whether user-code fails to translate the involved goal, it remains the same. Regardless that goal is translated or not, an argument expansion will be performed over all goals if applicable (see `arg_expander/6` predicate).

Variable (unknown at compile time) goals will also be attempt to translate.

Meta-predicate with arguments: `body_expander(pred(3),pred(3),pred(3),?,?,?)`.

Usage:
`body_expander(GoalTrans,FactTrans,SpecTrans,Module,Body,ExpandedBody)`

- *Description:* Translates `Body` to `ExpandedBody` by the usage of user-defined translators `GoalTrans`, `FactTrans` and `SpecTrans`. The module where the original body appears must be unified with `Module` argument.
- *The following properties should hold at call time:*

  `GoalTrans` is a user-defined predicate which performs *goal* meta-type translation (`expansion_tools:goal_expander/1`)

  `FactTrans` is a user-defined predicate which performs *fact* meta-type translation (`expansion_tools:fact_expander/1`)

  `SpecTrans` is a user-defined predicate which performs *spec* meta-type translation (`expansion_tools:spec_expander/1`)

  `Module` is an atom. (`basic_props:atm/1`)

  `Body` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `ExpandedBody` is a free variable. (`term_typing:var/1`)

arg_expander/6: PREDICATE

This predicate is an auxiliary translation tool, which is used by `body_expander/6` predicate. It remains exported as a macro. The predicate navigates through the *meta-argument specification* of a goal. Whether a *goal,fact or spec* argument appears, user-code is called in order to perform a translation. Whether user-code fails to translate the involved argument, it remains the same. Builtins as ','/2 or ';'/2 are treated as meta-predicates defining *goal* meta-arguments. When a *goal* meta-argument is located, `body_expander/6` will be called in order to navigate through it. Notice that a *goal* meta-argument may be unified with another goal defining another meta-argument, so navigation is required. If arguments are not known to arg_expander/6, translation will not occur. This is posible whether goal or qualifing module are variables.

Meta-predicate with arguments: `arg_expander(pred(3),pred(3),pred(3),?,?,?)`.

Usage:
`arg_expander(GoalTrans,FactTrans,SpecTrans,Module,Goal,ExpandedGoal)`

- *Description:* Translates `Goal` to `ExpandedGoal` by applying user-defined translators (`GoalTrans`, `FactTrans` and `SpecTrans`) to each meta-argument present at such goal. The module where the original goal appears must be unified with `Module` argument.
- *The following properties should hold at call time:*

  `GoalTrans` is a user-defined predicate which performs *goal* meta-type translation (`expansion_tools:goal_expander/1`)

  `FactTrans` is a user-defined predicate which performs *fact* meta-type translation (`expansion_tools:fact_expander/1`)

  `SpecTrans` is a user-defined predicate which performs *spec* meta-type translation (`expansion_tools:spec_expander/1`)

  `Module` is an atom. (`basic_props:atm/1`)

  `Goal` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `ExpandedBody` is a free variable. (`term_typing:var/1`)

76.3 Documentation on internals (`expansion_tools`)

expander_pred/1: PROPERTY

Usage: `expander_pred(Pred)`

- *Description:* `Pred` is a user-defined predicate used to perform code translations. First argument will be binded to the corresponding term to be translated. Second argument must be binded to the corresponding translation. Third argument will be binded to the current module were first argument appears. Additional arguments will be user-defined.

76.4 Known bugs and planned improvements (`expansion_tools`)

- *pred(N)* meta-arguments are not supported at this moment.

77 conc_aggregates (library)

Author(s): Concurrent-safe (and incomplete) version of the aggregates predicates, based on the regular versions by Richard A. O'Keefe and David H.D. Warren. Concurrency-safeness provided by Manuel Carro..

This module implements thread-safe aggregation predicates. Its use and results should be the same as those in the aggregates library, but several goals can use them concurrently without the interference and wrong results (due to implementation reasons) aggregates might lead to. This particular implementation is completely based on the one used in the aggregates library.

77.1 Usage and interface (`conc_aggregates`)

- **Library usage:**
  `:- use_module(library(conc_aggregates)).`
- **Exports:**
  - *Predicates:*
    `findall/3`.

77.2 Documentation on exports (`conc_aggregates`)

findall/3: PREDICATE

Meta-predicate with arguments: `findall(?,goal,?)`.

Usage: `findall(?(Template),+Generator,?(List))` ⟨• ISO •⟩

- *Description:* A special case of bagof, where all free variables in the `Generator` are taken to be existentially quantified. Safe in concurrent applications.

77.3 Known bugs and planned improvements (`conc_aggregates`)

- Thread-safe `setof/3` is not yet implemented.
- Thread-safe `bagof/3` is not yet implemented.
- Incomplete, not all predicates rewritten.

78 davinci (library)

Author(s): Francisco Bueno.

This library allows connecting a Ciao Prolog application with daVinci V2.X.

The communication is based on a two-way channel: after daVinci is started, messages are sent in to it and read in from it on demand by different Prolog predicates. Messages are sent via writing the term as text; messages are received by reading text and returning an atom. Commands sent and answers received are treated as terms from the Prolog side, since for daVinci they are text but have term syntax; the only difficulty lies in strings, for which special Prolog syntax is provided.

See accompanying file `library('davinci/commands')` for examples on the use of this library.

daVinci is developed by U. of Bremen, Germany.

78.1 Usage and interface (davinci)

- **Library usage:**
  `:- use_module(library(davinci)).`
- **Exports:**
  - *Predicates:*
    `davinci/0`, `topd/0`, `davinci_get/1`, `davinci_get_all/1`, `davinci_put/1`, `davinci_quit/0`, `davinci_ugraph/1`, `davinci_lgraph/1`, `ugraph2term/2`, `formatting/2`.
- **Other modules used:**
  - *System library modules:*
    `aggregates`, `prompt`, `errhandle`, `format`, `read`, `graphs/ugraphs`, `write`, `system`.

78.2 Documentation on exports (davinci)

davinci/0: PREDICATE

No further documentation available for this predicate.

topd/0: PREDICATE

No further documentation available for this predicate.

davinci_get/1: PREDICATE

No further documentation available for this predicate.

davinci_get_all/1: PREDICATE

No further documentation available for this predicate.

davinci_put/1: PREDICATE

No further documentation available for this predicate.

davinci_quit/0: PREDICATE

No further documentation available for this predicate.

davinci_ugraph/1: PREDICATE

No further documentation available for this predicate.

davinci_lgraph/1: PREDICATE

No further documentation available for this predicate.

ugraph2term/2: PREDICATE

No further documentation available for this predicate.

formatting/2: PREDICATE

No further documentation available for this predicate.

PART VII - Ciao Prolog extensions

The libraries documented in this part extend the Ciao language in several different ways. The extensions include:

- pure Prolog programming (well, this can be viewed more as a restriction than an extension);
- feature terms or *records* (i.e., structures with names for each field);
- parallel programming (e.g., &-Prolog style);
- functional syntax;
- higher-order library;
- global variables;
- `setarg` and `undo`;
- delaying predicate execution;
- active modules;
- breadth-first execution;
- iterative deepening-based execution;
- constraint logic programming;
- object oriented programming.

79 Pure Prolog package

This library package allows the use of *pure Prolog* in a Ciao module/program. It is based on the fact that if an *engine module* is imported explicitly then all of them have to be imported explicitly. The engine modules are:

- `engine(arithmetic)`
  Chapter 20 [Arithmetic], page 105.
- `engine(atomic_basic)`
  Chapter 19 [Basic predicates handling names of constants], page 101.
- `engine(attributes)`
  Chapter 29 [Attributed variables], page 153.
- `engine(basic_props)`
  Chapter 15 [Basic data types and properties], page 85.
- `engine(basiccontrol)`
  Chapter 13 [Control constructs/predicates], page 79.
- `engine(concurrency)`
  Chapter 26 [Low-level concurrency/multithreading primitives], page 137.
- `engine(data_facts)`
  Chapter 24 [Fast/concurrent update of facts], page 125.
- `engine(exceptions)`
  Chapter 23 [Exception handling], page 123.
- `engine(io_aux)`
  Chapter 28 [Message printing primitives], page 149.
- `engine(io_basic)`
  Chapter 21 [Basic input/output], page 109.
- `engine(prolog_flags)`
  Chapter 25 [Changing system behaviour and various flags], page 131.
- `engine(streams_basic)`
  Chapter 22 [Basic file/stream handling], page 115.
- `engine(system_info)`
  Chapter 30 [Gathering some basic internal info], page 155.
- `engine(term_basic)`
  Chapter 17 [Basic term manipulation], page 95.
- `engine(term_compare)`
  Chapter 18 [Comparing terms], page 97.
- `engine(term_typing)`
  Chapter 16 [Extra-logical properties for typing], page 91.

Note that if any of these modules is explicitely imported in a program then the language defaults to Pure Prolog, plus the functionality added by the modules explicitely imported.

It is recommended that if you explicitely import an engine module you also use this package, which will guarantee that the predicate `true/0` is defined (note that this is the only Ciao builtin which cannot be redefined).

79.1 Usage and interface (pure)

- **Library usage:**
  `:- use_package(pure).`
  or
  `:- module(...,...,[pure]).`

80 argnames (library)

80.1 Usage and interface (argnames)

- **Library usage:**

  `:- use_package(argnames).`

  or

  `:- module(...,...,[argnames]).`
- **New operators defined:**

  `$/2` [150,xfx], `=>/2` [950,xfx], `argnames/1` [1150,fx].
- **Other modules used:**
  - *System library modules:*

    `aggregates`, `dynamic`, `iso_misc`, `iso_byte_char`, `iso_incomplete`, `operators`, `read`, `streams`, `write`.

81 Independent and-parallel execution

Author(s): Manuel Hermenegildo, Manuel Carro.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.3#120 (1999/11/26, 12:5:17 MET)

This library allows and-parallel execution of goals in (Herbrand-)independent fashion. It resembles the execution rules of &-Prolog [HG90]. Basically, goals are run in and-parallel *provided that their arguments do not share bindings*, i.e., are not bound to terms which contain a common variable.

Beware: it is highly unstable at the moment!

81.1 Usage and interface (`andprolog`)

- **Library usage:**
  `:- use_module(library(andprolog)).`
- **Exports:**
  - *Predicates:*
    `&/2`, `=>/2`, `active_agents/1`.
  - *Properties:*
    `indep/1`, `indep/2`.
- **Other modules used:**
  - *System library modules:*
    `assertions/doc_props`.

81.2 Documentation on exports (`andprolog`)

&/2: PREDICATE

`&(GoalA,GoalB)`

`GoalA` and `GoalB` are run in independent and-parallel fashion. This is just a first sketch, and valid only for deterministic independent goals.

Meta-predicate with arguments: `&(goal,goal)`.

=>/2: PREDICATE

`=>(A,B)`

If `A` is true `B` is run in parallel, otherwise sequentially.

Meta-predicate with arguments: `=>(goal,goal)`.

Usage: `=>(A,B)`

- *The following properties should hold at call time:*
  `B` is of the form `goal&goal`. (`andprolog:andcallable/1`)
- *The following properties hold globally:*
  Documentation is still incomplete: `=>(A,B)` may not conform the functionality documented. (`doc_props:doc_incomplete/1`)

indep/1: PROPERTY

`indep(X)`

The variables in each pair of the list `X` are pairwise independent.

Usage: `indep(X)`

– *Description:* The variables in pairs in `X` are pairwise independent.

indep/2: PROPERTY

`indep(X,Y)`

`X` and `Y` are *independent*, i.e., they are bound to terms which have no variables in common. For example, `indep(X,Y)` holds for `X=f(Z),Y=g(K)` and also for `X=f(a),Y=X` (since both `X` and `Y` are bound to ground terms). It does not hold for `X=f(Z),Y=g(Z)` and for `X=Y`.

Usage: `indep(X,Y)`

– *Description:* `X` and `Y` do not have variables in common.

active_agents/1: PREDICATE

Usage: `active_agents(?(NumberOfAgents))`

– *Description:* Tests/sets the number of active agents looking for goals to execute. As for now, those agents are resource-consuming, even when they are just looking for work, and not executing any user goals.

– *The following properties should hold at call time:*

`?(NumberOfAgents)` is an integer. (`basic_props:int/1`)

81.3 Known bugs and planned improvements (andprolog)

- The fact that only the first solution is returned for the conjunction is due to performance matters, and we expect to remove it in a near future.
- CGEs (i.e., `=>`) are not supported.
- The `indep/1`, `indep/2`, and `ground/1` are not very efficient; they might be replaced by native versions in a future.

82 functions (library)

82.1 Usage and interface (`functions`)

- **Library usage:**
  `:- use_package(functions).`
  or
  `:- module(...,...,[functions]).`
- **New operators defined:**
  `function/1` [1150,fx], `:=/2` [800,xfx], `~/1` [50,fx].
- **Other modules used:**
  - *System library modules:*
    `aggregates`, `dynamic`, `iso_misc`, `iso_byte_char`, `iso_incomplete`, `operators`, `read`, `streams`, `write`.

83 Some Higher-Order Predicates

Author(s): Daniel Cabeza.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.5#65 (2000/3/15, 22:9:28 CET)

This library implements a few basic higher-order predicates. These add functionality to the basic higher-order functionality of Ciao. Examples of the latter are:

Using pred(1):

```
list(L, between(1,6))
list(L, functor(_,2))
list(L, >(0))
```

Using pred(2):

83.1 Usage and interface (hiord)

- **Library usage:**
  `:- use_module(library(hiord)).`
- **Exports:**
  - *Predicates:*
    `map/3`, `foldl/4`.

83.2 Documentation on exports (hiord)

map/3: PREDICATE

Meta-predicate with arguments: `map(?,pred(2),?)`.

Usage: `map(LList,Op,RList)`

- *Description:* Examples of use:

```
map([1,3,2], arg(f(a,b,c,d)), [a,c,b]) or
map([1,3,2], nth([a,b,c,d]), [a,c,b])
map(["D","C"], append("."), ["D.","C."])
```

foldl/4: PREDICATE

Meta-predicate with arguments: `foldl(?,?,pred(3),?)`.

Usage: `foldl(List,Seed,Op,Result)`

- *Description:* Example of use:

```
?- foldl(["daniel","cabeza","gras"], "",
         (:(X,Y,Z) :- append(X, " "||Y, Z)), R).

R = "daniel cabeza gras " ?
```

84 global (library)

Version: 0.4#5 (1998/2/24)

84.1 Usage and interface (`global`)

- **Library usage:**
  `:- use_module(library(global)).`
- **Exports:**
  - *Predicates:*
    `set_global/2`, `get_global/2`, `push_global/2`, `pop_global/2`, `del_global/1`.

84.2 Documentation on exports (`global`)

set_global/2: PREDICATE
No further documentation available for this predicate.

get_global/2: PREDICATE
No further documentation available for this predicate.

push_global/2: PREDICATE
No further documentation available for this predicate.

pop_global/2: PREDICATE
No further documentation available for this predicate.

del_global/1: PREDICATE
No further documentation available for this predicate.

85 Miscellaneous predicates

Author(s): Manuel Carro, Daniel Cabeza.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.3#95 (1999/11/8, 18:37:30 MET)

This module implements some miscellaneous non-logical (but sometimes very useful) predicates.

85.1 Usage and interface (odd)

- **Library usage:**
  `:- use_module(library(odd)).`
- **Exports:**
  - *Predicates:*
    `setarg/3`, `undo/1`.

85.2 Documentation on exports (odd)

setarg/3: PREDICATE

Usage: `setarg(Index,Term,NewArg)`

- *Description:* Replace destructively argument `Index` in `Term` by `NewArg`. The assignment is undone on backtracking. This is a major change to the normal behavior of data assignment in Ciao Prolog.
- *The following properties should hold at call time:*
  `Index` is currently instantiated to an integer. (`term_typing:integer/1`)
  `Term` is a compound term. (`basic_props:struct/1`)
  `NewArg` is any term. (`basic_props:term/1`)
- *The following properties hold upon exit:*
  `Index` is currently instantiated to an integer. (`term_typing:integer/1`)
  `Term` is a compound term. (`basic_props:struct/1`)
  `NewArg` is any term. (`basic_props:term/1`)

undo/1: PREDICATE

Usage: `undo(Goal)`

- *Description:* `call(Goal)` is executed on backtracking. This is a major change to the normal control of Ciao Prolog execution.
- *The following properties should hold at call time:*
  `Goal` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)
- *The following properties hold upon exit:*
  `Goal` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

86 Delaying predicates (freeze)

Author(s): Manuel Carro, Daniel Cabeza.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.5#72 (2000/3/19, 19:9:14 CET)

This library offers a simple implementation of `freeze/2`, `frozen/2`, etc. [Col82,Nai85,Nai91,Car87] based on the use of attributed variables [Hol92,Hol90].

86.1 Usage and interface (`freeze`)

- **Library usage:**
  `:- use_module(library(freeze)).`
- **Exports:**
  - *Predicates:*
    `freeze/2`, `frozen/2`.
  - *Multifiles:*
    `verify_attribute/2`, `combine_attributes/2`.

86.2 Documentation on exports (`freeze`)

freeze/2: PREDICATE

Meta-predicate with arguments: `freeze(?,goal)`.

Usage: `freeze(X,Goal)`

- *Description:* If `X` is free delay `Goal` until `X` is non-variable.
- *The following properties should hold at call time:*
  `Goal` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

frozen/2: PREDICATE

Meta-predicate with arguments: `frozen(?,goal)`.

Usage: `frozen(X,Goal)`

- *Description:* `Goal` is currently delayed until variable `X` becomes bound.
- *The following properties should hold upon exit:*
  `Goal` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

86.3 Documentation on multifiles (`freeze`)

verify_attribute/2: PREDICATE

No further documentation available for this predicate.

The predicate is *multifile*.

combine_attributes/2: PREDICATE

No further documentation available for this predicate.

The predicate is *multifile*.

87 actmods (library)

87.1 Usage and interface (actmods)

- **Library usage:**
  `:- use_package(actmods).`
  or
  `:- module(...,...,[actmods]).`
- **Other modules used:**
  - *System library modules:*
    `actmods/actmodrt`, `aggregates`, `dynamic`, `iso_misc`, `iso_byte_char`, `iso_incomplete`, `operators`, `read`, `streams`, `write`.

88 Breadth-first execution

Author(s): Daniel Cabeza, Manuel Carro.

This package implements breadth-first execution of predicates. Predicates written with operators '<-'/1 (facts) and '<-'/2 (clauses) are executed using breadth-first search. This may be useful in search problems when a complete proof procedure is needed. An example of code would be:

```
:- module(bf_example, _, [bf]).

test(bf) :- bfchain(a,d).
test(df) :- chain(a,d).   % loops!

bfchain(X,X) <- .
bfchain(X,Y) <- arc(X,Z), bfchain(Z,Y).

chain(X,X).
chain(X,Y) :- arc(X,Z), chain(Z,Y).

arc(a,b).
arc(a,d).
arc(b,c).
arc(c,a).
```

88.1 Usage and interface (bf)

- **Library usage:**
  `:- use_package(bf).`
  or
  `:- module(...,...,[bf]).`
- **New operators defined:**
  `<-/2` [1200,xfx], `<-/1` [1200,xf].

88.2 Known bugs and planned improvements (bf)

- Does not correctly work in user files.

89 Iterative-deepening execution

Author(s): Claudio Vaucheret, Manuel Hermenegildo.

This package applies a *compiling control* technique to implement *depth first iterative deepening* execution [BibRef: iterative-deepening]. It changes the usual *depth-first* computation rule by *iterative-deepening* on those predicates specifically marked. This is very useful in search problems when a complete proof procedure is needed.

When this computation rule is used, first all goals are expanded only up to a given depth. If no solution is found or more solutions are needed by backtracking, the depth limit is incremented and the whole goal is repeated. Although it might seem that this approach is very inefficient because all higher levels are repeated for the deeper ones, it has been shown that is performs only about b/(b - 1) times as many operations than the corresponding breadth-first search, (where b is the branching factor of the proof tree) while the waste of memory is the same as depth first.

The usage is by means of the following directive:

`:- iterative(Name, FirstCut, Formula).`

which states than the predicate 'Name' given in functor/arity form will be executed using iterative deepening rule starting at the depth 'FirstCut' with depth being incremented by the predicate 'Formula'. This predicate compute the new depth using the previous one. It must implement a dilating function i.e. the new depth must be greater. For example, to start with depth 5 and increment by 10 you can write:

`:- iterative(p/1,5,f).`

`f(X,Y) :- Y is X + 10.`

or if you prefer,

`:- iterative(p/1,5,(_(X,Y):- Y is X + 10)).`

You can also use a fourth parameter to set a limiting depth. All goals below the given depth limit simply fail. Thus, with the following directive:

`:- iterative(p/1,5,(_(X,Y):- Y is X + 10),100).`

all goals deeper than 100 will fail.

An example of code using this package would be:

```
:- module(example_id, _,[id]).

test(id) :-
        idchain(a,d).
test(df) :-
        chain(a,d).   % loops!

:- iterative(idchain/2, 3, ( _(X,Z) :- Z is X + 1) ).

idchain(X,X).
idchain(X,Y) :-
        arc(X,Z),
        idchain(Z,Y).

chain(X,X).
chain(X,Y) :-
        arc(X,Z),
        chain(Z,Y).

arc(a,b).
```

```
arc(a,d).
arc(b,c).
arc(c,a).
```

The order of solutions are first the shallower and then the deeper. Solutions which are between two cutoff are given in the usual left to right order. For example,

```
:- module(_,_,[id]).

:- iterative(p/1,0,(_(X,Z) :- Z is X + 1),2).

%:- iterative(p/1,1,(_(X,Z) :- Z is X + 3)). % change the solutions' order
                                             % to goal p(X).

p(X) :- q(X).
p(a).

q(X) :- r(X).
q(b).

r(X) :- s(X).
r(c).

s(d).
```

It is possible to preserve the iterative-deepening behavior for calls to predicates defined in other modules. These modules should obviously also use this package. In addition *all* predicates from such modules should imported, i.e., the directive `:- use_module(module)`, should be used in this case instead of `:- use_module(module,[...])`. Otherwise calls to predicates outside the module will be treated in the usual way i.e. by depth-first computation.

Another complete proof procedure implemented is the `bf` package (breadth first execution).

89.1 Usage and interface (`id`)

- **Library usage:**
  `:- use_package(id).`
  or
  `:- module(...,...,[id]).`

90 Constraint programming over rationals

Author(s): Christian Holzbaur, Daniel Cabeza.

Note: This package is currently being adapted to the new characteristics of the Ciao module system. This new version now works right now to some extent, but it is under further development at the moment. Use with (lots of) caution.

90.1 Usage and interface (clpq)

- **Library usage:**
  `:- use_package(clpq).`
  or
  `:- module(...,...,[clpq]).`

90.2 Other information (clpq)

90.2.1 Some CLP(Q) examples

(Other examples can be found in the source and library directories.)

- 'Reversible' Fibonacci (clpq):

```
:- use_package(clpq).

fib(X,Y):- X .=. 0, Y .=. 0.
fib(X,Y):- X .=. 1, Y .=. 1.
fib(N,F) :-
        N .>. 1,
        N1 .=. N - 1,
        N2 .=. N - 2,
        fib(N1, F1),
        fib(N2, F2),
        F .=. F1+F2.
```

- Matrix multiplication (clpq):

```
:- use_package(clpq).

mmultiply([],_,[]).
mmultiply([V0|Rest], V1, [Result|Others]):-
            mmultiply(Rest, V1, Others),
                multiply(V1,V0,Result).

multiply([],_,[]).
multiply([V0|Rest], V1, [Result|Others]):-
            multiply(Rest, V1, Others),
                vmul(V0,V1,Result).

vmul([],[],0).
```

```
vmul([H1|T1], [H2|T2], Result):-
        vmul(T1,T2, Newresult),
        Result .=. H1*H2+Newresult.

matrix(1,[[1,2,3,4,5],[4,0,-1,5,6],[7,1,-2,8,9],[-1,0,1,3,2],[1,5,-3,2,4]]).
matrix(2,[[3,2,1,0,-1],[-2,1,3,0,2],[1,2,0,-1,5],[1,3,2,4,5],[-5,1,4,2,2]]).
```

- Queens (clpq):

```
:- use_package(clpq).

queens(N, Qs) :- constrain_values(N, N, Qs), place_queens(N, Qs).

constrain_values(0, _N, []).
constrain_values(N, Range, [X|Xs]) :-
        N .>. 0, X .>. 0, X .=<. Range,
        N1 .=. N - 1,
        constrain_values(N1, Range, Xs), no_attack(Xs, X, 1).

no_attack([], _Queen, _Nb).
no_attack([Y|Ys], Queen, Nb) :-
        abs(Queen - (Y + Nb)) .>. 0,  % Queen =\= Y + Nb
        abs(Queen - (Y - Nb)) .>. 0,  % Queen =\= Y - Nb
        Nb1 .=. Nb + 1,
        no_attack(Ys, Queen, Nb1).

place_queens(0, _).
place_queens(N, Q) :-
        N .>. 0, member(N, Q), N1 .=. N - 1, place_queens(N1, Q).

member(X, [X|_]).
member(X, [_|Xs]) :- member(X, Xs).
```

91 Constraint programming over reals

Author(s): Christian Holzbaur, Daniel Cabeza.

Note: This package is currently being adapted to the new characteristics of the Ciao module system. This new version now works right now to some extent, but it under further development at the moment. Use with (lots of) caution.

91.1 Usage and interface (clpr)

- **Library usage:**
  `:- use_package(clpr).`
  or
  `:- module(...,...,[clpr]).`

91.2 Other information (clpr)

91.2.1 Some CLP(R) examples

(Other examples can be found in the source and library directories.)

- 'Reversible' Fibonacci (clpr):

```
:- use_package(clpr).

fib(X,Y):- X .=. 0, Y .=. 0.
fib(X,Y):- X .=. 1, Y .=. 1.
fib(N,F) :-
        N .>. 1,
        N1 .=. N - 1,
        N2 .=. N - 2,
        fib(N1, F1),
        fib(N2, F2),
        F .=. F1+F2.
```

- Dirichlet problem for Laplace's equation (clpr):

```
%
% Solve the Dirichlet problem for Laplace's equation using
% Leibman's five-point finit-differenc approximation.
% The goal ?- go1 is a normal example, while the goal ?- go2
% shows output constraints for a small region where the boundary conditions
% are not specified.
%
:- use_package(clpr).
:- use_module(library(write)).

laplace([_, _]).
laplace([H1, H2, H3|T]):-
        laplace_vec(H1, H2, H3),
```

```
        laplace([H2, H3|T]).

laplace_vec([_, _], [_, _], [_, _]).
laplace_vec([_TL, T, TR|T1], [ML, M, MR|T2], [_BL, B, BR|T3]):-
        B + T + ML + MR - 4 * M .=. 0,
        laplace_vec([T, TR|T1], [M, MR|T2], [B, BR|T3]).

printmat([]).
printmat([H|T]):-
        printvec(H),
        printmat(T).

printvec([]):- nl.
printvec([H|T]):-
        Hdd .=. integer(H*100),
        print(Hdd),
        printvec(T).

go1:-
        X =  [
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [100, _, _, _, _, _, _, _, _, _, 100],
    [100, _, _, _, _, _, _, _, _, _, 100],
    [100, _, _, _, _, _, _, _, _, _, 100],
    [100, _, _, _, _, _, _, _, _, _, 100],
    [100, _, _, _, _, _, _, _, _, _, 100],
    [100, _, _, _, _, _, _, _, _, _, 100],
    [100, _, _, _, _, _, _, _, _, _, 100],
    [100, _, _, _, _, _, _, _, _, _, 100],
    [100, _, _, _, _, _, _, _, _, _, 100],
    [100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100]
    ],
        laplace(X),
        printmat(X).

% Answer:
%   0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00   0.00
% 100.00 51.11 32.52 24.56 21.11 20.12 21.11 24.56 32.52 51.11 100.00
% 100.00 71.91 54.41 44.63 39.74 38.26 39.74 44.63 54.41 71.91 100.00
% 100.00 82.12 68.59 59.80 54.97 53.44 54.97 59.80 68.59 82.12 100.00
% 100.00 87.97 78.03 71.00 66.90 65.56 66.90 71.00 78.03 87.97 100.00
% 100.00 91.71 84.58 79.28 76.07 75.00 76.07 79.28 84.58 91.71 100.00
% 100.00 94.30 89.29 85.47 83.10 82.30 83.10 85.47 89.29 94.30 100.00
% 100.00 96.20 92.82 90.20 88.56 88.00 88.56 90.20 92.82 96.20 100.00
% 100.00 97.67 95.59 93.96 92.93 92.58 92.93 93.96 95.59 97.67 100.00
% 100.00 98.89 97.90 97.12 96.63 96.46 96.63 97.12 97.90 98.89 100.00
% 100.00100.00100.00100.00100.00100.00100.00100.00100.00100.00 100.00

go2 :-
        laplace([
            [_B11, B12, B13, _B14],
```

```
                [B21, M22, M23, B24],
                [B31, M32, M33, B34],
                [_B41, B42, B43, _B44]
        ]),
            dump([B31, M32, M33, B34, B42, B43, B12, B13, B21, M22, M23, B24]).

% Answer:
%  B12 =  -B21 - 4*B31 + 16*M32 - 8*M33 + B34 - 4*B42 + B43
%  B13 =  -B24 + B31 - 8*M32 + 16*M33 - 4*B34 + B42 - 4*B43
%  M22 =  -B31 + 4*M32 - M33 - B42
%  M23 =  -M32 + 4*M33 - B34 - B43
```

92 Object oriented programming

Author(s): Angel Fernandez Pineda.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.3#63 (1999/9/29, 19:54:17 MEST)

O'Ciao is a set of libraries which allows object-oriented programming in Ciao Prolog. It extends the Ciao Prolog module system by introducing two new concepts:

- Inheritance.
- Instantiation.

Polymorphism is the third fundamental concept provided by object oriented programming. This concept is not mentioned here since **traditional PROLOG systems are polymorphic by nature**.

Classes are declared in the same way as modules. However, they may be enriched with inheritance declarations and other object-oriented constructs. For an overview of the fundamentals of O'Ciao, see `http://www.clip.dia.fi.upm.es/~clip/papers/ociao-tr.ps.gz`. However, we will introduce the concepts in a tutorial way via examples.

92.1 Early examples

The following one is a very simple example which declares a class – a simple stack. Note that if you replace *class/1* declaration with a *module/1* declaration, it will compile correctly, and can be used as a normal Prolog module.

```
%%--------------------------------------------%%
%% class/1, at the beginning of the file,     %%
%% will declare current source to be a class. %%
%%--------------------------------------------%%

:- class(stack,[],[]).

% State declaration: storage/1 will be an attribute.
:- dynamic storage/1.

% Interface declaration: the following predicates will
% be available at run-time.

:- export(push/1).
:- export(pop/1).
:- export(top/1).
:- export(is_empty/0).

% Code

push(Item) :-
        nonvar(Item),
        asserta_fact(storage(Item)).

pop(Item) :-
        var(Item),
        retract_fact(storage(Item)).
```

```
top(Top) :-
        storage(Top),
        !.

is_empty :-
        storage(_),
        !,
        fail.

is_empty.

:- comment(version_maintenance,off).
```

If we load this code at the Ciao toplevel shell:

```
?- use_package(objects).

yes
?- use_class(library('class/examples/stack')).

yes
?-
```

we can create two stack *instances* :

```
?- St1 new stack,St2 new stack.

St1 = stack('9254074093385163'),
St2 = stack('9254074091') ? ,
```

and then, we can operate on them separately:

```
1 ?- St1:push(8),St2:push(9).

St1 = stack('9254074093385163'),
St2 = stack('9254074091') ?

yes
1 ?- St1:top(I),St2:top(K).

I = 8,
K = 9,
St1 = stack('9254074093385163'),
St2 = stack('9254074091') ?

yes
1 ?-
```

The interesting point is that there are two stacks. If the previous example had been a normal module, we would have a stack , but **only one** stack.

The next example introduces the concepts of *inheritable* predicate, *constructor*, *destructor* and *virtual method*. Refer to the following sections for further explanation.

```
%%----------------------------------%%
%% A generic class to perform       %%
%% item storage.                    %%
```

```
%%----------------------------------%%
:- class(generic).

% Public interface declaration:
:- export([set/1,get/1,callme/0]).

% An attribute
:- data datum/1.

% inheritable/1 is used for datum/1
% to become available to descendant
% classes (if any).
:- inheritable(datum/1).

% Attribute initialization. Attributes
% are easily initialized by writing clauses
% for them.

datum(none).

% Code

set(X) :-
        type_check(X),
        set_fact(datum(X)).

get(X) :-
        datum(X).

:- virtual type_check/1.

type_check(X) :-
        nonvar(X).

callme :-
        a_virtual(IMPL),
        display(IMPL),
        display(' implementation of a_virtual/0 '),
        nl.

% virtual/1 will tell the compiler to use the most
% descendant implementation of a_virtual/1 when calling
% it from inside this code (for example,see callme/0).
% If there is no descendant implementation for it,
% the one defined bellow will be used.

:- virtual a_virtual/1.
a_virtual(generic).

% Constructor: in this case, every time an instance
% of this class is created, it will display a message.
```

```
generic :-
        display(' generic class constructor '),
        nl.

% Destructor: analogous to the previous constructor,
% it will display a message every time an instance
% of this class is eliminated.

destructor :-
        display(' generic class destructor '),
        nl.
```

And the following example, is an extension of previous class. This is performed by establishing an inheritance relationship:

```
%%----------------------------------%%
%% This class provides additional   %%
%% functionality to the generic     %%
%% class.                           %%
%%----------------------------------%%

:- class(specific).

% establish an inheritance relationship with "generic" class.
:- inherit_class(library('class/examples/generic')).

% override inherited datum/1.
% datum/1 is said to be overriden because there are both
% an inherited definition (from "generic" class) and a local one.

:- data datum/1.
:- inheritable datum/1.

% Extend the public interface inherited from "generic".
% note that set/1 and a_virtual/0 has been also overriden.
% undo/0 is a new added functionality.

:- export([set/1,undo/0]).

% Code

set(Value) :-
        inherited datum(OldValue),
        !,
        inherited set(Value),
        asserta_fact(datum(OldValue)).


set(Value) :-
        inherited set(Value).
```

```
undo :-
        retract_fact(datum(Last)),
        !,
        asserta_fact(inherited(datum(Last))).

undo :-
        retractall_fact(inherited(datum(_))).

% new implementation of a_virtual. This
% implementation will be used from inherited callme/0
% instead of that version defined at "generic".

a_virtual(specific).

% constructor

specific :-
        generic,
        retractall_fact(inherited(datum(_))),
        display(' specific class constructor '),
        nl.

% destructor

destructor :-
        display(' specific class destructor '),
        nl.
```

Additional examples may be found on the *library/class/examples* directory relative to your Ciao Prolog instalation.

92.2 Recommendations on when to use objects

We would like to give some advice in the use of object oriented programming, in conjunction with the declarative paradigm.

You should reconsider using O'Ciao in the following cases:

- The pretended "objects" have no state,i.e., no data or dynamic predicates. In this case, a normal module will suffice.
- There is state, but there will be only one instance of a pretended class. Again, a module suffices.
- The "objects" are data structures (list,trees,etc) already supported by Prolog. However, it does make sense to model, using objects, data structures whose change implies a side-effect such as drawing a particular window on the screen.

We recommend the usage of O'Ciao in the following cases:

- You feel you will need to have several copies of a "module".
- Local copies of a module are needed instead of a global module beeing modified by several ones.
- The "classes" are a representation of external entities to Prolog. For example: the X-Window system.

- There is state or code outside the Prolog system which needs to be manipulated. For example: interfaces to Java or Tcl/Tk code.
- You are not familiar with Prolog, but you know about object oriented programming. O'Ciao may be used as a learning tool to introduce yourself on the declarative programming paradigm.

92.3 Limitations on object usage

O'Ciao run-time speed is limited by the usage of meta-programming structures, for instance: `X = (Object:mymethod(25)), call(X)`. O'Ciao will optimize static manipulation of objects (those that can be determined at compile time).

93 Declaring classes and interfaces

Author(s): Angel Fernandez Pineda.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.5#89 (2000/3/24, 14:24:48 CET)

O'Ciao classes are declared in the same way as traditional prolog modules. The general mechanism of *source expansion* will translate object-oriented declarations to normal prolog code. This is done transparently to the user.

Abstract *interfaces* are restricted classes which declare exported predicates with no implementation. The implementation itselt will be provided by some class using an `implements/1` declaration. Only `export/1` and `data/1` declarations are allowed when declaring an interface. Normal classes may treated as interfaces just ignoring all exported predicate implementations.

93.1 Usage and interface (`class`)

- **Library usage:**

  To declare a class the compiler must be told to use the `class` *source expansion*. To do so, source code must start with a module declaration which loads the class package:

  ```
  :- class(ClassName).
  ```

  or a `module/3` declaration, as follows:

  ```
  :- module(ClassName,[],[class]).
  ```

  interfaces are declared in a similar way:

  ```
  :- interface(InterfaceName).
  ```

  Please, do not use SICStus-like module declaration, with a non-empty export list. In other case, some non-sense errors will be reported by normal Ciao module system.

  Most of the regular Ciao declarations may be used when defining a class, such as `concurrent/1`, `dynamic/1`, `discontiguous/1`, `multifile/1`, and so on.

  However, there are some restrictions wich apply to those declarations:

  - `meta_predicate/1` declaration is not allowed to hold *addmodule and pred(N) meta-arguments*, except for previously declared multifiles.
  - Attribute and multifile predicates must be declared before any clause of the related predicate.
  - There is no sense in declaring an attribute as meta_predicate.

  It is a good practique to put all your declarations at the very begining of the file, just before the code itself.

- **Exports:**
  - *Predicates:*

    `inherited/1`, `self/1`, `constructor/0`, `destructor/0`.

- **New declarations defined:**

  `export/1`, `public/1`, `inheritable/1`, `data/1`, `dynamic/1`, `concurrent/1`, `inherit_class/1`, `implements/1`, `virtual/1`.

- **Other modules used:**
  - *System library modules:*

    `objects/objects_rt`.

93.2 Documentation on new declarations (`class`)

export/1: DECLARATION

Declares a method or attribute to be part of the *public interface*.

The public interface is the set of predicates wich will be accesible from any code establishing an usage relationship with this class (see `use_class/1` for further information).

Publishing an attribute or method is very similar to *exporting* a predicate in a Prolog module.

Whether an inherited and exported predicate is overriden, it must be explicitly exported again.

An inherited (but not exported) predicate may become exported, without overriding it by the usage of this declaration.

Usage: :- `export(Spec)`.

- *Description:* `Spec` will be part of the public (exported) interface.
- *The following properties should hold at call time:*
  `Spec` is a method or attribute specification. (`objects_rt:method_spec/1`)

public/1: DECLARATION

Just an alias for `export/1`.

Usage: :- `public(Spec)`.

- *Description:* This declaration may be used instead of `export/1`.
- *The following properties should hold at call time:*
  `Spec` is a method or attribute specification. (`objects_rt:method_spec/1`)

inheritable/1: DECLARATION

Declares a method or attribute to be inherited by descendant classes. Notice that all **public predicates are inheritable by default**. There is no need to mark them as inheritable.

Traditionaly, object oriented languages makes use of the *protected* concept. Inheritable/1 may be used as the same concept.

The set of inheritable predicates is called the *inheritable interface*.

Usage: :- `inheritable(MethodSpec)`.

- *Description:* `MethodSpec` is accessible to descendant classes.
- *The following properties should hold at call time:*
  `MethodSpec` is a method or attribute specification. (`objects_rt:method_spec/1`)

data/1: DECLARATION

Declares an *attribute* at current class. Attributes are used to build the internal state of instances. So, each instance will own a particular copy of those attribute definitions. In this way, one instance may have different state from another.

O'Ciao attributes are restricted to hold simple facts. It is not possible to hold a Head :- Body clause at an instance attribute.

Notice that attributes are *multi-evaluated* by nature, and may be manipulated by the habitual **assert/retract** family of predicates.

Attributes may also be initialized. In order to do so, simply put some clauses after the attribute definition. Each time an instance is created, its initial state will be built from those *initialization clauses.*

Note: whether a data/1 declaration appears inside an interface, it will be automatically exported.

Usage: :- `data Spec`.

- *Description:* `Spec` is an attribute.
- *The following properties should hold at call time:*
  `Spec` is a method or attribute specification. (`objects_rt:method_spec/1`)

dynamic/1: DECLARATION

Just an alias for `data/1`.

Usage: :- `dynamic Spec`.

- *Description:* You may use this declaration instead of `data/1`.
- *The following properties should hold at call time:*
  `Spec` is a method or attribute specification. (`objects_rt:method_spec/1`)

concurrent/1: DECLARATION

Declares a *concurrent attribute* at current class. Concurrent attributes are just the same as normal attributes, those declared using `data/1`, except for they may freeze the calling thread instead of failing when no more choice points are remaining on the concurrent attribute.

In order to get more information about concurrent behavior take a look to the concurrent/1 built-in declaration on Ciao Prolog module system.

Usage: :- `concurrent Spec`.

- *Description:* Declares `Spec` to be a concurrent attribute.
- *The following properties should hold at call time:*
  `Spec` is a method or attribute specification. (`objects_rt:method_spec/1`)

inherit_class/1: DECLARATION

Makes any public and/or inheritable predicate at inherited class to become accesible by any instance derived from current class.

Inherited class is also called the *super class.*

Only one inherit_class/1 declaration is allowed to be present at current source.

Notice that inheritance is public by default. Any public and/or inheritable declaration will remain the same to descendant classes. However, any inherited predicate may be *overriden* (redefined).

A predicate is said to be *overriden* when it has been inherited from super class, but there are clauses (or a `data/1` declaration) present at current class for such a predicate.

Whether a **public** predicate is overriden, the local definition must also be exported, otherwise an error is reported.

Whether an **inheritable** predicate (not public) is overriden, the local definition must also be marked as inheritable or exported, otherwise an error is also reported.

Note: whether inherit_class/1 appears inside an interface, it will be used as an `implements/1` declaration.

Usage: :- `inherit_class(Source)`.

- *Description:* Establish an *inheritance relationship* between current class and the class defined at `Source` file.
- *The following properties should hold at call time:*

  `Source` is a valid path to a prolog file containing a class declaration (without .pl extension). (`objects_rt:class_source/1`)

implements/1: DECLARATION

Forces current source to provide an implementation for the given interface file. Such interface file may declare another class or a specific interface.

Every public predicate present at given interface file will be automatically declared as public at current source, so you **must** provide an implementation for such predicates.

The effect of this declaration is called *interface inheritance*,and there is no restriction on the number of implements/1 declarations present at current code.

Usage: :- `implements(Interface)`.

- *Description:* Current source is supposed to provide an implementation for `Interface`.
- *The following properties should hold at call time:*

  `Interface` is a valid path to a prolog file containing a class declaration or an interface declaration (without .pl extension). (`objects_rt:interface_source/1`)

virtual/1: DECLARATION

This declaration may be used whenever descendant classes are to implement different versions of a given predicate.

virtual predicates give a chance to handle, in an uniform way, different implementations of the same functionality.

Whether a virtual predicate is declared as a method, there must be at least one clause of it present at current source. Whenever no special implementation is needed at current class, a never-fail/allways-fail clause may be defined (depending on your needs). For example:

```
:- virtual([ test1/1 , test2/2 ]).
test1(_).
test2(_,_) :- fail.
```

This kind of virtual methods are also known as *abstract methods*, since implementation is fully delegated to descendant classes.

An attribute may be also declared as a virtual one, but there is no need to write clauses for it.

Usage: :- `virtual(VirtualMethodSpec)`.

- *Description:* All calls to `VirtualMethodSpec` predicate in current source will use the most descendant implementation of it.
- *The following properties should hold at call time:*

  `VirtualMethodSpec` is a method specification. (`objects_rt:virtual_method_spec/1`)

93.3 Documentation on exports (`class`)

inherited/1: PREDICATE

This predicate qualificator may be used whenever you need to reference an attribute or method on the super class.

Since methods and attributes may be redefined, this qualificator is need to distinguish between a locally declared predicate and the inherited one, which has the same name.

There is no need to use inherited/1 if a particular inherited predicate has not been redefined at current class.

Usage: `inherited(Goal)`

- *Description:* References a given `Goal` at the super class
- *The following properties should hold at call time:*

  `Goal` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

self/1: PREDICATE

Determines which instance is currently executing self/1 goal.

Predicate will fail if argument is not a free variable. Otherwise, it will allways succeed, retrieving the instance identifier which is executing current code.

This functionality is very usefull since an object must have knowledge of other object's identifier in order to send messages to it.For example:

:- concurrent ack/0.

send_data_to_object(Data,Obj) :- self(X), Obj:take_this(Data,X), current_fact(ack).

acknowledge :- asserta_fact(ack).

take_this(Data,Sender) :- validate_data(Data), Sender:acknowledge.

Usage: `self(Variable)`

- *Description:* Retrieves current instance identifier in `Variable`
- *The following properties should hold at call time:*

  `Variable` is a free variable. (`term_typing:var/1`)
- *The following properties should hold upon exit:*

  `Variable` is an unique term which identifies an object. (`objects_rt:instance_id/1`)

constructor/0: PREDICATE

A *constructor* is a special case of method which complains the following conditions:

- The constructor functor matches the current class name.
- A constructor may hold any number of arguments.
- If an inheritance relationship was defined, an inherited constructor must be manually called (see below).
- When instance creation takes place, any of the declared constructors are implicitly called. The actual constructor called depends on the `new/2` goal specified by the user.

This is a simple example of constructor declaration for the foo class:

```
foo :-
    display('an instance was born').
```

Constructor declaration is not mandatory, and there may be more than one constructor declarations (with different arity) at the source code.

This functionality is usefull when some computation is needed at instance creation. For example: opening a socket, clearing the screen, etc.

Whenever an inheritance relationship is established, and there is any constructor defined at the super class, you must call manually an inherited constructor. Here is an example:

```
:- class(foo).
:- inherit_class(myclass).

foo :-
    myclass(0),
    display('an instance was born').

foo(N) :- myclass(N).
```

Consequences may be unpredictable, if you forget to call an inherited constructor. You should also take care not to call an inherited constructor twice.

All defined constructors are inheritable by default. A constructor may also be declared as public (by the user), but it is not mandatory.

Usage:

– *Description:* Constructors are implicitly declared

destructor/0: PREDICATE

A *destructor* is a special case of method which will be automatically called when instance destruction takes place.

A destructor will never be wanted to be part of the public interface, and there is no need to mark them as inheritable, since all inherited destructors are called by O'Ciao just before yours.

This is a simple example of destructor declaration:

```
destructor :-
    display('goodbye, cruel world!!!').
```

Destructor declaration is not mandatory. Failure or sucess of destructors will be ignored by O'Ciao, and they will be called only once.

This functionality is usefull when some computation is need at instance destruction. For example: closing an open file.

Usage:

– *Description:* Destructors are implicitly declared

93.4 Other information (class)

This describes the errors reported when declaring a class or an interface. The first section will explain compile-time errors, this is, any semantic error which may be determined at compile time. The second section will explain run-time errors, this is, any exception that may be raisen by the incorrect usage of O'Ciao. Some of those errors may be not reported at compile time, due to the use of meta-programational structures. For example:

`functor(X,my_method,0),call(X).`

O'Ciao is not able to check whether my_method/0 is a valid method or not. So, this kind of checking is left to run time.

93.4.1 Class and Interface error reporting at compile time

- **ERROR : multiple inheritance not allowed.**

  There are two or more inherit_class/1 declarations found at your code. Only one declaration is allowed, since there is no multiple code inheritance support.

- **ERROR : invalid inheritance declaration.**

  The given parameter to inherit_class/1 declaration is not a valid path to a Prolog source.

- **ERROR : sorry, addmodule meta-arg is not allowed at** ***F/A*****.**

  You are trying to declare *F/A* as meta-predicate, and one of the meta-arguments is *addmodule*. This is not allowed in O'Ciao due to implementation restrictions. For example:

  `:- meta_predicate example(addmodule).`

  `example(X,FromModule) :- call(FromModule:X).`

- **ERROR : invalid attribute declaration for** ***Arg*****.**

  Argument to data/1 or dynamic/1 declaration is not a valid predicate specification of the form *Functor/Arity*. For example:

  `:- data attr.`

  `:- dynamic attr(_).`

  `:- data attr/m.`

  etc,etc...

- **ERROR : pretended attribute** ***F/A*** **was assumed to be a method.**

  You put some clauses of *F/A* before the corresponding data/1 or dynamic/1 declaration. For example:

  `attr(initial_value).`

  `:- data attr/1.`

  It is a must to declare attributes before any clause of the given predicate.

- **ERROR : destructor/0 is not allowed to be an attribute.**

  There is a :- data(destructor/0) or :- dynamic(destructor/0). declaration in your code. This is not allowed since destructor/0 is a reserved predicate, and must be allways a method.

- **ERROR :** ***Constructor*** **is not allowed to be an attribute.**

  As the previos error, you are trying to declare a constructor as an attribute. A constructor must be allways a method.

- **ERROR : invalid multifile: destructor/0 is a reserved predicate.**

  There is a :- multifile(destructor/0). declaration in your code. This is not allowed since destructor/0 is a reserved predicate, and must be allways a method.

- **ERROR : invalid multifile:** ***Constructor*** **is a reserved predicate.**

  As the previos error, you are trying to declare a constructor as a multifile. Any constructor must allways be a method.

- **ERROR : multifile declaration of** ***F/A*** **ignored: it was assumed to be a method.**

  You put some clauses of *F/A* before the corresponding multifile/1 declaration. For example:

  example(a,b).

  `:- multifile example/2.`

  Multifile predicates must be declared before any clause of the given predicate.

- **ERROR : invalid multifile declaration: multifile(*Arg*).**

  Given argument to multifile/1 declaration is not a valid predicate specification, of the form *Functor/Arity*.

- **ERROR : invalid public declaration:** ***Arg*****.**

  Given argument *Arg* to public/1 or export/1 declaration is not a valid predicate specification, of the form *Functor/Arity*.
- **ERROR : invalid inheritable declaration: inheritable(*****Arg*****).**

  Given argument *Arg* to inheritable/1 declaration is not a valid predicate specification, of the form *Functor/Arity*.
- **ERROR : destructor/0 is not allowed to be virtual.**

  There is a :- virtual(destructor/0) declaration present at your code. Destructors and/or constructors are not allowed to be virtual.
- **ERROR :** ***Constructor*** **is not allowed to be virtual.**

  As the previous error, you are trying to declare a constructor as virtual. This is not allowed.
- **ERROR : invalid virtual declaration: virtual(*****Arg*****).**

  Given argument to virtual/1 declaration is not a valid predicate specification, of the form *Functor/Arity*.
- **ERROR : clause of** ***F/A*** **ignored : only facts are allowed as initial state.**

  You declared *F/A* as an attribute, then you put some clauses of that predicate in the form *Head :- Body*. For example:

  `:- data my_attribute/1.`

  `my_attribute(X) :- X>=0 , X<=2.`

  This is not allowed since attributes are assumed to hold simple facts. The correct usage for those *initialization clauses* is:

  `:- data my_attribute/1.`

  `my_attribute(0).`

  `my_attribute(1).`

  `my_attribute(2).`
- **ERROR : multifile** ***F/A*** **is not allowed to be public.**

  The given *F/A* predicate is both present at multifile/1 and public/1 declarations. For example:

  `:- public(p/1).`

  `:- multifile(p/1).`

  This is not allowed since multifile predicates are not related to Object Oriented Programming.
- **ERROR : multifile** ***F/A*** **is not allowed to be inheritable.**

  Analogous to previous error.
- **ERROR : multifile** ***F/A*** **is not allowed to be virtual.**

  Analogous to previous error.
- **ERROR : virtual** ***F/A*** **must be a method or attribute defined at this class.**

  There is a virtual/1 declaration for *F/A*, but there is not any clause of that predicate nor a `data/1` declaration. You must declare at least one clause for every virtual method. Virtual attributes does not require any clause but a `data/1` declaration must be present.
- **ERROR : implemented interface** ***Module*** **is not a valid interface.**

  There is an `implements/1` declaration present at your code where given *Module* is not declared as class nor interface.
- **ERROR : predicate** ***F/A*** **is required both as method (at** ***Itf1*** **interface) and attribute (at** ***Itf2*** **interface).**

  There is no chance to give a correct implementation for *F/A* predicate since *Itf1* and *Itf2* interfaces require different definitions. To avoid this error, you must remove one of the related `implements/1` declaration.

- **ERROR : inherited *Source* must be a class.**

  There is an :- inherit_class(*Source*) declaration, but that source was not declared as a class.
- **ERROR : circular inheritance: *Source* is not a valid super-class.**

  Establishing an inheritance relationship with *Source* will cause current class to be present twice in the inheritance line. This is not allowed. The cause of this is error is simple : There is some inherited class from *Source* which also establishes an inheritance relationship with current source.
- **ERROR : method/attribute *F/A* must be implemented.**

  Some of the implemented interfaces requires *F/A* to be defined, but there is no definition for such predicate, even an inherited one.
- **ERROR : local implementation of *F/A* hides inheritable/public definition.**

  There is an inherited definition for *F/A* which is been redefined at current class, but there is no valid inheritable/public declaration for the last one. Overriden public predicates must be also declared as public. Overriden inheritable predicates must be declared either as public or inheritable.
- **ERROR : public predicate *F/A* was not defined nor inherited.**

  There is a `public/1` declaration for *F/A*, but there is no definition for it at current class nor an inherited one.
- **ERROR : argument to self/1 must be a free variable.**

  Argument to self/1 is not a variable, for example: `self(abc)`.
- **ERROR : unknown inherited attribute in *Goal*.**

  Goal belongs to assert/retract family of predicates, and given argument is not a valid inherited attribute. The most probable causes of this error are:
  - The given predicate is defined at super-class, but you forgot to mark it as inheritable (or public), at such class.
  - The given predicate was not defined (at super-class) as an attribute, just as a method.
- **ERROR : unknown inherited goal: *Goal*.**

  The given *Goal* was not found at super-class, or it is not accessible. Check whether *Goal* was marked as inheritable (or public) at super-class.
- **ERROR : invalid argument: *F/A* is not an attribute.**

  You are trying to pass a method as an argument to any predicate which expect a *fact* predicate.
- **ERROR : unknown inherited fact: *Fact*.**

  There is a call to any predicate which expects a *fact* argument (those declared as data or dynamic),but the actual argument is not an inherited attribute.For example:

  `asserta_fact(inherited(not_an_attribute(8)))`

  where not_an_attribute/1 was not declared as data or dynamic by the super-class (or corresponding ascendant).
- **ERROR : unknown inherited spec: *F/A*.**

  There is a reference to an inherited predicate specification, but the involved predicate has not been inherited.
- **WARNING : meta-predicate specification of *F/A* ignored since this is an attribute.**

  You declared *F/A* both as an attribute and a meta-predicate. For example:

  `:- meta_predicate attr(goal).`

  `:- data attr/1.`

  There is no sense in declaring an attribute as meta-predicate.

- **WARNING : class destructor is public**

  There is a :- public(destructor/0) declaration present at your code. Marking a destructor as public is a very bad idea since anybody may destroy or corrupt an instance before the proper time.
- **WARNING : class destructor is inheritable**

  Analogous to previous error.
- **WARNING : There is no call to inherited constructor/s**

  You have not declared any constructor at your class, but there is any inherited constructor that should be called. Whenever you do not need constructors, but there is an inheritance relationship (where super-class declares a constructor), you should write a simple constructor as the following example:

  ```
  :- class(myclass).
  :- inherit_class(other_class).

  myclass :-
          other_class.
  ```
- **WARNING : multifile** ***F/A*** **hides inherited predicate.**

  You declared as multifle a predicate which matches an inherited predicate name. Any reference to the inherited predicate must be done by the ways of the inherited/1 qualificator.

93.4.2 Class and Interface error reporting at run time

- **EXCEPTION : error(existence_error(object_goal,*Goal*),*Mod*).**

  Called *Goal* from module (or class) *Mod* is unknown or has not been published.

93.4.3 Normal Prolog module system interaction

O'Ciao works in conjunction with the Ciao Prolog module system, which also reports its own error messages. This will cause Ciao to report a little criptic error messages due to the general mechanism of source-to-source expansion. Those are some tips you must consider when compiling a class:

- Any error relative to method 'm' with arity A will be reported for predicate 'obj$m'/A+1. For example :

  `WARNING: (lns 28-30) [Item,Itema] - singleton variables in obj$remove/2`

  This error is relative to method remove/1.
- `set_prolog_flag/1` declaration will be usefull when declaring multiple constructors. It will avoid some awful warnings. Example:

  ```
  :- class(myclass).

  %% Use this declaration whenever several constructors are needed.

  :- set_prolog_flag(multi_arity_warnings,off).

  myclass(_).

  myclass(_,_).

  :- set_prolog_flag(multi_arity_warnings,on).
  ```

93.5 Known bugs and planned improvements (`class`)

- addmodule and pred(N) meta-arguments are not allowed on meta-predicates.

94 Compile-time usage of objects

Author(s): Angel Fernandez Pineda.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.5#90 (2000/3/24, 14:29:27 CET)

This package is required to enable user code to create objects and manipulate them, as well as loading any needed class.

94.1 Usage and interface (`objects`)

- **Library usage:**

  Any code which needs to use objects must include the objects package:

  `:- module(`*ModuleName*`,`*Exports*`,[objects]).`

  You can use objects even if your code is a class. Note that declaring a class does not automatically enables the code to create instances.

  `:- class(`*ModuleName*`,[],[objects]).`

  This package enables both static and dynamic usage of objects.
- **New declarations defined:**

  `use_class/1`, `instance_of/2`, `new/2`.
- **Other modules used:**
  - *System library modules:*

    `objects/objects_rt`.

94.2 Documentation on new declarations (`objects`)

use_class/1: DECLARATION

It establishes an usage relationship between the given file (which is supposed to declare a class) and current source. Usage relationships are needed in order to enable code to create instances of the given class, and to make calls to instances derived from such class.

Since an interface is some kind of class, they may be used within this declaration but only for semantic checking porpouses. Instances will not be derived from interfaces.

use_class/1 is used in the same way as `use_module/1`.

Usage: :- `use_class(ClassSource)`.

- *Description:* Establish usage relationship with `ClassSource`.
- *The following properties should hold at call time:*

  `ClassSource` is a valid path to a prolog file containing a class declaration (without .pl extension). (`objects_rt:class_source/1`)

instance_of/2: DECLARATION

Statically declares an identifier to be an instance of a given class.

It may be used as `new/2` predicate except for:

- The instance identifier will not be a variable, it must be provided by the user, and must be unique.
- Instance creation will never fail, even if the constructor fails.

For every statically declared object the given constructor will be called at program startup. Those instances may be destroyed manually, but it is not recommended.

When reloading the involved class from the Ciao toplevel shell. It may destroy statically declared instances, and create them again.

Statically declared instances must be called using a specifically designed module-qualification: `ClassName(Object):Goal`. For example:

```
:- module(example,[main/0],[objects]).
:- use_class(library(counter)).
:- cnt instance_of counter(10).

main :-
     counter(cnt):decrease(1),
     counter(cnt):current_value(X),
     display(X).
```

But **statically written code** (only) is allowed to use module-style qualifications as a macro:

```
main :-
     cnt:decrease(1),
     cnt:current_value(X),
     display(X).
```

Notice that dynamically expanded goals such as `X=cnt,X:decrease(1)` will not work, use `X=counter(cnt),X:decrease(1)` instead.

Usage: :- `instance_of(Object,Constructor)`.

- *Description:* Declares `Object` to be an instance of the class denoted by `Constructor`.
- *The following properties should hold at call time:*

  `Object` is an unique term which identifies an object. (`objects_rt:instance_id/1`)

  `Constructor` is a term whose functor matches a class name. (`objects_rt:constructor/1`)

new/2: DECLARATION

This declaration has the same effect as `instance_of/2`.

Usage: :- `new(Object,Constructor)`.

- *Description:* Just an alias for `instance_of/2`.
- *The following properties should hold at call time:*

  `Object` is an unique term which identifies an object. (`objects_rt:instance_id/1`)

  `Constructor` is a term whose functor matches a class name. (`objects_rt:constructor/1`)

94.3 Other information (`objects`)

Compile-time errors are restricted to some local analysis. Since there is no type declaration in the Prolog language, there is no posibility to determine whenever a given variable will hold an instance of any class.

However, little semantic analysis is performed. User may aid to perform such an analysis by the usage of run time checks (which are also detected at compile time), or static declarations. For example:

`clause(Obj) :- Obj:a_method(334).`

O'Ciao may be not able to determine whenever a_method/1 is a valid method for instance Obj, unless some help is provided:

`clause(Obj) :- Obj instance_of myclass,Obj:a_method(334).`

In such case, O'Ciao will report any semantic error at compile-time.

Most of the run-time errors are related to normal Ciao Prolog module system. Since objects are treated as normal Prolog modules at run time, there is no further documentation here about that stuff.

94.3.1 Error reporting at compile time (objects)

- **ERROR : invalid instance identifier *ID*: must be an atom**

  There is a `instance_of/2` or `new/2` declaration where first argument *ID* must be an unique atom, but currently it is not. Statically declared instances needs an identifier to be provided by the user.
- **ERROR : instance identifier *ID* already in use**

  There are two or more `instance_of/2` declarations with the same first argument *ID*. Instance identifiers must be unique.
- **ERROR : invalid use_class/1 declaration: *SourceFile* is not a class**

  Those are the causes for this error:
  - The given *SourceFile* does not exist, or is not accesible.
  - The given *SourceFile* is not a Prolog source.
  - The given *SourceFile* is a valid Prolog source, but it does not declare a class.
- **ERROR : unknown class on *ID* instance declaration**

  The class defined on the `instance_of/2` declaration for *ID* instance has not been loaded by a `use_class/1` declaration.
- **ERROR : instance identifier *ID* is an exisisting Prolog module**

  There is an statically declared instance whose identifier may cause interference with the Ciao Prolog module system. Use another instance identifier.
- **ERROR : unknown constructor on *ID* instance declaration**

  The given constructor on the `instance_of/2` declaration for *ID* has not been defined at the corresponding class.
- **ERROR : constructor is needed on *ID* instance declaration**

  No constructor was defined on the `instance_of/2` declaration for *ID* and default constructor is not allowed. You must provide a constructor.
- **ERROR : static instance *ID* was derived from a different constructor at *AnotherModule***

  ID has been declared to be an static instance both on *AnotherModule* and current source, but different constructors were used. The most probable causes for this error are:
  - Occasionally, there is another module using the same instance identifier and it was not noticed by you. Another different identifier may be used instead.
  - It was you intention to use the same object as declared by the other module. In this case, the same constructor must be used.
- **ERROR : invalid first argument in call to new(*Arg*,_)**

  There is a new/1 goal in your code where first argument is not a free variable. For example:

`myobj new myclass`

First argument must be a variable in order to receive a run-time generated object identifier.

- **ERROR : unknown class in call to new(?,*Constructor*)**

  The given *Constructor* in call to new/2 does not correspond to any used class at current code. The most probable cause of this may be:

  - You forgot to include a `use_class/1` declaration in your code.
  - There is a spelling mistake in the constructor.For example:

    :- use_class(myclass).

    foo(X) :- X new mclass.
- **ERROR : can not create an instance from an interface: new(?,*Constructor*)**

  Given *Constructor* references an interface rather than a class. Instances can not be derived from interface-expanded code.
- **ERROR : unknown constructor in call to new(?,*Constructor*)**

  As the previous error, there is a mistake in the given *Constructor*. This error is reported when you are trying to call a constructor which was not defined at the corresponding class. Check the class definition to find what is going on.

  Another cause for this error is the incorrect usage of the default constructor. Whenever there are one or more constructors defined at the involved class, you are restricted to chose one of them. This seems that default constructor will be available, if and only if, there are no constructors defined at the involved class.
- **ERROR : call to non-public *ID:Goal***

  You are trying to call a method which was not declared as public by the class specified in `instance_of/2` declaration for *ID*.
- **ERROR : call to inaccessible predicate at instance *ID:Goal***

  There is a call to *Goal* at statically declared instance *ID* which is unknown or was not declared as public.
- **ERROR : unknown instance *ID* of class *Class* at *Goal***

  There is a call to *Goal* where involved statically declared instance *ID* is unknown or is not derived from *Class*. Check whether it was declared by a `instance_of/2` declaration.
- **ERROR : inaccessible attribute *Fact* at instance *ID***

  There is an attempt to use *ID:Fact* but it was not declared as public.
- **ERROR : unknown attribute *Fact* at instance *ID***

  There is an attempt to use *ID:Fact* but it is unknown or it is not an attribute (may be a method).
- **WARNING : invalid call to new(?,_)**

  There is a call to new/2 in you code where first argument variable has been determined to hold any other instance. For example:

  `foo :- X new myclass,X new otherclass.`

  or

  `foo(X) :- X instance_of myclass, X new myclass.`

  The related call to new/2 will allways fail.
- **WARNING : called *Goal* is not public at any used class**

  There is a call to *Var*:*Goal* where *Var* has not been determined to be compatible with any class. However, *Goal* is not public at any class specified by the `use_class/1` declaration.

  This is a warning (not an error) since *Var*:*Goal* may be not related to Object Oriented Programing.

94.3.2 Error reporting at run time (objects)

- **EXCEPTION : instantiation_error('1st argument must be free variable')**
  Calling to new/1 requieres first argument to be a free variable. For example:
  `X = this_will_raise_an_exception,X new myclass.`
- **EXCEPTION : instantiation_error('class not given')**
  You called new/2 using a free variable as second argument.
- **EXCEPTION : instantiation_error(inaccesible_class(*Class*), from(*Module*))**
  Module tried to create an instance of *Class* by the ways of new/2, but there is no usage relationship between *Module* and *Class*.
- **EXCEPTION : instantiation_error(invalid_constructor(*Constructor*))**
  Constructor was not defined by the corresponding class.

95 Run time usage of objects

Author(s): Angel Fernandez Pineda, Angel Fernandez Pineda.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.5#4 (1999/11/29, 19:11:21 MET)

This library provides run-time support for object creation and manipulation. Objects are also called class instances, or simply instances.

Objects in Ciao are treated as normal modules. This is, an object is a run-time generated Prolog module, which may be identified by an unique term across the whole application.

This is a very simple example of how to create an instance, and how to make calls to it:

```
AnObj new myclass,
AnObj:mymethod.
```

In order to make any object accessible from code, an usage relationship must be established between the class (from which instances are derived) and the code itself. Refer to `use_class/1` predicate or `use_class/1` declaration in order to do so.

95.1 Usage and interface (objects_rt)

- **Library usage:**
  This library is automatically loaded when using the *objects* package:
  `:- module(`*ModuleName*`,`*Exports*`,[objects]).`
  Nothing special needs to be done.
- **Exports:**
  - *Predicates:*
    `new/2`, `instance_of/2`, `derived_from/2`, `interface/2`, `instance_codes/2`, `destroy/1`, `use_class/1`.
  - *Properties:*
    `constructor/1`, `class_name/1`, `interface_name/1`, `instance_id/1`, `class_source/1`, `interface_source/1`, `method_spec/1`, `virtual_method_spec/1`.
- **Other modules used:**
  - *System library modules:*
    `aggregates`, `dynamic`, `iso_misc`, `iso_byte_char`, `iso_incomplete`, `operators`, `read`, `streams`, `write`, `dynmods`, `prolog_sys`, `system`.

95.2 Documentation on exports (objects_rt)

new/2: PREDICATE

Dynamic instance creation takes place by the ways of this predicate.

It takes a free variable as first argument which will be instantiated to an internal object identifier.

Second argument must be instantiated to a class constructor. Class constructors are designed to perform an initialization on the new created instance. Notice that instance initialization may involve some kind of computation, not only *state initialization*.

A class constructor is made by a functor, which must match the intended class name, and any number of parameters. For example:

```
Obj new myclass(1500,'hello, world!!!')
```

Those parameters depends (obviously) on the constructors defined at the class source. If no constructors where defined, no parameters are needed. This is called the default constructor. An example:

```
Obj new myclass
```

The default constructor can not be called if there is any constructor available at the class source.

Instantiation will raise an exception and fail whenever any of this conditions occur:

- First argument is not a free variable.
- Second argument functor is a class, but there is no usage relationship with it.
- Second argument functor is not a class.
- The given constructor is unknown.
- The given constructor fails (notice that default constructor never fails).

Objects may also be statically declared, refer to `instance_of/2` declaration.

Usage: `new(InstanceVar,Constructor)`

- *Description:* Creates a new instance of the class specified by `Constructor` returning its identifier in `InstanceVar`
- *The following properties should hold at call time:*
  `InstanceVar` is a free variable. (`term_typing:var/1`)
  `Constructor` is a term whose functor matches a class name. (`objects_rt:constructor/1`)
- *The following properties should hold upon exit:*
  `InstanceVar` is an unique term which identifies an object. (`objects_rt:instance_id/1`)

instance_of/2: PREDICATE

This predicate is used to perform dynamic type checking. You may check whether a particular instance belongs to a particular class or related descendants.

instance_of/2 is used to perform static semantic analisys over object oriented code constructions.

By the use of instance_of/2 you may help to perform such analisys.

Usage 1: `instance_of(Instance,Class)`

- *Description:* Test whether `Instance` was derived from any descendant of `Class`, or that class itself
- *The following properties should hold at call time:*
  `Instance` is an unique term which identifies an object. (`objects_rt:instance_id/1`)
  `Class` is an atom denoting a class. (`objects_rt:class_name/1`)

Usage 2: `instance_of(Instance,Class)`

- *Description:* Retrieves, on backtracking, the inheritance line of `Instance` commencing on the creation class (that specified on call to `new/2`) and continuing on the rest of ascendant classes, if any.

- *The following properties should hold at call time:*
  `Instance` is an unique term which identifies an object. (`objects_rt:instance_id/1`)
  `Class` is a free variable. (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  `Class` is an atom denoting a class. (`objects_rt:class_name/1`)

derived_from/2: PREDICATE

Test whether an object identifier was derived directly from a class, by the usage of `new/2` or a static instance declaration (`instance_of/2`).

Usage 1: `derived_from(Instance,Class)`
- *Description:* Test derivation of `Instance` from `Class`
- *The following properties should hold at call time:*
  `Instance` is an unique term which identifies an object. (`objects_rt:instance_id/1`)
  `Class` is an atom denoting a class. (`objects_rt:class_name/1`)

Usage 2: `derived_from(Instance,Class)`
- *Description:* Retrieves the `Class` responsable of the derivation of `Instance`.
- *The following properties should hold at call time:*
  `Instance` is an unique term which identifies an object. (`objects_rt:instance_id/1`)
  `Class` is a free variable. (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  `Class` is an atom denoting a class. (`objects_rt:class_name/1`)

interface/2: PREDICATE

This predicate is used to ensure a given interface to be implemented by a given instance.

Usage 1: `interface(Instance,Interface)`
- *Description:* Check whether `Instance` implements the given `Interface`.
- *The following properties should hold at call time:*
  `Instance` is an unique term which identifies an object. (`objects_rt:instance_id/1`)
  `Interface` is an unique atom which identifies a public interface. (`objects_rt:interface_name/1`)

Usage 2: `interface(Instance,Interfaces)`
- *Description:* Retrieves on backtracking all the implemented `Interfaces` of `Instance`.
- *The following properties should hold at call time:*
  `Instance` is an unique term which identifies an object. (`objects_rt:instance_id/1`)
  `Interfaces` is a free variable. (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  `Interfaces` is an unique atom which identifies a public interface. (`objects_rt:interface_name/1`)

instance_codes/2: PREDICATE

Retrieves a character string representation from an object identifier and vice-versa.

Usage 1: `instance_codes(Instance,String)`

- *Description:* Retrieves a `String` representation of given `Instance`.
- *The following properties should hold at call time:*

  `Instance` is an unique term which identifies an object. (`objects_rt:instance_id/1`)

  `String` is a free variable. (`term_typing:var/1`)
- *The following properties should hold upon exit:*

  `String` is a string (a list of character codes). (`basic_props:string/1`)

Usage 2: `instance_codes(Instance,String)`

- *Description:* Reproduces an `Instance` from its `String` representation. Such an instance must be alive across the application: this predicate will fail whether the involved instance has been destroyed.
- *The following properties should hold at call time:*

  `Instance` is a free variable. (`term_typing:var/1`)

  `String` is a string (a list of character codes). (`basic_props:string/1`)
- *The following properties should hold upon exit:*

  `Instance` is an unique term which identifies an object. (`objects_rt:instance_id/1`)

destroy/1: PREDICATE

As well as instances are created, they must be destroyed when no longer needed in order to release system resources.

Unfortunately, current O'Ciao implementation does not support automatic instance destruction, so user must manually call *destroy/1* in order to do so.

The programmer **must ensure** that no other references to the involved object are left in memory when destroy/1 is called. If not, unexpected results may be obtained.

Usage: `destroy(Instance)`

- *Description:* Destroys the object identified by `Instance`.
- *The following properties should hold at call time:*

  `Instance` is an unique term which identifies an object. (`objects_rt:instance_id/1`)

use_class/1: PREDICATE

The behaviour of this predicate is identical to that provided by the declaration of the same name `use_class/1`. It allows user programs to dynamically load classes. Whether the given source is not a class it will perform a `use_module/1` predicate call.

Usage: `use_class(ClassSource)`

- *Description:* Dynamically loads the given `ClassSource`
- *The following properties should hold at call time:*

  `ClassSource` is a valid path to a prolog file containing a class declaration (without .pl extension). (`objects_rt:class_source/1`)

constructor/1: PROPERTY

Usage: `constructor(Cons)`

– *Description:* `Cons` is a term whose functor matches a class name.

class_name/1: PROPERTY

Usage: `class_name(ClassName)`

– *Description:* `ClassName` is an atom denoting a class.

interface_name/1: PROPERTY

Usage: `interface_name(Interface)`

– *Description:* `Interface` is an unique atom which identifies a public interface.

instance_id/1: PROPERTY

Usage: `instance_id(ID)`

– *Description:* `ID` is an unique term which identifies an object.

class_source/1: PROPERTY

Usage: `class_source(Source)`

– *Description:* `Source` is a valid path to a prolog file containing a class declaration (without .pl extension).

interface_source/1: PROPERTY

Usage: `interface_source(Source)`

– *Description:* `Source` is a valid path to a prolog file containing a class declaration or an interface declaration (without .pl extension).

method_spec/1: PROPERTY

There is no difference between method or attribute specifications, and habitual predicate specifications. It is just a Functor/Arity term.

Usage: `method_spec(Spec)`

– *Description:* `Spec` is a method or attribute specification.

virtual_method_spec/1: PROPERTY

Usage: `virtual_method_spec(Spec)`

– *Description:* `Spec` is a method specification.

95.3 Known bugs and planned improvements (`objects_rt`)

- Not really a bug: when loading code which declares static instances from the toplevel shell, predicate `use_module/1`) will not work properly: those instances may be not correctly created, and predicates will fail whenever they are not supposed to do. This may be avoided by reloading again the involved module, but make sure it is modified and saved to disk before doing so.

PART VIII - Interfaces to other languages and systems

The following interfaces to/from Ciao Prolog are documented in this part:

- External interface (e.g., to C).
- Socket interface.
- Tcl/tk interface.
- Web interface (http, html, xml, etc.);
- Persistent predicate databases (interface between the Prolog internal database and the external file system).
- SQL-like database interface (interface between the Prolog internal database and external SQL/ODBC systems).
- Java interface.
- Calling emacs from Prolog.

96 foreign_interface (library)

96.1 Usage and interface (foreign_interface)

- **Library usage:**
  `:- use_package(foreign_interface).`
  or
  `:- module(...,...,[foreign_interface]).`
- **Exports:**
  - *Properties:*
    size_of/3, do_not_free/2, returns/2, foreign/1, foreign/2, native/1, native/2.
  - *Regular Types:*
    address/1, byte/1, byte_list/1.
- **New operators defined:**
  =>/2 [975,xfx], ::/2 [978,xfx], decl/1 [1150,fx], decl/2 [1150,xfx], pred/1 [1150,fx], pred/2 [1150,xfx], prop/1 [1150,fx], prop/2 [1150,xfx], modedef/1 [1150,fx], calls/1 [1150,fx], calls/2 [1150,xfx], success/1 [1150,fx], success/2 [1150,xfx], comp/1 [1150,fx], comp/2 [1150,xfx], entry/1 [1150,fx], regtype/1 [1150,fx], regtype/2 [1150,xfx].
- **New declarations defined:**
  comment/2, decl/1, decl/2, pred/1, pred/2, prop/1, prop/2, modedef/1, calls/1, calls/2, success/1, success/2, comp/1, comp/2, entry/1, regtype/1, regtype/2.

96.2 Documentation on new declarations (foreign_interface)

comment/2: DECLARATION

No further documentation available for this predicate.

decl/1: DECLARATION

No further documentation available for this predicate.

decl/2: DECLARATION

No further documentation available for this predicate.

pred/1: DECLARATION

No further documentation available for this predicate.

pred/2: DECLARATION

No further documentation available for this predicate.

prop/1: DECLARATION

No further documentation available for this predicate.

prop/2: DECLARATION

No further documentation available for this predicate.

modedef/1: DECLARATION

No further documentation available for this predicate.

calls/1: DECLARATION

No further documentation available for this predicate.

calls/2: DECLARATION

No further documentation available for this predicate.

success/1: DECLARATION

No further documentation available for this predicate.

success/2: DECLARATION

No further documentation available for this predicate.

comp/1: DECLARATION

No further documentation available for this predicate.

comp/2: DECLARATION

No further documentation available for this predicate.

entry/1: DECLARATION

No further documentation available for this predicate.

regtype/1: (UNDOC_REEXPORT)

Imported from `basic_props` (see the corresponding documentation for details).

regtype/2: DECLARATION

No further documentation available for this predicate.

96.3 Documentation on exports (`foreign_interface`)

address/1: REGTYPE

Usage: `address(Address)`

– *Description:* `Address` is a valid memory address.

byte/1: REGTYPE

Usage: `byte(Byte)`

– *Description:* `Byte` is a byte

byte_list/1: REGTYPE

Usage: `byte_list(List)`

– *Description:* `List` is a list of bytes.

size_of/3: PROPERTY

Usage: `size_of(Name,ListVar,SizeVar)`

– *Description:* For predicate `Name`, the size of the argument of type `byte_list/1`, `ListVar`, is given by the argument of type integer `SizeVar`.

do_not_free/2: PROPERTY

Usage: `do_not_free(Name,Var)`

– *Description:* For predicate `Name`, the C argument passed to (returned from) the foreign function will not be freed after calling the foreign function.

returns/2: PROPERTY

Usage: `returns(Name,Var)`

– *Description:* The result of the foreign function that implements the Prolog predicate `Name` is unified with the Prolog variable `Var`. Cannot be used without `foreign/1` or `foreign/2`.

foreign/1: PROPERTY

Usage: `foreign(Name)`

– *Description:* The Prolog predicate `Name` is implemented using the foreign function `Name`.

foreign/2: PROPERTY

Usage: `foreign(PrologName,ForeignName)`

– *Description:* The Prolog predicate `PrologName` is implemented using the foreign function `ForeignName`.

native/1: PROPERTY

Usage: `native(Name)`

- *Description:* The Prolog predicate `Name` is implemented using the native function `Name`.

native/2: PROPERTY

Usage: `native(PrologName,ForeignName)`

- *Description:* The Prolog predicate `PrologName` is implemented using the native function prolog_`ForeignName`.

97 Utilities for on-demand compilation of foreign files

Author(s): Manuel Carro, Jose Morales.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.3#36 (1999/7/20, 10:37:31 MEST)

This module provides two predicates which give the user information regarding how to compile external (C) files in order to link them with the Ciao Prolog engine at runtime.

These predicates are not intended to be called directly by the end-user. Instead, a tool or module whose aim is generating dynamically loadable files from source files should use the predicates in this file in order to find out what are the proper compiler and linker to use, and which options must be passed to them in the current architecture.

97.1 Usage and interface (`foreign_compilation`)

- **Library usage:**
  `:- use_module(library(foreign_compilation)).`
- **Exports:**
  - *Predicates:*
    `compiler_and_opts/2`, `linker_and_opts/2`.
- **Other modules used:**
  - *System library modules:*
    `system`.

97.2 Documentation on exports (`foreign_compilation`)

compiler_and_opts/2: PREDICATE

Usage: `compiler_and_opts(?(Compiler),?(Opts))`

- *Description:* If you want to compile a foreign language file for dynamic linking in the current operating system and architecture, you have to use the compiler `Compiler` and gite it the options `Opts`. A variable in `Opts` means that no special option is needed.
- *The following properties should hold at call time:*
  `?(Compiler)` is currently instantiated to an atom. (`term_typing:atom/1`)
  `?(Opts)` is a list of `atom`s. (`basic_props:list/2`)

linker_and_opts/2: PREDICATE

Usage: `linker_and_opts(?(Linker),?(Options))`

- *Description:* If you want to link a foreign language file for dynamic linking in the current operating system and architecture, you have to use the linker `Compiler` and gite it the options `Opts`. A variable in `Opts` means that no special option is needed.
- *The following properties should hold at call time:*
  `?(Linker)` is currently instantiated to an atom. (`term_typing:atom/1`)
  `?(Options)` is a list of `atom`s. (`basic_props:list/2`)

98 The socket interface

Author(s): Manuel Carro, Daniel Cabeza.

Version: 1.5#128 (2000/5/2, 18:4:14 CEST)

Version of last change: 1.5#127 (2000/5/2, 16:50:36 CEST)

This module defines primitives to open sockets, send, and receive data from them. This allows communicating with other processes, on the same machine or across the Internet. The reader should also consult standard bibliography on the topic for a proper use of these primitives.

98.1 Usage and interface (sockets)

- **Library usage:**
  `:- use_module(library(sockets)).`
- **Exports:**
  - *Predicates:*
    `connect_to_socket/3`, `socket_recv/2`, `socket_recv_code/3`, `socket_send/2`, `select_socket/5`, `socket_accept/2`, `bind_socket/3`, `connect_to_socket_type/4`.
  - *Regular Types:*
    `socket_type/1`.
- **Other modules used:**
  - *System library modules:*
    `sockets/sockets_c`.

98.2 Documentation on exports (sockets)

connect_to_socket/3: PREDICATE

Usage: `connect_to_socket(+Host,+Port,-(Stream))`

- *Description:* Calls `connect_to_socket_type/4` with SOCK_STREAM connection type. This is the connection type you want in order to use the `write/2` and `read/2` predicates (and other stream IO related predicates).
- *Call and exit should be compatible with:*
  `+Host` is an atom. (`basic_props:atm/1`)
  `+Port` is an integer. (`basic_props:int/1`)
  `-(Stream)` is an open stream. (`streams_basic:stream/1`)

socket_recv/2: PREDICATE

Usage: `socket_recv(+Stream,?(String))`

- *Description:* As `socket_recv_code/3`, but the return code is ignored.
- *Call and exit should be compatible with:*
  `+Stream` is an open stream. (`streams_basic:stream/1`)
  `?(String)` is a string (a list of character codes). (`basic_props:string/1`)

socket_type/1: REGTYPE

Defines the atoms which can be used to specify the socket type recognized by `connect_to_socket_type/4`. Defined as follows:

```
socket_type(stream).
socket_type(dgram).
socket_type(raw).
socket_type(seqpacket).
socket_type(rdm).
```

Usage: `socket_type(T)`

- *Description:* `T` is a valid socket type.

socket_recv_code/3: PREDICATE

Usage: `socket_recv_code(+Stream,?(String),?(Length))`

- *Description:* Receives a `String` from the socket associated to `Stream`, and returns its `Length`. If `Length` is -1, no more data is available.
- *Call and exit should be compatible with:*

  `+Stream` is an open stream. (`streams_basic:stream/1`)

  `?(String)` is a string (a list of character codes). (`basic_props:string/1`)

  `?(Length)` is an integer. (`basic_props:int/1`)

socket_send/2: PREDICATE

Usage: `socket_send(+Stream,+String)`

- *Description:* Sends `String` to the socket associated to `Stream`. The socket has to be in connected state. `String` is not supposed to be NULL terminated, since it is a Prolog string. If a NULL terminated string is needed at the other side, it has to be explicitly created in Prolog.
- *Call and exit should be compatible with:*

  `+Stream` is an open stream. (`streams_basic:stream/1`)

  `+String` is a string (a list of character codes). (`basic_props:string/1`)

select_socket/5: PREDICATE

Usage: `select_socket(+Socket,-(NewStream),+TO_ms,+Streams,-(ReadStreams))`

- *Call and exit should be compatible with:*

  `+Socket` is an integer. (`basic_props:int/1`)

  `-(NewStream)` is an open stream. (`streams_basic:stream/1`)

  `+TO_ms` is an integer. (`basic_props:int/1`)

  `+Streams` is a list of `streams`. (`basic_props:list/2`)

  `-(ReadStreams)` is a list of `streams`. (`basic_props:list/2`)

socket_accept/2: PREDICATE

Usage: `socket_accept(+Sock,-(Stream))`

- *Description:* Creates a new `Stream` connected to `Sock`.

- *Call and exit should be compatible with:*
  +Sock is an integer. (basic_props:int/1)
  -(Stream) is an open stream. (streams_basic:stream/1)

bind_socket/3: PREDICATE

Usage: `bind_socket(?(Port),+Length,-(Socket))`

- *Description:* Returs an AF_INET `Socket` bound to `Port` (which may be assigned by the OS or defined by the caller), and listens to it (hence no listen call in this set of primitives). `Length` specifies the maximum number of pending connections.
- *Call and exit should be compatible with:*
  ?(Port) is an integer. (basic_props:int/1)
  +Length is an integer. (basic_props:int/1)
  -(Socket) is an integer. (basic_props:int/1)

connect_to_socket_type/4: PREDICATE

Usage: `connect_to_socket_type(+Host,+Port,+Type,-(Stream))`

- *Description:* Returns a `Stream` which connects to `Host`. The `Type` of connection can be defined. A `Stream` is returned, which can be used to `write/2` to, to `read/2`, to `socket_send/2` to, or to `socket_recv/2` from the socket.
- *Call and exit should be compatible with:*
  +Host is currently instantiated to an atom. (term_typing:atom/1)
  +Port is an integer. (basic_props:int/1)
  +Type is a valid socket type. (sockets:socket_type/1)
  -(Stream) is an open stream. (streams_basic:stream/1)

99 The tcl/tk interface

Author(s): Montse Iglesias.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.5#73 (2000/3/19, 19:54:31 CET)

The `tcltk` library package is a bidirectional to the *Tcl* (pronounced Tickle) language and *Tk* toolkit. Tcl is an interpreter scripting language with many extensions packages, in particular the graphical interface toolkit Tk

99.1 Usage and interface (`tcltk`)

- **Library usage:**
  `:- use_module(library(tcltk)).`
- **Exports:**
  - *Predicates:*
    `tcl_new/1`, `tcl_eval/3`, `tcl_delete/1`, `tcl_event/3`, `tk_event_loop/1`, `tk_new/2`, `tk_new/4`.
  - *Regular Types:*
    `tclInterpreter/1`, `tclCommand/1`.
- **Other modules used:**
  - *System library modules:*
    `tcltk/tcltk_low_level`, `write`, `strings`.

99.2 Documentation on exports (`tcltk`)

tclInterpreter/1: REGTYPE

Usage: `tclInterpreter(I)`

- *Description:* `I` is a reference to a *Tcl* interpreter.

tclCommand/1: REGTYPE

A *Tcl* command is specified as follows:

```
Command         --> Atom  { other than [] }
                  | Number
                  | chars(PrologString)
                  | write(Term)
                  | format(Fmt,Args)
                  | dq(Command)
                  | br(Command)
                  | sqb(Command)
                  | min(Command)
                  | ListOfCommands
ListOfCommands  --> []
                  |[Command|ListOfCommands]
```

where:

Atom

Number denote their printed representations.

chars(PrologString)
: denotes the string represented by *PrologString* (a list of character codes).

write(Term)
: denotes the string that is printed by the corresponding built-in pridicate.

format(Term)
: denotes the string that is printed by the corresponding built-in pridicate.

dq(Command)
: denotes the string specified by *Command*, enclosed in double quotes.

br(Command)
: denotes the string specified by *Command*, enclosed in braces.

sqb(Command)
: denotes the string specified by *Command*, enclosed in square brackets.

min(Command)
: denotes the string specified by *Command*, immediately preceded by a hyphen.

ListOfCommands
: denotes the strings denoted by each element, separated by spaces.

Usage: `tclCommand(C)`
- *Description:* `C` is a *Tcl* command.

tcl_new/1: PREDICATE

Usage: `tcl_new(-(TclInterpreter))`
- *Description:* Creates a new interpreter, initializes it, and returns a reference to it in `TclInterpreter`.
- *Call and exit should be compatible with:*
  `-(TclInterpreter)` is a reference to a *Tcl* interpreter. (`tcltk:tclInterpreter/1`)

tcl_eval/3: PREDICATE

Usage: `tcl_eval(+TclInterpreter,+Command,-(Result))`
- *Description:* Evaluates the commands and the result will be stored as a string in *Result*
- *Call and exit should be compatible with:*
  `+TclInterpreter` is a reference to a *Tcl* interpreter. (`tcltk:tclInterpreter/1`)
  `+Command` is a *Tcl* command. (`tcltk:tclCommand/1`)
  `-(Result)` is a string (a list of character codes). (`basic_props:string/1`)

tcl_delete/1: PREDICATE

Usage: `tcl_delete(+TclInterpreter)`
- *Description:* Deletes the interpreter in `TclInterpreter` and the memory used by it
- *Call and exit should be compatible with:*
  `+TclInterpreter` is a reference to a *Tcl* interpreter. (`tcltk:tclInterpreter/1`)

tcl_event/3: PREDICATE

Usage: `tcl_event(+TclInterpreter,+Command,-(Events))`

- *Description:* Do the receive non blocking of the event and terms will be stored from Tcl by the *prolog_event* command as a list of terms in *Events*.
- *Call and exit should be compatible with:*

  `+TclInterpreter` is a reference to a *Tcl* interpreter. (`tcltk:tclInterpreter/1`)

  `+Command` is a *Tcl* command. (`tcltk:tclCommand/1`)

  `-(Events)` is a list. (`basic_props:list/1`)

tk_event_loop/1: PREDICATE

No further documentation available for this predicate.

tk_new/2: PREDICATE

Usage: `tk_new(+Options,-(TclInterpreter))`

- *Description:* Performs basic Tcl and Tk initialization and creates the main window of a Tk application.*Options* is a list of optional elements according to :

  name — Sets the Tk application name. The application name will be displayed in the main window and is also used for communicating between applications in Tk. Default name is an empty string.

  display — Gives the name of the screen on which to create the main window. Default is normally determined by the DISPLAY environment variable.

  file — Open the sript file, commands will not be read from standard input and the execution returns back to Prolog only after all windows (and the interpreter) have been deleted.
- *Call and exit should be compatible with:*

  `+Options` is a list. (`basic_props:list/1`)

  `-(TclInterpreter)` is a reference to a *Tcl* interpreter. (`tcltk:tclInterpreter/1`)

tk_new/4: PREDICATE

No further documentation available for this predicate.

100 Connecting with the Web (PiLLoW)

This package implements the PiLLoW library [CHV96a] in Ciao. It is not yet documented here, but you can access `http://clip.dia.fi.upm.es/Software/pillow/pillow.html` to see documentation on previous versions. However, note that this version is newer and has some bug fixes and added functionality (templates, XML processing, and others, which you can try to guess the rest by looking at the source). You can also find a paper and some additional information in the `library/pillow/doc` directory of the distribution.

100.1 Usage and interface (pillow)

- **Library usage:**
  `:- use_package(pillow).`
  or
  `:- module(...,...,[pillow]).`
- **New operators defined:**
  `$/2` [150,xfx], `$/1` [150,fx].
- **Other modules used:**
  - *System library modules:*
    `pillow/http`, `pillow/html`.

101 html (library)

Version: 1.5#128 (2000/5/2, 18:4:14 CEST)
Version of last change: 1.5#114 (2000/4/11, 20:23:43 CEST)

101.1 Usage and interface (`html`)

- **Library usage:**
  `:- use_module(library(html)).`
- **Exports:**
  - *Predicates:*
    `url_info/2`, `url_info_relative/3`, `url_query/2`, `output_html/1`, `html2terms/2`, `xml2terms/2`, `html_template/3`, `http_lines/3`, `html_report_error/1`, `get_form_input/1`, `get_form_value/3`, `form_empty_value/1`, `text_lines/2`, `form_default/3`, `my_url/1`, `form_request_method/1`, `icon_address/2`, `html_protect/1`.
- **Other modules used:**
  - *System library modules:*
    `strings`, `lists`, `system`, `pillow/common`.

101.2 Documentation on exports (`html`)

url_info/2: PREDICATE
No further documentation available for this predicate.

url_info_relative/3: PREDICATE
No further documentation available for this predicate.

url_query/2: PREDICATE
No further documentation available for this predicate.

output_html/1: PREDICATE
No further documentation available for this predicate.

html2terms/2: PREDICATE
No further documentation available for this predicate.

xml2terms/2: PREDICATE
No further documentation available for this predicate.

html_template/3: PREDICATE

No further documentation available for this predicate.

http_lines/3: PREDICATE

No further documentation available for this predicate.

html_report_error/1: PREDICATE

No further documentation available for this predicate.

get_form_input/1: PREDICATE

No further documentation available for this predicate.

get_form_value/3: PREDICATE

No further documentation available for this predicate.

form_empty_value/1: PREDICATE

No further documentation available for this predicate.

text_lines/2: PREDICATE

No further documentation available for this predicate.

form_default/3: PREDICATE

No further documentation available for this predicate.

my_url/1: PREDICATE

No further documentation available for this predicate.

form_request_method/1: PREDICATE

No further documentation available for this predicate.

icon_address/2: PREDICATE

No further documentation available for this predicate.

html_protect/1: PREDICATE

No further documentation available for this predicate.

Meta-predicate with arguments: `html_protect(goal)`.

102 http (library)

Version: 1.5#128 (2000/5/2, 18:4:14 CEST)
Version of last change: 1.3#114 (1999/11/24, 0:57:16 MET)

102.1 Usage and interface (`http`)

- **Library usage:**
  `:- use_module(library(http)).`
- **Exports:**
  - *Predicates:*
    `fetch_url/3`.
- **Other modules used:**
  - *System library modules:*
    `sockets/sockets`, `strings`, `lists`, `file_utils`, `pillow/common`.

102.2 Documentation on exports (`http`)

fetch_url/3: PREDICATE

No further documentation available for this predicate.

103 Persistent predicate database

Author(s): J.M. Gomez, D. Cabeza, and M. Hermenegildo, `clip@dia.fi.upm.es`, `http://www.clip.dia.fi.upm.es/`, The CLIP Group, Facultad de Informática, Universidad Politécnica de Madrid.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 0.8#31 (1998/12/27, 19:17:20 MET)

103.1 Introduction to persistent predicates

This library implements a *generic persistent predicate database*. The basic notion implemented by the library is that of a persistent predicate. The persistent predicate concept provides a simple, yet powerful generic persistent data access method [CHGT98,Par97]. A persistent predicate is a special kind of dynamic, data predicate that "resides" in some persistent medium (such as a set of files, a database, etc.) that is typically external to the program using such predicates. The main effect is that any changes made to to a persistent predicate from a program "survive" across executions. I.e., if the program is halted and restarted the predicate that the new process sees is in precisely the same state as it was when the old process was halted (provided no change was made in the meantime to the storage by other processes or the user).

Persistent predicates appear to a program as ordinary predicates, and calls to these predicates can appear in clause bodies in the usual way. However, the definitions of these predicates do not appear in the program. Instead, the library maintains automatically the definitions of predicates which have been declared as persistent in the persistent storage.

Updates to persistent predicates can be made by calling predicates similar to `assertz_fact/1` and `retract_fact/1`. The library makes sure that each update is a transactional update, in the sense that if the update terminates, then the permanent storage has definitely been modified. For example, if the program making the updates is halted just after the update and then restarted, then the updated state of the predicate will be seen. This provides security against possible data loss due to, for example, a system crash. Also, due to the atomicity of the transactions, persistent predicates allow concurrent updates from several programs.

103.2 Persistent predicates, files, and relational databases

The concept of persistent predicates provided by this library essentially implements a lightweight, simple, and at the same time powerful form of relational database (a deductive database), and which is standalone, in the sense that it does not require external support, other than the file management capabilities provided by the operating system. This is due to the fact that the persistent predicates are in fact stored in one or more auxiliary files in a given directory.

This type of database is specially useful when building small to medium-sized standalone applications in Prolog which require persistent storage. In many cases it provides a much easier way of implementing such storage than using files under direct program control. For example, interactive applications can use persistent predicates to represent their internal state in a way that is close to the application. The persistence of such predicates then allows automatically restoring the state to that at the end of a previous session. Using persistent predicates amounts to simply declaring some predicates as such and eliminates having to worry about opening files, closing them, recovering from system crashes, etc.

In other cases, however, it may be convenient to use a relational database as persistent storage. This may be the case, for example, when the data already resides in such a database (where it is perhaps accessed also by other applications) or the volume of data is very large. `persdb_sql` [CCG98] is a companion library which implements the same notion of persistent predicates used herein, but keeping the storage in a relational database. This provides a very natural and transparent way to access SQL database relations from a Prolog program. In that library, facilities are also provided for reflecting more complex *views* of the database relations as predicates. Such views can be constructed as conjunctions, disjunctions, projections, etc. of database relations, and may include SQL-like aggregation operations.

A nice characteristic of the notion of persistent predicates used in both of these libraries is that it abstracts away how the predicate is actually stored. Thus, a program can use persistent predicates stored in files or in external relational databases interchangeably, and the type of storage used for a given predicate can be changed without having to modify the program (except for replacing the corresponding `persistent/2` declarations).

An example application of the `persdb` and `persdb_sql` libraries (and also the `pillow` library [CH97]), is `WebDB` [GCH98]. `WebDB` is a generic, highly customizable *deductive database engine* with an *html interface*. `WebDB` allows creating and maintaining Prolog-based databases as well as relational databases (residing in conventional relational database engines) using any standard WWW browser.

103.3 Using file-based persistent predicates

Persistent predicates can be declared statically, using `persistent/2` declarations (which is the preferred method, when possible), or dynamically via calls to `make_persistent/2`. Currently, persistent predicates may only contain facts, i.e., they are *dynamic* predicates of type `data/1`, and *should be declared as such.*

Predicates declared as persistent are linked to a directory, and the persistent state of the predicate will be kept in several files in that directory. The files in which the persistent predicates are stored are in readable, plain ASCII format, and in Prolog syntax. One advantage of this approach is that such files can also be created or edited by hand, in a text editor, or even by other applications.

An example definition of a persistent predicate implemented by files follows:

```
:- persistent(p/3,dbdir).

:- multifile persistent_dir/2.
:- data persistent_dir/2.

persistent_dir(dbdir, '/home/clip/public_html/db').
```

The first line declares the predicate `p/3` persistent. The argument `dbdir` is a key used to index into a fact of the relation `persistent_dir/2`, which specifies the directory where the corresponding files will be kept. The effect of the declaration, together with the `persistent_dir/2` fact, is that, although the predicate is handled in the same way as a normal data predicate, in addition the system will create and maintain efficiently a persistent version of `p/3` via files in the directory `/home/clip/public_html/db`.

The level of indirection provided by the `dbdir` argument makes it easy to place the storage of several persistent predicates in a common directory, by specifying the same key for all of them. It also allows changing the directory for several such persistent predicates by modifying only one fact in the program. Furthermore, the `persistent_dir/2` predicate can even be dynamic and specified at run-time.

103.4 Implementation Issues

We outline the current implementation approach. This implementation attempts to provide at the same time efficiency and security. To this end, up to three files are used for each predicate (the persistence set): the data file, the operations file, and the backup file. In the updated state the facts (tuples) that define the predicate are stored in the data file and the operations file is empty (the backup file, which contains a security copy of the data file, may or may not exist).

While a program using a persistent predicate is running, any insertion (assert) or deletion (retract) operations on the predicate are performed on both the program memory and on the persistence set. However, in order to incurr only a small overhead in the execution, rather than changing the data file directly, a record of each of the insertion and deletion operations is *appended* to the operations file. The predicate is then in a transient state, in that the contents of the data file do not reflect exactly the current state of the corresponding predicate. However, the complete persistence set does.

When a program starts, when it is halted, or, periodically, when it is idle for some time, all pending operations in the operations file are performed on the data file. A backup of the data file is created first to prevent data loss if the system crashes during this operation. The order in which this updating of files is done ensures that, if at any point the process dies, on restart data will be completely recovered. This process of updating the persistence set can also be triggered at any point in the execution of a program by calling `update_files/2`.

103.5 Using persistent predicates from the top level

Special care must be taken when using persistent predicates from the top level. This includes not only defining persistent predicates on the fly from de top level (which is not really very useful in practice) but also the more frequent case of loading into the top level modules or user files which use persistent predicates. As mentioned before, the persistence set is updated automatically each time a program using the corresponding persistent predicates is run or halted. However, since the top level itself is also a standard program, persistent predicates would only be updated whenever the top level is started, when they are typically still not loaded.

If a program launched from a top level needs to update the persistence sets of any persistent predicate it must be done by calling the `update_files/2` method explicitly.

103.6 Usage and interface (persdbrt)

- **Library usage:**

  Typically, this library is used including the 'persdb' package into the syntax list of the module, or using the `syntax/1` declaration:

  In a module:

  ```
  :- module(bar, [main/1], [persdb]).
  ```

  or

  ```
  :- module(bar, [main/1]).
  :- include(library(persdb)).
  ```

  In a *user* file:

  ```
  :- syntax([persdb]).
  ```

  or

  ```
  :- include(library(persdb)).
  ```

  This syntax file loads the run-time and compile-time versions of the library (`persdbtr.pl` and `persdbrt.pl`) and includes some needed declarations.
- **Exports:**
  - *Predicates:*

    `passertz_fact/1`, `pretract_fact/1`, `pcurrent_fact/1`, `init_persdb/0`, `initialize_db/0`, `make_persistent/2`, `update_files/2`.
  - *Multifiles:*

    `persistent_dir/2`, `$is_persistent/2`.
- **Other modules used:**
  - *System library modules:*

    `lists`, `streams`, `read`, `system`, `file_locks/file_locks`.

103.7 Documentation on exports (persdbrt)

passertz_fact/1: PREDICATE

Meta-predicate with arguments: `passertz_fact(fact)`.

Usage: `passertz_fact(Fact)`

- *Description:* Persistent version of `assertz_fact/1`: the current instance of `Fact` is interpreted as a fact (i.e., a relation tuple) and is added at the end of the definition of the corresponding predicate. The predicate concerned must be declared `persistent`. Any uninstantiated variables in the `Fact` will be replaced by new, private variables.
- *The following properties should hold at call time:*

  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

pretract_fact/1: PREDICATE

Meta-predicate with arguments: `pretract_fact(fact)`.

Usage: `pretract_fact(Fact)`

- *Description:* Persistent version of `retract_fact/1`: deletes on backtracking all the facts which unify with `Fact`. The predicate concerned must be declared `persistent`.
- *The following properties should hold at call time:*

  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

pcurrent_fact/1: PREDICATE

Meta-predicate with arguments: `pcurrent_fact(fact)`.

Usage: `pcurrent_fact(Fact)`

- *Description:* Persistent version of `current_fact/1`: the fact `Fact` exists in the current database. The predicate concerned must be declared `persistent`. Provides on backtracking all the facts (tuples) which unify with `Fact`.
- *The following properties should hold at call time:*

  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:callable/1`)

init_persdb/0: PREDICATE

Usage:

- *Description:* Executes `initialize_db/0` if no initialization has been done yet. Needed when accesing persistent predicates before doing any of `passertz_fact/1`, `pretract_fact/1`, or `pcurrent_fact/1`.

initialize_db/0: PREDICATE

Usage:

- *Description:* Initializes the whole database, updating the state of the declared persistent predicates.

make_persistent/2: PREDICATE

Meta-predicate with arguments: `make_persistent(spec,?)`.

Usage: `make_persistent(PredDesc,Keyword)`

- *Description:* Dynamic version of the `persistent` declaration.
- *The following properties should hold at call time:*

  `PredDesc` is a Name/Arity structure denoting a predicate name:

  ```
  predname(P/A) :-
          atm(P),
          int(A).
  ```

  (`basic_props:predname/1`)

  `Keyword` is an atom corresponding to a directory identifier. (`persdbrt:keyword/1`)

update_files/2: PREDICATE

Usage: `update_files(PredDesc,Arity)`

- *Description:* Updates the state of the given persistent predicate and its corresponding persistence set.
- *The following properties should hold upon exit:*

  `PredDesc` is a Name/Arity structure denoting a predicate name:

  ```
  predname(P/A) :-
          atm(P),
          int(A).
  ```

  (`basic_props:predname/1`)

  `Arity` is an integer. (`basic_props:int/1`)

103.8 Documentation on multifiles (`persdbrt`)

persistent_dir/2: PREDICATE

The predicate is *multifile*.

The predicate is of type *data*.

Usage: `persistent_dir(Keyword,Location_Path)`

- *Description:* Relates identifiers of locations (the `Keyword`s) with descriptions of such locations (`Location_Path`s). `Location_Path` is **a directory** and it means that the definition for the persistent predicates associated with `Keyword` is kept in files in that directory. These files, in the updated state, contain the actual definition of the predicate in Prolog syntax (but with module names resolved).
- *The following properties should hold at call time:*

  `Keyword` is an atom corresponding to a directory identifier. (`persdbrt:keyword/1`)

  `Location_Path` is an atom which is the name of a directory. (`persdbrt:directoryname/1`)

$is_persistent/2: PREDICATE

No further documentation available for this predicate.

The predicate is *multifile*.

103.9 Documentation on internals (`persdbrt`)

keyword/1: REGTYPE

An atom which identifies a fact of the `persistent_dir/2` relation. This fact relates this atom to a directory in which the persistent storage for one or more persistent predicates is kept.

Usage: `keyword(X)`

- *Description:* `X` is an atom corresponding to a directory identifier.

directoryname/1: REGTYPE

Usage: `directoryname(X)`

- *Description:* `X` is an atom which is the name of a directory.

103.10 Known bugs and planned improvements (`persdbrt`)

- Shouldn't the declarations for persistent_dir/2 be inside persdb.pl? (would save a lot of writing).
- make_persistent/2 should really be persistent/2 (since it doesn't really make a predicate persistent but rather declares it as such, i.e., we do not use make_data/1, we use data/1) ?
- having to use pcurrent_fact is not so nice
- passert, pretract, etc. should really be overloaded versions of assert, retract etc., i.e., it should be possible to use assertz_fact directly
- we need also asserta_fact, right? Otherwise isn't it difficult to do many things?

104 Using the persdb library

Through the following examples we will try to illustrate the two mains ways of declaring and using persistent predicates: statically (the preferred method) and dynamically (necessary when the new persistent predicates have to be defined at run-time). The final example is a small application implementing a simple persistent queue.

104.1 An example of persistent predicates (static version)

```
:- include(library(persdb)).

%% Required declarations for persistent_dir/2.
:- multifile persistent_dir/2.
:- data persistent_dir/2.

%% Declare the directory associated to the key "db" where the
%% persistence sets of the persistent predicates are stored:
persistent_dir(db,'./').

%% Declare a persistent predicate:
:- persistent(bar/1, db).

%% Read a term, storing it in a new fact of the persistent predicate
%% and list all the current facts of that predicate
main:-
     read(X),
     passertz_fact(bar(X)),
     findall(Y,bar(Y),L),
     write(L).
```

104.2 An example of persistent predicates (dynamic version)

```
:- include(library(persdb)).

%% Required declarations for persistent_dir/2.
:- multifile persistent_dir/2.
:- data persistent_dir/2.

main([X]):-
%%   Declare the directory associated to the key "db"
     asserta_fact(persistent_dir(db,'./')),
%%   Declare the predicate bar/1 as dynamic (and data) at run-time
     data(bar/1),
%%   Declare the predicate bar/1 as persistent at run-time
     make_persistent(bar/1, db),
     passertz_fact(bar(X)),
     findall(Y, bar(Y), L),
     write(L).
```

104.3 A simple application / a persistent queue

```
:- module(queue, [main/0],[]).

:- use_module(library(read)).
:- use_module(library(write)).
:- use_module(library(aggregates)).

:- include(library(persdb)).

:- multifile persistent_dir/2.
:- data persistent_dir/2.

:- persistent(queue/1, queue_dir).

persistent_dir(queue_dir,'./pers').

main:-
     write('Action ( in(Term). | out. | list. | halt. ): '),
     read(A),
     (  handle_action(A)
     -> true
     ;  write('Unknown command.'), nl ),
     main.

handle_action(halt) :-
     halt.
handle_action(in(Term)) :-
     passertz_fact(queue(Term)),
     main.
handle_action(out) :-
     (  pretract_fact(queue(Term))
     -> write('Out '), write(Term)
     ;  write('FIFO empty.') ),
     nl,
     main.
handle_action(list) :-
     findall(Term,pcurrent_fact(queue(Term)),Terms),
     write('Contents: '), write(Terms), nl,
     main.
```

105 SQL persistent database interface

Author(s): I. Caballero, D. Cabeza, J.M. Gómez, and M. Hermenegildo, `clip@dia.fi.upm.es`, `http://www.clip.dia.fi.upm.es/`, The CLIP Group, Facultad de Informática, Universidad Politécnica de Madrid.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.5#105 (2000/4/3, 14:21:58 CEST)

The purpose of this library is to implement an instance of the generic concept of persistent predicates, where external relational databases are used for storage (see the documentation of the `persdb` library and [CHGT98,Par97] for details). To this end, this library exports SQL persistent versions of the `assertz_fact/1`, `retract_fact/1` and `retractall_fact/1` builtin predicates. Persistent predicates also allow concurrent updates from several programs, since each update is atomic.

The notion of persistence provides a very natural and transparent way to access database relations from a Prolog program. Stub definitions are provided for such predicates which access the database when the predicate is called (using the `db_client` library). A Prolog to SQL translator is used to generate the required SQL code dynamically (see library `pl2sql`).

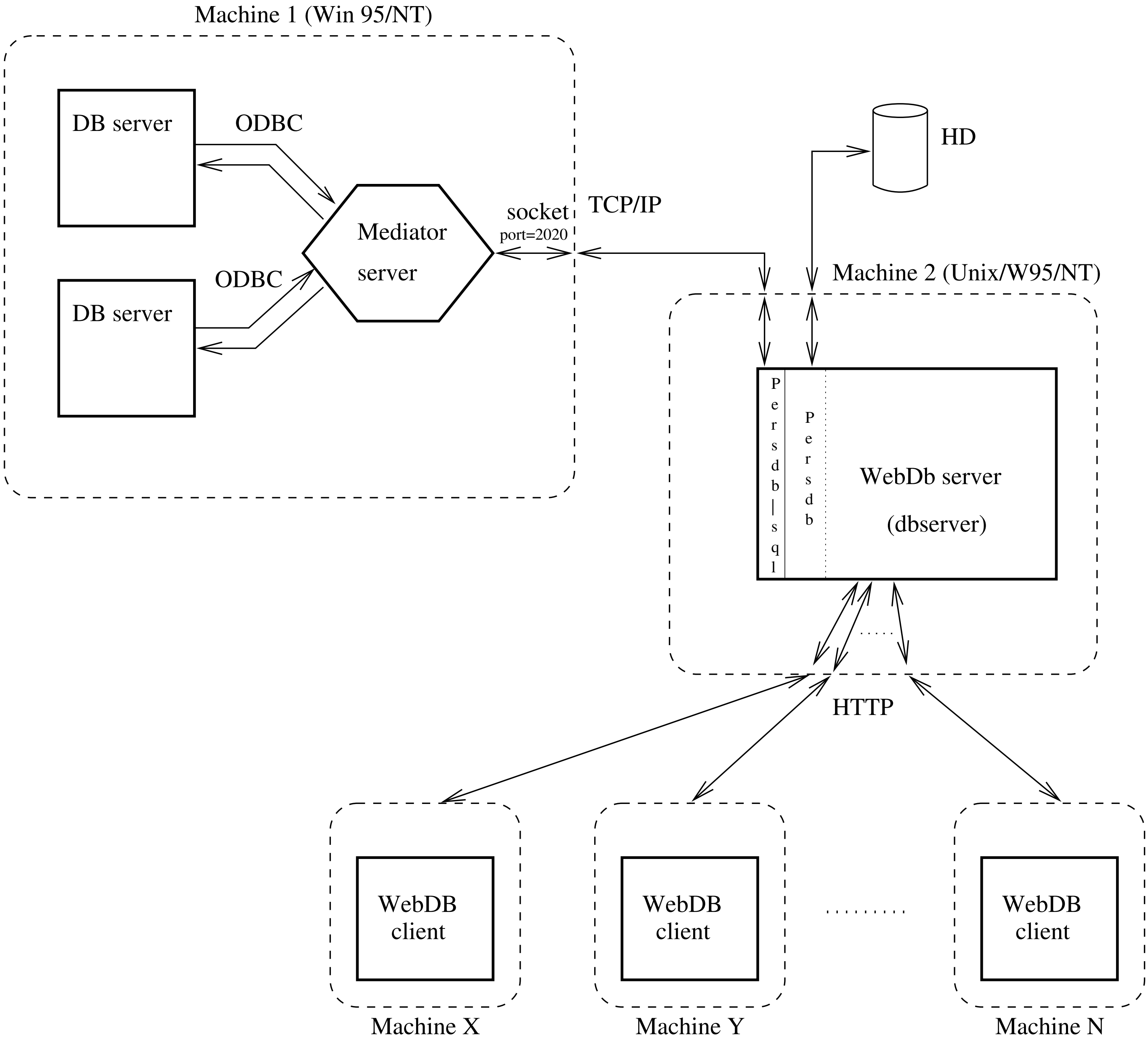
This library also provides facilities for reflecting more complex views of the database relations as Prolog predicates. Such views can be constructed as conjunctions, disjunctions, projections, etc. of database relations. Also, SQL-like aggregation operations are supported.

105.1 Implementation of the Database Interface

The architecture of the low-level implementation of the database interface was defined with two goals in mind:

- to simplify the communication between the Prolog system and the relational database engines as much as possible, and
- to give as much flexibility as possible to the overall system. This includes simultaneous access to several databases, allowing both the databases and clients to reside on the same physical machine or different machines, and allowing the clients to reside in Win95/NT or Unix machines.

In order to allow the flexibility mentioned above, a client-sever architecture was chosen. The following figure depicts the overall architecture of the system:



At the server side, a "database mediator server" connects on one side to the databases using the ODBC interface (this interface is available for the databases of the RadioWeb project, as well as for the majority of the databases running in the Win95/NT operating systems) and on the other it is connected to the network by TCP/IP using a fixed socket number / service (currently fixed to socket number 2020).

The mediator server must run on the Windows (NT/95) operating system, on the machine where the databases are also running. The (Prolog) clients which connect to it can be run locally at the server machine. In addition, remote clients running on different machines can also connect to the mediator server by connecting to its socket number (service). Such clients can run on either Unix or Windows systems.

After the connection is established a client can send commands to the mediator server which will pass them to the corresponding database server, and then the data will traverse in the opposite direction. These messages include logging on and off from the database, sending SQL queries, and receiving the responses.

The low level implementation of the current library is accomplished by providing several abstraction levels over the socket interface library of the Prolog engine. These layers of abstraction implement the persistent predicate view, build the appropriate commands for the database using a translator of Prolog goals to SQL commands, issue such commands using the mediator send/receive procedures, parse the responses, and present such responses to the Prolog engine via backtracking.

105.2 Example(s)

```
:- include(library('persdb_sql/persdb_sql')).
:- use_module(library(format)).

%% ------------------------------------------------- %%
%% First sample database : 'SA 6.0 Sample' %%
%% ------------------------------------------------- %%

%% Declare product/4 a persistent predicate, storage in 'radiowebdb':
:- sql_persistent(product( int,    int, string, string ),
               product( quantity,   id,      name,   size   ),
               sampledb).
sql_persistent_location(sampledb, %% The 'sampledb' descriptor:
         db('ASA 6.0 Sample','dba', 'sql','r2d5.dia.fi.upm.es':2020)).

main0 :-
%% Prints the contents of the relation 'product/4' by backtracking over it:
         format("Printing table:\n",[]),
         product(Quantity, Id, Name, Size),
         format("Tuple: ~w \t ~w \t ~w \t ~w \n",
                [Quantity, Id, Name, Size]),
         fail.
main0 :-
         format("Done.\n",[]).

%% Generalizing table printing:
%% Prints the contents of Pred by backtracking over it
print_predicate(Pred):-
        format("Printing relation:\n",[]),
        Pred, %% predicate call
        Pred=..[_|Args],
        format("\t Tuple: ~w \n",[Args]),
        fail.
print_predicate(_Pred):-
        format("Done.~n",[]).

main1 :- %% similar to main0:
%% Prints the contents of the relation 'product/4' by backtracking over it:
        print_predicate(product(_Quantity,_Id,_Name,_Size)).
```

```
main2 :- %% Issues a complex query inside a db_findall:
         dbfindall(sampledb,
                   foo(Quantity, Id, Name, Size, Bar),
                   ( product(Quantity, Id, Name, Size),
                     I^N^S^(Bar is avg(Q,product(Q, I, N, S)))
                   ),
                   Results),
         format("Results = ~w~n",[Results]).
main21 :- %% Using db_call
         dbcall(sampledb,
              (product(Quantity, Id, Name, Size),
               product(Quantity, Id1, _Name1, _Size1),Id1>Id)),
%%       product(Quantity, Id, Name, Size),
%%          product(Quantity, Id1, Name1, Size1),
%%          Id1>Id,
         format("Tuple: ~w \t ~w \t ~w \t ~w \n",
                [Quantity, Id, Name, Size]).

main22 :-  % get the set of tables from a database
        sql_get_tables(sampledb,TablesList),
        display(TablesList).

main23 :- % get table attributes and its types
        sql_table_types(sampledb, 'Customer', AttList),
        display(AttList).

%% ------------------------------------------------ %%
%% Second sample database : 'Literature'            %%
%% ------------------------------------------------ %%

:- sql_persistent(authors(string,string,int),
               authors(firstName,lastName,id), %% 'id' is the primary key
               literature_db).
sql_persistent_location(literature_db,
        db('Literature','dba','sql','r2d5.dia.fi.upm.es':2020)).

main3:- %% Prints the contents of authors
         print_predicate(authors(_FirstName,_LastName,_Id)) .

main4:- %% assert a persistent fact
         dbassertz_fact(authors('Pedro','Calderon de la Barca',17)).

main6:- %% retract a persistent fact
         dbretract_fact(authors('Pedro','Calderon de la Barca',17)).


main9:- %% checking if a persistent fact is a current fact
        dbcurrent_fact(authors(_X,_Y,_Z)).
```

105.3 Usage and interface (persdbrtsql)

- **Library usage:**

  Typically, this library is used including the 'persdb_sql' package into the package list of the module, or using the `use_package/1` declaration:

  In a module:

  ```
  :- module(bar, [main/1], [persdb_sql]).
  ```
  or
  ```
  :- module(bar, [main/1]).
  :- include(library(persdb_sql)).
  ```

  In a *user* file:

  ```
  :- use_package([persdb_sql]).
  ```
  or
  ```
  :- include(library(persdb_sql)).
  ```

  This loads the run-time and compile-time versions of the library (`persdbtr_sql.pl` and `persdbrt_sql.pl`) and includes some needed declarations.
- **Exports:**
  - *Predicates:*

    `init_sql_persdb/0`, `dbassertz_fact/1`, `dbretract_fact/1`, `dbcurrent_fact/1`, `dbretractall_fact/1`, `make_sql_persistent/3`, `dbfindall/4`, `dbcall/2`, `sql_query/3`, `sql_get_tables/2`, `sql_table_types/3`.
  - *Multifiles:*

    `sql_persistent_location/2`.
- **Other modules used:**
  - *System library modules:*

    `persdb_sql/db_client`, `persdb_sql/pl2sql`, `persdb_sql/sqltypes`, `dynamic`, `terms`, `metaterms`, `messages`, `lists`, `aggregates`, `persdb_sql/insert_compiler/pl2sqlinsert`.

105.4 Documentation on exports (persdbrtsql)

init_sql_persdb/0: PREDICATE

Usage:

- *Description:* Internal predicate, used to transform predicates statically declared as persistent (see `sql_persistent/3`) into real persistent predicates.

sql_persistent/3: PREDICATE

No further documentation available for this predicate.

dbassertz_fact/1: PREDICATE

Usage: `dbassertz_fact(+Fact)`

- *Description:* Persistent extension of `assertz_fact/1`: the current instance of `Fact` is interpreted as a fact (i.e., a relation tuple) and is added to the end of the definition of the corresponding predicate. If any integrity constraint violation is done (database stored predicates), an error will be displayed. The predicate concerned must be statically (`sql_persistent/3`) or dinamically (`make_sql_persistent/3`) declared. Any uninstantiated variables in the `Fact` will be replaced by new, private variables. **Note:** *assertion of facts with uninstantiated variables not implemented at this time.*
- *Call and exit should be compatible with:*

  `+Fact` is a fact (a term whose main functor is not `':-'/2`). (`persdbrtsql:fact/1`)

dbretract_fact/1: PREDICATE

Usage: `dbretract_fact(+Fact)`

- *Description:* Persistent extension of `retract_fact/1`: deletes on backtracking all the facts which unify with `Fact`. The predicate concerned must be statically (`sql_persistent/3`) or dinamically (`make_sql_persistent/3`) declared.
- *Call and exit should be compatible with:*

  `+Fact` is a fact (a term whose main functor is not `':-'/2`). (`persdbrtsql:fact/1`)

dbcurrent_fact/1: PREDICATE

Usage: `dbcurrent_fact(+Fact)`

- *Description:* Persistent extension of `current_fact/1`: the fact `Fact` exists in the current database. The predicate concerned must be declared `sql_persistent/3`. Provides on backtracking all the facts (tuples) which unify with `Fact`.
- *Call and exit should be compatible with:*

  `+Fact` is a fact (a term whose main functor is not `':-'/2`). (`persdbrtsql:fact/1`)

dbretractall_fact/1: PREDICATE

Usage: `dbretractall_fact(+Fact)`

- *Description:* Persistent extension of `retractall_fact/1`: when called deletes all the facts which unify with `Fact`. The predicate concerned must be statically (`sql_persistent/3`) or dinamically (`make_sql_persistent/3`) declared.
- *Call and exit should be compatible with:*

  `+Fact` is a fact (a term whose main functor is not `':-'/2`). (`persdbrtsql:fact/1`)

make_sql_persistent/3: PREDICATE

Meta-predicate with arguments: `make_sql_persistent(addmodule,?,?)`.

Usage: `make_sql_persistent(PrologPredTypes,TableAttributes,Keyword)`

- *Description:* Dynamic version of the `sql_persistent/3` declaration.
- *The following properties should hold upon exit:*

  `PrologPredTypes` is a structure describing a Prolog predicate name with its types. (`persdbrtsql:prologPredTypes/1`)

  `TableAttributes` is a structure describing a table name and some attributes. (`persdbrtsql:tableAttributes/1`)

  `Keyword` is the name of a persistent storage location. (`persdbrtsql:persLocId/1`)

dbfindall/4: PREDICATE

Usage: `dbfindall(+DBId,+Pattern,+ComplexGoal,-(Results))`

- *Description:* Similar to `findall/3`, but `Goal` is executed in database `DBId`. Certain restrictions and extensions apply to both `Pattern` and `ComplexGoal` stemming from the Prolog to SQL translation involved (see the corresponding type definitions for details).
- *Call and exit should be compatible with:*

  `+DBId` a unique identifier of a database session connection. (`db_client:dbconnection/1`)

  `+Pattern` is a database projection term. (`pl2sql:projterm/1`)

  `+ComplexGoal` is a database query goal. (`pl2sql:querybody/1`)

  `-(Results)` is a list. (`basic_props:list/1`)

dbcall/2: PREDICATE

Usage: `dbcall(+DBId,+ComplexGoal)`

- *Description:* Internal predicate, used by the transformed versions of the persistent predicates. Not meant to be called directly by users. It is exported by the library so that it can be used by the transformed versions of the persistent predicates in the modules in which they reside. Sends `ComplexGoal` to database `DBId` for evaluation. `ComplexGoal` must be a call to a persistent predicate which resides in database `DBId`.
- *Call and exit should be compatible with:*

  `+DBId` a unique identifier of a database session connection. (`db_client:dbconnection/1`)

  `+ComplexGoal` is a database query goal. (`pl2sql:querybody/1`)

sql_query/3: PREDICATE

Usage: `sql_query(+DBId,+SQLString,AnswerTableTerm)`

- *Description:* `ResultTerm` is the response from database `DBId` to the SQL query in `SQLString` to database `DBId`. `AnswerTableTerm` can express a set of tuples, an error answer or a 'ok' response (see `answertableterm/1` for details). At the moment, `sql_query/3` log in and out for each query. This should be changed to log in only the first time and log out on exit and/or via a timer in the standard way.
- *Call and exit should be compatible with:*

  `+DBId` a unique identifier of a database session connection. (`db_client:dbconnection/1`)

  `+SQLString` is a string containing SQL code. (`pl2sql:sqlstring/1`)

  `AnswerTableTerm` is a response from the ODBC database interface. (`db_client:answertableterm/1`)

sql_get_tables/2: PREDICATE

Usage 1: `sql_get_tables(+Location,-(Tables))`

- *Description:* `Tables` contains the tables available in `Location`.
- *Call and exit should be compatible with:*

  `persdbrtsql:persLocation(+Location)` (`persdbrtsql:persLocation/1`)

  `-(Tables)` is a list of `atms`. (`basic_props:list/2`)

Usage 2: `sql_get_tables(+DbConnection,-(Tables))`

- *Description:* `Tables` contains the tables available in `DbConnection`.
- *Call and exit should be compatible with:*

  `+DbConnection` a unique identifier of a database session connection. (`db_client:dbconnection/1`)

  `-(Tables)` is a list of `atm`s. (`basic_props:list/2`)

sql_table_types/3: PREDICATE

Usage 1: `sql_table_types(+Location,+Table,-(AttrTypes))`

- *Description:* `AttrTypes` are the attributes and types of `Table` in `Location`.
- *Call and exit should be compatible with:*

  `persdbrtsql:persLocation(+Location)` (`persdbrtsql:persLocation/1`)

  `+Table` is an atom. (`basic_props:atm/1`)

  `-(AttrTypes)` is a list. (`basic_props:list/1`)

Usage 2: `sql_table_types(+DbConnection,+Table,-(AttrTypes))`

- *Description:* `AttrTypes` are the attributes and types of `Table` in `DbConnection`.
- *Call and exit should be compatible with:*

  `+DbConnection` a unique identifier of a database session connection. (`db_client:dbconnection/1`)

  `+Table` is an atom. (`basic_props:atm/1`)

  `-(AttrTypes)` is a list. (`basic_props:list/1`)

socketname/1: REGTYPE

Usage: `socketname(IPP)`

- *Description:* `IPP` is a structure describing a complete TCP/IP port address.

dbname/1: REGTYPE

Usage: `dbname(DBId)`

- *Description:* `DBId` is the identifier of an ODBC database.

user/1: REGTYPE

Usage: `user(User)`

- *Description:* `User` is a user name in the ODBC database.

passwd/1: REGTYPE

Usage: `passwd(Passwd)`

- *Description:* `Passwd` is the password for the user name in the ODBC database.

projterm/1: REGTYPE

Usage: `projterm(DBProjTerm)`

– *Description:* `DBProjTerm` is a database projection term.

querybody/1: REGTYPE

Usage: `querybody(DBGoal)`

– *Description:* `DBGoal` is a database query goal.

sqltype/1: (UNDOC_REEXPORT)

Imported from `sqltypes` (see the corresponding documentation for details).

105.5 Documentation on multifiles (`persdbrtsql`)

sql_persistent_location/2: PREDICATE

Relates names of locations (the `Keywords`) with descriptions of such locations (`Locations`).

The predicate is *multifile*.

The predicate is of type *data*.

Usage 1: `sql_persistent_location(Keyword,Location)`

– *Description:* `Keyword` is an identifier for the persistent data location `Location`.

– *Call and exit should be compatible with:*

`Keyword` is the name of a persistent storage location. (`persdbrtsql:persLocId/1`)

`persdbrtsql:persLocation(Location)` (`persdbrtsql:persLocation/1`)

Usage 2: `sql_persistent_location(Keyword,DBLocation)`

– *Description:* In this usage, `DBLocation` is a *relational database*, in which case the predicate is stored as tuples in the database.

– *The following properties should hold upon exit:*

`Keyword` is the name of a persistent storage location. (`persdbrtsql:persLocId/1`)

`DBLocation` is a structure describing a database. (`persdbrtsql:database_desc/1`)

105.6 Documentation on internals (`persdbrtsql`)

db_query/4: PREDICATE

Usage: `db_query(+DBId,+ProjTerm,+Goal,ResultTerm)`

– *Description:* `ResultTerm` contains all the tuples which are the response from database `DBId` to the Prolog query `Goal`, projected onto `ProjTerm`. Uses `pl2sqlstring/3` for the Prolog to SQL translation and `sql_query/3` for posing the actual query.

– *Call and exit should be compatible with:*

`+DBId` a unique identifier of a database session connection. (`db_client:dbconnection/1`)

`+ProjTerm` is a database projection term. (`pl2sql:projterm/1`)

`+Goal` is a database query goal. (`pl2sql:querybody/1`)

`ResultTerm` is a tuple of values from the ODBC database interface. (`db_client:tuple/1`)

db_query_one_tuple/4: PREDICATE

Usage: `db_query_one_tuple(+DBId,+ProjTerm,+Goal,ResultTerm)`

- *Description:* `ResultTerm` is one of the tuples which are the response from database `DBId` to the Prolog query `Goal`, projected onto `ProjTerm`. Uses `pl2sqlstring/3` for the Prolog to SQL translation and `sql_query_one_tuple/3` for posing the actual query. After last tuple has been reached, a null tuple is unified with ResultTerm, and the connection to the database finishes.
- *Call and exit should be compatible with:*

  `+DBId` a unique identifier of a database session connection. (`db_client:dbconnection/1`)

  `+ProjTerm` is a database projection term. (`pl2sql:projterm/1`)

  `+Goal` is a database query goal. (`pl2sql:querybody/1`)

  `ResultTerm` is a predicate containing a tuple. (`db_client:answertupleterm/1`)

sql_query_one_tuple/3: PREDICATE

Usage: `sql_query_one_tuple(+DBId,+SQLString,ResultTuple)`

- *Description:* `ResultTuple` contains an element from the set of tuples which represents the response in `DBId` to the SQL query `SQLString`. If the connection is kept, succesive calls return consecutive tuples, until the last tuple is reached. Then a null tuple is unified with `ResultTuple` and the connection is finished (calls to `db_logoff/1` and `odbc_disconnect/1`).
- *Call and exit should be compatible with:*

  `+DBId` a unique identifier of a database session connection. (`db_client:dbconnection/1`)

  `+SQLString` is a string containing SQL code. (`pl2sql:sqlstring/1`)

  `ResultTuple` is a tuple of values from the ODBC database interface. (`db_client:tuple/1`)

dbconnection/1: REGTYPE

Usage: `dbconnection(H)`

- *Description:* `H` a unique identifier of a database session connection.

tuple/1: REGTYPE

Usage: `tuple(T)`

- *Description:* `T` is a tuple of values from the ODBC database interface.

105.7 Known bugs and planned improvements (`persdbrtsql`)

- At least in the shell, reloading a file after changing the definition of a persistent predicate does not eliminate the old definition...
- Functionality missing: some questions need to be debugged.
- Warning: still using kludgey string2term and still using some non-uniquified temp files.
- Needs to be unified with the file-based library.

106 Prolog to SQL translator

Author(s): C. Draxler. Adapted by M. Hermenegildo and I. Caballero.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 0.9#96 (1999/5/21, 19:53:48 MEST)

This library performs translation of Prolog queries into SQL. The code is an adaptation for Ciao of the Prolog to SQL compiler written by Christoph Draxler, CIS Centre for Information and Speech Processing, Ludwig-Maximilians-University Munich, `draxler@cis.uni-muenchen.de`, Version 1.1. Many thanks to Christoph for allowing us to include this adaptation of his code with Ciao.

The translator needs to know the correspondence between Prolog predicates and the SQL tables in the database. To this end this module exports two multifile predicates, `relation/3` and `attribute/4`. See the description of these predicates for details on how such correspondance is specified.

The main entry points to the translator are `pl2sqlstring/3` and `pl2sqlterm/3`. Details on the types of queries allowed can be found in the description of these predicates.

Example: the following program would print out a term representing the SQL query corresponding to the given Prolog query:

```
:- use_module(library('persdb_sql/pl2sql')).
:- use_module(library(strings)).

:- multifile [relation/3,attribute/4].
:- data [relation/3,attribute/4].

relation(product,3,'PRODUCT').
attribute(1,'PRODUCT','ID',int).
attribute(2,'PRODUCT','QUANTITY',int).
attribute(3,'PRODUCT','NAME',string).

main :-
     pl2sqlstring( f(L,K),
          ((product(L,N,a); product(L,N,b)),
           \+ product(2,3,b),
           L + 2 > avg(Y, Z^product(Z,Y,a)),
           K is N + max(X, product(X,2,b))
           ), T),
     write_string(T).

%%     printqueries(T).
```

Note: while the translator can be used directly in programs, it is more convenient to use a higher-level abstraction: persistent predicates (implemented in the `persdb` library). The notion of persistent predicates provides a completely transparent interface between Prolog and relational databases. When using this library, the Prolog to SQL translation is called automatically as needed.

106.1 Usage and interface (pl2sql)

- **Library usage:**
  `:- use_module(library(pl2sql)).`
- **Exports:**
  - *Predicates:*
    `pl2sqlstring/3`, `pl2sqlterm/3`, `printqueries/1`, `sqlterm2string/2`.
  - *Regular Types:*
    `querybody/1`, `projterm/1`, `sqlstring/1`.
  - *Multifiles:*
    `relation/3`, `attribute/4`.
- **Other modules used:**
  - *System library modules:*
    `persdb_sql/sqltypes`, `write`, `read`, `dec10_io`, `system`, `iso_misc`, `lists`, `aggregates`, `messages`.

106.2 Documentation on exports (pl2sql)

pl2sqlstring/3: PREDICATE

Usage: `pl2sqlstring(+ProjectionTerm,+DatabaseGoal,-(SQLQueryString))`

- *Description:* This is the top level predicate which translates complex Prolog goals into the corresponding SQL code.

  The query code is prepared in such a way that the result is projected onto the term `ProjectionTerm` (also in a similar way to the first argument of `setof/3`)). See the predicate `translate_projection/3` for restrictions on this term.

  `SQLQueryString` contains the code of the SQL query, ready to be sent to an SQL server.
- *Call and exit should be compatible with:*

  `+ProjectionTerm` is a database projection term. (`pl2sql:projterm/1`)

  `+DatabaseGoal` is a database query goal. (`pl2sql:querybody/1`)

  `-(SQLQueryString)` is a string containing SQL code. (`pl2sql:sqlstring/1`)

querybody/1: REGTYPE

`DBGoal` is a goal meant to be executed in the external database. It can be a complex term containing conjunctions, disjunctions, and negations, of:

- Atomic goals, which must have been defined via `relation/3` and `attribute/4` and reside in the (same) database. Their arguments must be either ground or free variables. If they are ground, they must be bound to constants of the type declared for that argument. If an argument is a free variable, it may *share* with (i.e., be the same variable as) other free variables in other goal arguments.
- Database comparison goals, whose main functor must be a database comparison operator (see `pl2sql: comparison/2`) and whose arguments must be *database arithmetic expressions.*

- Database calls to is/2. The left side of such a call may be either unbound, in which case it is bound to the result of evaluating the right side, or bound in which case an equality condition is tested. The right side must be a *database arithmetic expression*.

The binding of variables follows Prolog rules:

- variables are bound by positive base goals and on the left side of the `is/2` predicate.
- Comparison operations, negated goals, and right sides of the `is/2` predicate do not return variable bindings and may even require all arguments to be bound for a safe evaluation.

Database arithmetic expressions may contain:

- Numeric constants (i.e., integers, reals, etc.).
- Bound variables, i.e., variables which will be bound during execution through occurrence within a positive database goal, or by a preceding arithmetic function.
- Database arithmetic functions, which are a subset of those typically accepted within `is/2` (see `pl2sql`: `arithmetic_functor/2`).
- Database aggregation functions, each of which has two arguments: a variable indicating the argument over which the function is to be computed, and a goal argument which must contain in at least one argument position the variable (e.g. `avg(Seats,plane(Type,Seats))`). The goal argument may only be a conjunction of (positive or negative) base goals. See `pl2sql`: `aggregate_functor/2` for the admissible aggregate functions.

In addition, variables can be existentially quantified using `^/2` (in a similar way to how it is done in `setof/3`).

Note that it is assumed that the arithmetic operators in Prolog and SQL are the same, i.e., `+` is addition in Prolog and in SQL, etc.

Usage: `querybody(DBGoal)`

– *Description:* `DBGoal` is a database query goal.

projterm/1: REGTYPE

`DBProjTerm` is a term onto which the result of a database query code is (in a similar way to the first argument of `setof/3`)).

A `ProjectionTerm` must meet the following restrictions:

- The functor of `ProjectionTerm` may not be one of the built-in predicates, i.e. ',', ';', etc. are not allowed.
- Only variables and constants are allowed as arguments, i.e., no structured terms may appear.

Usage: `projterm(DBProjTerm)`

– *Description:* `DBProjTerm` is a database projection term.

sqlstring/1: REGTYPE

```
sqlstring(S) :-
        string(S).
```

Usage: `sqlstring(S)`

– *Description:* `S` is a string containing SQL code.

pl2sqlterm/3: PREDICATE

Usage: `pl2sqlterm(+ProjectionTerm,+DatabaseGoal,-(SQLQueryTerm))`

- *Description:* Similar to `pl2sqlstring/3` except that `SQLQueryTerm` is a representation of the SQL query as a Prolog term.
- *Call and exit should be compatible with:*

  `+ProjectionTerm` is a database projection term. (`pl2sql:projterm/1`)

  `+DatabaseGoal` is a database query goal. (`pl2sql:querybody/1`)

  `-(SQLQueryTerm)` is a list of `sqlterm`s. (`basic_props:list/2`)

printqueries/1: PREDICATE

Usage: `printqueries(SQLTermList)`

- *Description:* Print to standard output in SQL the list of SQL term `SQLTermList`.
- *Call and exit should be compatible with:*

  `SQLTermList` is a list of `sqlterm`s. (`basic_props:list/2`)

sqlterm2string/2: PREDICATE

Usage: `sqlterm2string(+Queries,-(QueryString))`

- *Description:* `QueryString` is a string representation of the list of queries in Prolog-term format in `Queries`.
- *Call and exit should be compatible with:*

  `+Queries` is a list of `sqlterm`s. (`basic_props:list/2`)

  `-(QueryString)` is a string containing SQL code. (`pl2sql:sqlstring/1`)

sqltype/1: (UNDOC_REEXPORT)

Imported from `sqltypes` (see the corresponding documentation for details).

106.3 Documentation on multifiles (`pl2sql`)

relation/3: PREDICATE

The predicate is *multifile*.

The predicate is of type *data*.

Usage: `relation(PredName,Arity,TableName)`

- *Description:* This predicate, together with `attribute/4`, defines the correspondence between Prolog predicates and the SQL tables in the database. These two relations constitute an extensible meta-database which maps Prolog predicate names to SQL table names, and Prolog predicate argument positions to SQL attributes.

  `PredName` is the chosen Prolog name for an SQL table. `Arity` is the number of arguments of the predicate. `TableName` is the name of the SQL table in the Database Management System.
- *Call and exit should be compatible with:*

  `PredName` is an atom. (`basic_props:atm/1`)

  `Arity` is an integer. (`basic_props:int/1`)

  `TableName` is an atom. (`basic_props:atm/1`)

attribute/4: PREDICATE

The predicate is *multifile*.

The predicate is of type *data*.

Usage: `attribute(ANumber,TblName,AName,AType)`

– *Description:* This predicate maps the argument positions of a Prolog predicate to the SQL attributes of its corresponding table. The types of the arguments need to be specified, and this information is used for consistency checking during the translation and for output formatting. A minimal type system is provided to this end. The allowable types are given by `sqltype/1`.

  `ANumber` is the argument number in the Prolog relation. `TblName` is the name of the SQL table in the Database Management System. `AName` is the name of the corresponding attribute in the table. `AType` is the (translator) data type of the attribute.

– *Call and exit should be compatible with:*

  `ANumber` is an integer. (`basic_props:int/1`)

  `TblName` is an atom. (`basic_props:atm/1`)

  `AName` is an atom. (`basic_props:atm/1`)

  `AType` is an SQL data type supported by the translator. (`sqltypes:sqltype/1`)

106.4 Documentation on internals (pl2sql)

query_generation/3: PREDICATE

Usage:

`query_generation(+ListOfConjunctions,+ProjectionTerm,-(ListOfQueries))`

– *Description:* For each Conjunction in `ListOfConjunctions`, translate the pair `(ProjectionTerm,Conjunction)` to an SQL query and connect each such query through a UNION-operator to result in the `ListOfQueries`.

  A Conjunction consists of positive or negative subgoals. Each subgoal is translated as follows:

  - the functor of a goal that is not a comparison operation is translated to a relation name with a range variable,
  - negated goals are translated to NOT EXISTS-subqueries with * projection,
  - comparison operations are translated to comparison operations in the WHERE-clause,
  - aggregate function terms are translated to aggregate function (sub)queries.

  The arguments of a goal are translated as follows:

  - `variables of a goal` are translated to qualified attributes,
  - variables occurring in several goals are translated to equality comparisons (equi join) in the WHERE-clause,
  - constant arguments are translated to equality comparisons in the WHERE-clause.

  Arithmetic functions are treated specially (`translate_arithmetic_function/5`). See also `querybody/1` for details on the syntax accepted and restrictions.

translate_conjunction/5: PREDICATE

Usage: `translate_conjunction(Conjunction,SQLFrom,SQLWhere,Dict,NewDict)`

– *Description:* Translates a conjunction of goals (represented as a list of goals preceeded by existentially quantified variables) to FROM-clauses and WHERE-clauses of an SQL query. A dictionary containing the associated SQL table and attribute names is built up as an accumulator pair (arguments `Dict` and `NewDict`).

translate_goal/5: PREDICATE

Usage: `translate_goal(Goal,SQLFrom,SQLWhere,Dict,NewDict)`

– *Description:* Translates:

- a positive database goal to the associated FROM- and WHERE clause of an SQL query,
- a negated database goal to a negated existential subquery,
- an arithmetic goal to an arithmetic expression or an aggregate function query,
- a comparison goal to a comparison expression, and
- a negated comparison goal to a comparison expression with the opposite comparison operator.

translate_arithmetic_function/5: PREDICATE

Usage:
`translate_arithmetic_function(Result,Expression,SQLWhere,Dict,NewDict)`

– *Description:* Arithmetic functions (left side of is/2 operator is bound to value of expression on right side) may be called with either:

- `Result` unbound: then `Result` is bound to the value of the evaluation of `Expression`,
- `Result` bound: then an equality condition is returned between the value of `Result` and the value of the evaluation of `Expression`.

  Only the equality test shows up in the WHERE clause of an SQLquery.

translate_comparison/5: PREDICATE

Usage: `translate_comparison(LeftArg,RightArg,CompOp,Dict,SQLComparison)`

– *Description:* Translates the left and right arguments of a comparison term into the appropriate comparison operation in SQL. The result type of each argument expression is checked for type compatibility.

aggregate_function/3: PREDICATE

Usage:
`aggregate_function(AggregateFunctionTerm,Dict,AggregateFunctionQuery)`

– *Description:* Supports the Prolog aggregate function terms listed in `aggregate_functor/2` within arithmetic expressions. Aggregate functions are translated to the corresponding SQL built-in aggregate functions.

comparison/2: PREDICATE

Usage: `comparison(PrologOperator,SQLOperator)`

– *Description:* Defines the mapping between Prolog operators and SQL operators:

```
comparison(=,=).
comparison(<,<).
comparison(>,>).
comparison(@<,<).
comparison(@>,>).
```

– *Call and exit should be compatible with:*

`PrologOperator` is an atom. (`basic_props:atm/1`)

`SQLOperator` is an atom. (`basic_props:atm/1`)

negated_comparison/2: PREDICATE

Usage: `negated_comparison(PrologOperator,SQLOperator)`

– *Description:* Defines the mapping between Prolog operators and the complementary SQL operators:

```
negated_comparison(=,<>).
negated_comparison(\==,=).
negated_comparison(>,=<).
negated_comparison(=<,>).
negated_comparison(<,>=).
negated_comparison(>=,<).
```

– *Call and exit should be compatible with:*

`PrologOperator` is an atom. (`basic_props:atm/1`)

`SQLOperator` is an atom. (`basic_props:atm/1`)

arithmetic_functor/2: PREDICATE

Usage: `arithmetic_functor(PrologFunctor,SQLFunction)`

– *Description:* Defines the admissible arithmetic functions on the Prolog side and their correspondence on the SQL side:

```
arithmetic_functor(+,+).
arithmetic_functor(-,-).
arithmetic_functor(*,*).
arithmetic_functor(/,/).
```

– *Call and exit should be compatible with:*

`PrologFunctor` is an atom. (`basic_props:atm/1`)

`SQLFunction` is an atom. (`basic_props:atm/1`)

aggregate_functor/2: PREDICATE

Usage: `aggregate_functor(PrologFunctor,SQLFunction)`

– *Description:* Defines the admissible aggregate functions on the Prolog side and their correspondence on the SQL side:

```
aggregate_functor(avg,'AVG').
aggregate_functor(min,'MIN').
aggregate_functor(max,'MAX').
aggregate_functor(sum,'SUM').
aggregate_functor(count,'COUNT').
```

– *Call and exit should be compatible with:*
`PrologFunctor` is an atom. (`basic_props:atm/1`)
`SQLFunction` is an atom. (`basic_props:atm/1`)

106.5 Known bugs and planned improvements (`pl2sql`)

- Need to separate db predicate names by module.
- Warning: using kludgy SQL term to string conversion because of bugs in sqlterm2string/2.

107 Low-level socket interface to SQL/ODBC databases

Author(s): D. Cabeza, M. Carro, I. Caballero, and M. Hermenegildo..

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.3#53 (1999/9/15, 23:37:55 MEST)

This library provides a socket-based interface to SQL Databases, using the database mediator server ODBC interface developed by C. Taboch and I. Caballero. The interface currently works for databases running in Win95/NT machines via ODBC. This low-level interface was defined with two goals in mind:

- To simplify the communication between the Prolog system and the relational database engines as much as possible.
- To give as much flexibility to the overall system. This includes supporting simultaneous access to several databases, and also allowing both the databases and clients to reside on the same or different physical Win95/NT machines. The clients can reside also in Unix/Linux machines. It is also possible to access database selections in either tuple at a time or set at a time fashion, as needed.

In order to allow the flexibility mentioned above, a socket (TCP/IP) client-server architecture was chosen. The interface has two main components:

- At the server side, a *database mediator server* connects on one side to the databases using the ODBC interface and on the other to a fixed port number (service), on which the mediator server listens for requests. The source code (and executables) for this server (slightly modified versions of C. Taboch original code) are in the same directory as this library. This executable must be running on the Windows (NT/95) machine where the databas(es) is(are) also running.[1]
- At the client side a Prolog client can connect to the database by loading this library and calling the appropriate predicates. The Prolog client can run on either Windows or Unix/Linux systems, locally at the server machine or remotely in different machines. The mediator server port number (service) used is currently fixed to 2020. After the connection is established a client can send commands to the mediator server which will pass them to the corresponding database server, and then the response data will return in the opposite direction. These messages include login on and off from the database, sending SQL queries, and receiving the responses.

Example:

```
:- use_module(library('persdb_sql/db_client')).
:- use_module(library(format)).
:- use_module(library(lists)).

:- multifile issue_debug_messages/1.
:- data issue_debug_messages/1.
issue_debug_messages('db_client').

main0:- %% getting the tables existing in a database
        odbc_connect('r2d5.dia.fi.upm.es':2020,Stream),
        db_login(Stream,'DaletDemo',dalet_admin,dalet_admin,
```

[1] The server can be downloaded from the same site(s) as the Ciao system. The downloaded package contains complete installation instructions for the server. Before trying to run the Ciao Prolog side of things, make sure that the server is correctly installed and running, and that it can be accessed from the small sample client that comes with the server distribution. Once this is achieved, connection from Ciao should not pose problems.

```
%%        db_login(Stream,'ASA 6.0 Sample',dba,sql,
                 dbconnection(Stream,DbHandle)),
%%        db_get_tables(dbconnection(Stream,DbHandle),TablesList),
        db_table_types(dbconnection(Stream,DbHandle),'Titles',AttList),
        db_logoff(dbconnection(Stream,DbHandle)),
        odbc_disconnect(Stream),
        format("Results: ~w \n",AttList).


main1 :- %% accessing the whole table in one go
        odbc_connect('r2d5.dia.fi.upm.es':2020,Stream),
        db_login(Stream,'ASA 6.0 Sample',dba,sql,Conn),
        db_eval_sql(Conn,"SELECT fname,lname,address from ""DBA"".customer
                   WHERE ((Id>100) AND (Id<105))",Term),
        write(Term), nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,
        functor(Term,X,_Y),
        write(X),nl,
        Term=..[_|L],
        write('That is L'),nl,
        write(L),nl,L=[M],length(M,N),write(N),
%        db_eval_sql(Conn,"SELECT id,name,description,color from
%                     ""DBA"".product WHERE size='One size fits all'
%                     OR size='Large'",Term2),
%        write(Term2), nl,
        db_logoff(Conn),
        odbc_disconnect(Stream).

main2:- %% accessing tuples one by one
        odbc_connect('r2d5.dia.fi.upm.es':2020,Stream),
        db_login(Stream,'ASA 6.0 Sample',dba,sql,Conn),
        db_stmt_handle(Conn,
              "SELECT fname,lname,address from ""DBA"".customer
               WHERE ((Id>100) AND (Id<105))",QueryConn),
        db_one_tuple(QueryConn,Answer),
        format("First Tuple: ~w \n",Answer),
        db_one_tuple(QueryConn,Answer2),
        format("Second Tuple: ~w \n",Answer2),
        db_logoff(Conn),
        odbc_disconnect(Stream).

main3:- %% bringing tuples (one to one) from Literature Database
        %% TO SEE : check it
        odbc_connect('r2d5.dia.fi.upm.es':2020,Stream),
        db_login(Stream,'Literature',dba,sql,Conn),
        db_stmt_handle(Conn,"SELECT * FROM AUTHORS ORDER BY ID",QueryConn),
        print_all_one_by_one(QueryConn),
        db_logoff(Conn),
        odbc_disconnect(Stream).

print_all_one_by_one(QueryConn):-
        db_one_tuple(QueryConn,Answer),
```

```
        Answer\=[],
        !,
        format(" ~w \n",Answer),
        print_all_one_by_one(QueryConn).
print_all_one_by_one(_QueryConn):-
        write(' Finished fetching query statement.'),
        nl.

main4:- %% inserting a tuple into the Literature database
        odbc_connect('r2d5.dia.fi.upm.es':2020,Stream),
        db_login(Stream,'Literature',dba,sql,Conn),
        db_stmt_handle(Conn,
               "INSERT INTO AUTHORS values ('Martin Gaite',
                'Carmen',10,'1910-7-10 00:00:00:000')",_QueryConn),
        db_logoff(Conn),
        odbc_disconnect(Stream).

main5:- %% creating a view
        odbc_connect('r2d5.dia.fi.upm.es':2020,Stream),
        db_login(Stream,'Literature',dba,sql,Conn),
        db_stmt_handle(Conn,
               "CREATE VIEW AntoniosDateOfBirth AS SELECT
                LastName,Date_of_birth FROM
                AUTHORS where FirstName='Antonio';",_QueryConn),
        db_logoff(Conn),
        odbc_disconnect(Stream).

main6:- %% printing the view
        odbc_connect('r2d5.dia.fi.upm.es':2020,Stream),
        db_login(Stream,'Literature',dba,sql,Conn),
        db_eval_sql(Conn,"select * from AntoniosDateOfBirth",Term),
        write(Term),
        db_logoff(Conn),
        odbc_disconnect(Stream).

main7:- %% dropping a view
        odbc_connect('r2d5.dia.fi.upm.es':2020,Stream),
        db_login(Stream,'Literature',dba,sql,Conn),
        db_eval_sql(Conn,"DROP VIEW AntoniosDateOfBirth",Term),
        write(Term),
        db_logoff(Conn),
        odbc_disconnect(Stream).
```

107.1 Usage and interface (db_client)

- **Library usage:**
  `:- use_module(library(db_client)).`
- **Exports:**
  - *Predicates:*
    `odbc_connect/2`, `db_login/5`, `db_eval_sql/3`, `db_stmt_handle/3`, `db_one_tuple/2`, `db_get_tables/2`, `db_table_types/3`, `db_logoff/1`, `odbc_disconnect/1`, `match_string/3`.
  - *Regular Types:*
    `socketname/1`, `dbname/1`, `user/1`, `passwd/1`, `dbconnection/1`, `answertableterm/1`, `tuple/1`, `dbqueryconnection/1`, `answertupleterm/1`.
- **Other modules used:**
  - *System library modules:*
    `sockets/sockets`, `write`, `strings`, `lists`, `persdb_sql/string2term`, `messages`.

107.2 Documentation on exports (db_client)

odbc_connect/2: PREDICATE

Usage: `odbc_connect(+DbAddress,-(Stream))`

- *Description:* Opens a socket connection to port `DbAddress`, which should be the address of a database mediator server. `Stream` is the identifier of the corresponding Prolog stream. It is simply a call to `connect_to_socket/3`
- *Call and exit should be compatible with:*
  `+DbAddress` is a structure describing a complete TCP/IP port address. (`db_client:socketname/1`)
  `-(Stream)` is an open stream. (`streams_basic:stream/1`)

db_login/5: PREDICATE

Usage: `db_login(+Stream,+DbName,+User,+Passwd,-(DbConnection))`

- *Description:* Logs on to the database `DbName` with user `User` and password `Passwd` via connection `Stream`. `DbConnection` contains the identifier of the session, to be used in the calls to other predicates defining the interface. It fails and display an error message if the login is not succesfully completed.
- *Call and exit should be compatible with:*
  `+Stream` is an open stream. (`streams_basic:stream/1`)
  `+DbName` is the identifier of an ODBC database. (`db_client:dbname/1`)
  `+User` is a user name in the ODBC database. (`db_client:user/1`)
  `+Passwd` is the password for the user name in the ODBC database. (`db_client:passwd/1`)
  `-(DbConnection)` a unique identifier of a database session connection. (`db_client:dbconnection/1`)

socketname/1: REGTYPE

```
socketname(IPAddress:PortNumber) :-
        atm(IPAddress),
        int(PortNumber).
```

Usage: `socketname(IPP)`

- *Description:* `IPP` is a structure describing a complete TCP/IP port address.

dbname/1: REGTYPE

```
dbname(DBId) :-
        atm(DBId).
```

Usage: `dbname(DBId)`

- *Description:* `DBId` is the identifier of an ODBC database.

user/1: REGTYPE

```
user(User) :-
        atm(User).
```

Usage: `user(User)`

- *Description:* `User` is a user name in the ODBC database.

passwd/1: REGTYPE

```
passwd(Passwd) :-
        atm(Passwd).
```

Usage: `passwd(Passwd)`

- *Description:* `Passwd` is the password for the user name in the ODBC database.

dbconnection/1: REGTYPE

```
dbconnection(dbconnection(DbStream,DbHandle)) :-
        stream(DbStream),
        dbhandle(DbHandle).
```

Usage: `dbconnection(H)`

- *Description:* `H` a unique identifier of a database session connection.

db_eval_sql/3: PREDICATE

Usage: `db_eval_sql(+DbConnection,+Sentence,-(AnswerTableTerm))`

- *Description:* Evaluates the SQL code in `Sentence` in database session `DbConnection`. `AnswerTableTerm` is the response. If a wrong answer is obtained, `AnswerTableTerm` will indicate the error (see `answertableterm` for details).
- *Call and exit should be compatible with:*

  `+DbConnection` a unique identifier of a database session connection. (`db_client:dbconnection/1`)

  `+Sentence` is a string of SQL code. (`db_client:sqlstring/1`)

  `-(AnswerTableTerm)` is a response from the ODBC database interface. (`db_client:answertableterm/1`)

answertableterm/1: REGTYPE

Represents the types of responses that will be returned from the ODBC database interface. These can be a set of answer tuples, or the atom `ok` in case of a successful addition or deletion.

Usage: `answertableterm(AT)`

- *Description:* `AT` is a response from the ODBC database interface.

tuple/1: REGTYPE

```
tuple(T) :-
        list(T,atm).
```

Usage: `tuple(T)`

- *Description:* `T` is a tuple of values from the ODBC database interface.

db_stmt_handle/3: PREDICATE

Usage: `db_stmt_handle(+DbConnection,+Sentence,-(DbQueryConnection))`

- *Description:* Evaluates the SQL code in `Sentence` in database session `DbConnection`. If `Sentence` is a succesfull selection, `DbQueryConnection` is a *handle* to the set of tuples produced by this selection. The individual members of this set can then be accessed via `DbQueryConnection` using the `db_one_tuple/2` predicate. If `Sentence` is not succesfull or is not a selection, `DbQueryConnection` will contain the answer received.
- *Call and exit should be compatible with:*

  `+DbConnection` a unique identifier of a database session connection. (`db_client:dbconnection/1`)

  `+Sentence` is a string of SQL code. (`db_client:sqlstring/1`)

  `-(DbQueryConnection)` is a unique identifier of a query answer in a database session connection. (`db_client:dbqueryconnection/1`)

dbqueryconnection/1: REGTYPE

```
dbqueryconnection(dbqueryconnection(DbStream,DbHandle,StmtHandle)) :-
        stream(DbStream),
        dbhandle(DbHandle),
        stmthandle(StmtHandle).
```

Usage: `dbqueryconnection(H)`

- *Description:* `H` is a unique identifier of a query answer in a database session connection.

db_one_tuple/2: PREDICATE

Usage: `db_one_tuple(+DbQueryConnection,-(TupleTerm))`

- *Description:* `TupleTerm` represents the last tuple read from handle `DbQueryConnection`. Will be an empty list if all the answers have already been read. It displays an error if any unformatted string is fetched.

- *Call and exit should be compatible with:*
  +DbQueryConnection is a unique identifier of a query answer in a database session connection. (db_client:dbqueryconnection/1)
  -(TupleTerm) is a predicate containing a tuple. (db_client:answertupleterm/1)

answertupleterm/1: REGTYPE

```
answertupleterm([]).
answertupleterm(tup(T)) :-
        tuple(T).
```

Usage: `answertupleterm(X)`

- *Description:* `X` is a predicate containing a tuple.

db_get_tables/2: PREDICATE

Usage: `db_get_tables(+DbConnection,-(Tables))`

- *Description:* `Tables` contains the tables available in `DbConnection`.
- *Call and exit should be compatible with:*
  `+DbConnection` a unique identifier of a database session connection. (db_client:dbconnection/1)
  `-(Tables)` is a list of `atms`. (basic_props:list/2)

db_table_types/3: PREDICATE

Usage: `db_table_types(+DbConnection,+Table,-(AttrTypes))`

- *Description:* `AttrTypes` are the attributes and types of `Table` in `DbConnection`.
- *Call and exit should be compatible with:*
  `+DbConnection` a unique identifier of a database session connection. (db_client:dbconnection/1)
  `+Table` is an atom. (basic_props:atm/1)
  `-(AttrTypes)` is a list. (basic_props:list/1)

db_logoff/1: PREDICATE

Usage: `db_logoff(+DbConnection)`

- *Description:* Logs off from the database identified by `DbConnection`. It fails and display an error message if the login is not succesfully completed.
- *Call and exit should be compatible with:*
  `+DbConnection` a unique identifier of a database session connection. (db_client:dbconnection/1)

odbc_disconnect/1: PREDICATE

Usage: `odbc_disconnect(+Stream)`

- *Description:* Closes the socket connection to `Stream`. It is simply a call to `close/1`.
- *Call and exit should be compatible with:*
  `+Stream` is an open stream. (streams_basic:stream/1)

match_string/3: PREDICATE

No further documentation available for this predicate.

107.3 Documentation on internals (db_client)

dbhandle/1: REGTYPE

```
dbhandle(H) :-
        string(H).
```

Usage: dbhandle(H)

- *Description:* H is the internal database identifier (handle) of a database session.

stmthandle/1: REGTYPE

```
stmthandle(H) :-
        string(H).
```

Usage: stmthandle(H)

- *Description:* H is the internal statement identifier (handle) of a query answer in a database session connection.

107.4 Known bugs and planned improvements (db_client)

- Need to fix the problem with parametric types being meta_predicates ... list(_,tuple/1).

108 Low-level Prolog to Java interface

Author(s): Jesús Correas.

This module defines the low level Prolog to Java interface. This interface allows a Prolog program to start a Java process, and create Java objects, invoke methods, set/get attributes (fields), and handle Java events.

Although the Java side interface is explained in Javadoc format, the general interface structure is detailed here.

108.1 Low-Level Prolog to Java Interface Structure

This low-level prolog to java interface is made up of two parts: a Prolog part and a Java part. The Prolog part receives requests from a Prolog program and sends them to the Java part through a socket. The Java part receives requests from the socket and performs the actions included in the requests.

If an event is thrown in the java side, an asynchronous message must be sent away to the prolog side, in order to launch a prolog goal to handle the event. This asynchronous communication is made by the means of a second socket. The nature of this communication needs the use of threads both in java and prolog: one thread to deal with the 'sequential program flow,' and other thread (may be several) to do the job of event handling.

In the java side the threads are automatically created by the context of the objects we use: adding an event listener implies automatically that there will be a thread ready to launch it when the event raises. The prolog side is different: there must be a thread in the low-level interface that listens to the asynchronous socket to launch the goals requested.

108.1.1 Prolog side

The prolog side receives the actions to do in the java side from the user program, and sends them to the java side through the socket connection. When the action is done in the java side, the result is returned to the user program, or the action fails if any problem in the java side is found.

In order to send and receive prolog terms and java object references using a socket, this layer must transform this elements in a serialized representation, just like the java serialization package does. This transformation is done in our implementation using the `fast_read/1` and `fast_write/1` predicates included in Ciao, so any prolog element can be translated to and from a list of bytes.

Prolog data representation of java elements is very simple in this low-level interface. Java primitive types such as integers and characters are translated into Prolog terms, and even some Java objects are translated that way (e. g. Java strings). Java objects are represented in Prolog as compound terms with a reference to identify the corresponding Java object. Data conversion is made automatically when the interface is used, so the Prolog user programs do not have to deal with the complexity of this tasks.

108.1.2 Java side

The java side of this layer is more complex than the prolog side. The tasks this part have to deal to are the following:

- Wait for requests from the prolog side.
- Translate the prolog terms received in a 'serialized' form in a more useful java representation.
- Interpret the requests received from the prolog side.

- Handle the set of objects created by or derived from the requests received from de prolog side.
- Handle the events raised in the java side, and launch the listeners added in the prolog side.
- Handle the exceptions raised in the java side, and send to the prolog side.

In the implementation of the java side, two items must be carefully designed: the handling of java objects, and the representation of prolog data structures. The last item is specially important because all the interactions between prolog and java are made using prolog structures, an easy way to standardize the different data management of both languages. Even the requests themselves are encapsulated using prolog structures. The overload of this encapsulation is not significant in terms of socket traffic, due to the optimal implementation of the prolog serialized term.

The java side must handle the objects created from the prolog side dinamically, and these objects must be accessed as fast as possible from the set of objects. The java API provides a powerful implementation of Hash tables that achieves all the requirements of our implementation.

On the other hand, the java representation of prolog terms is made using the inheritance of java classes. In the java side exists a representation of a generic prolog term, implemented as an abstract class in java. Variables, atoms, compound terms, lists, and numeric terms are classes in the java side which inherit from the term class. Java objects can be seen also under the prolog representation as compound terms, where the single argument corresponds to the Hash key of the actual java object in the Hash table referred to before. This behaviour makes the handling of mixed java and prolog elements easy. Prolog goals are represented in the java side as objects which contain a prolog compound term with the term representing the goal. This case will be seen more in depth next, when the java to prolog is explained.

108.2 Java event handling from Prolog

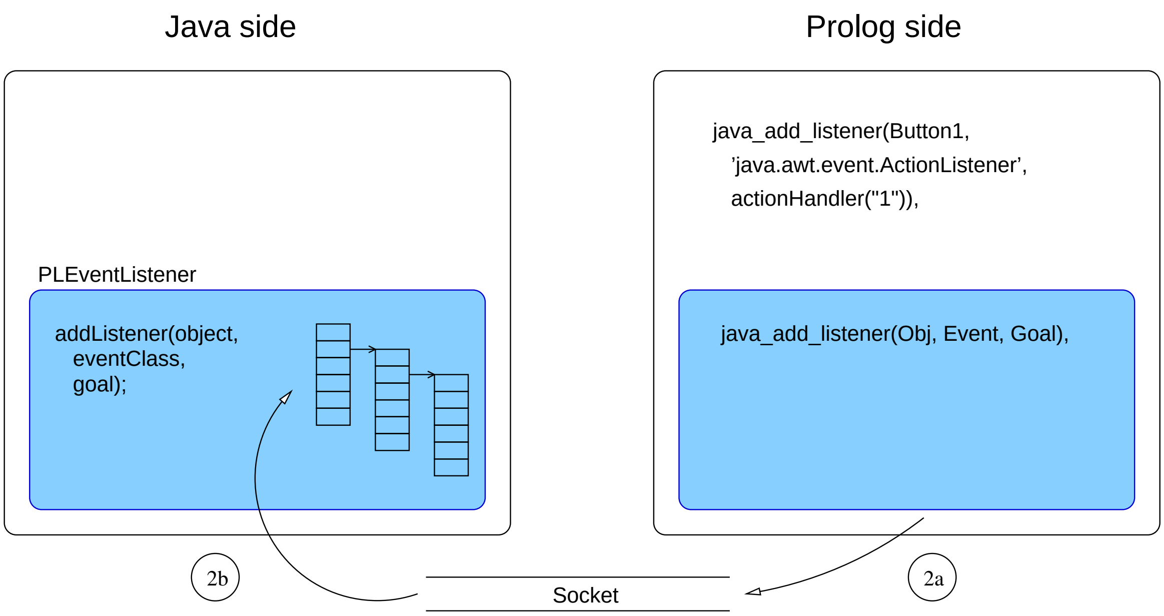
Java event handling is based on a delegation model since version 1.1.x. This approach to event handling is very powerful and elegant, but a user program cannot handle all the events that can raise on a given object: for each kind of event, a listener must be implemented and added specifically. However, the Java 2 API includes a special listener (`AWTEventListener`) that can manage the internal java event queue.

The prolog to java interface has been designed to emulate the java event handler, and is also based on event objects and listeners. The low level prolog to java interface implements its own event manager, to handle those events that have prolog listeners associated to the object that raises the event. From the prolog side can be added listeners to objects for specific events. The java side includes a list of goals to launch from the object and event type.

Due to the events nature, the event handler must work in a separate thread to manage the events asynchronously. The java side has its own mechanisms to work this way. The prolog side must be implemented specially for event handling using threads. The communication between java and prolog is also asynchronous, and a additional socket stream is used to avoid interferences with the main socket stream. The event stream will work in this implementation only in one way: from java to prolog. If an event handler needs to send back requests to java, it will use the main socket stream, just like the requests sent directly from a prolog program.

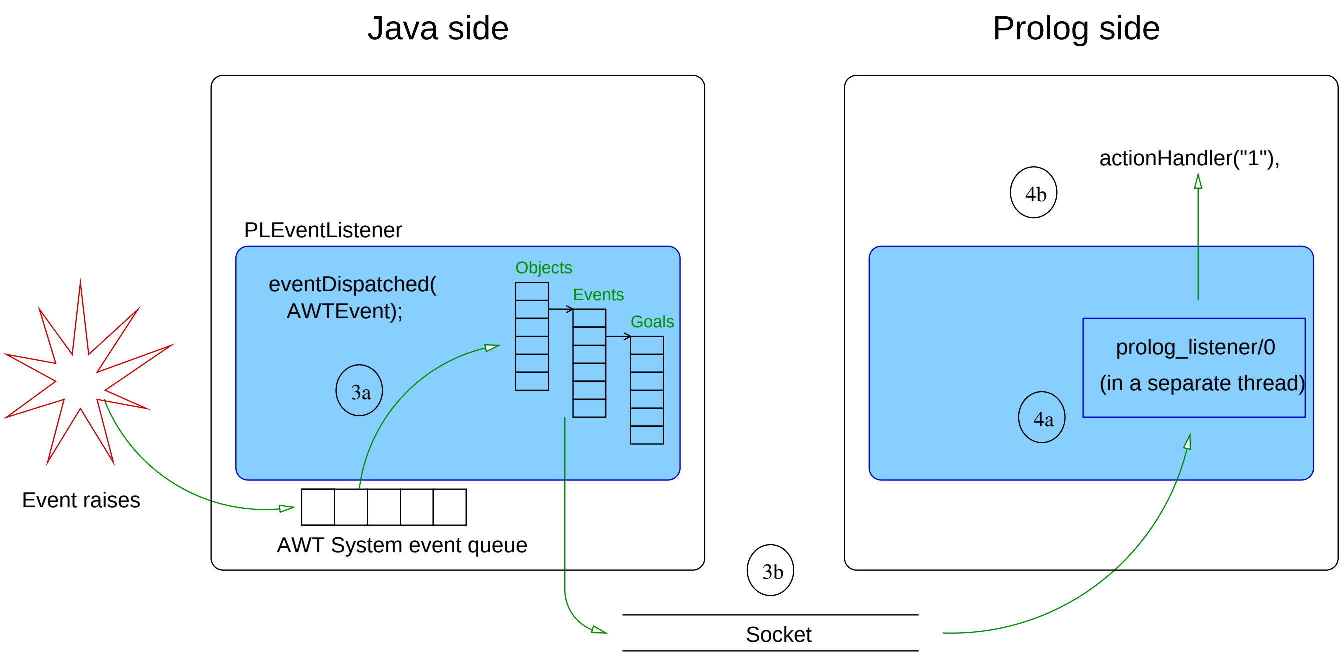
The internal process of register a Prolog event handler to a Java event is shown in the next figure:

Prolog registering of Java events



When an event raises, the low-level Prolog to Java interface have to send to the Prolog user program the goal to evaluate. Graphically, the complete process takes the tasks involved in the following figure:

Prolog handling of Java events



108.3 Java exception handling from Prolog

Java exception handling is very similar to the peer prolog handling: it includes some specific statements to trap exceptions from user code. In the java side, the exceptions can be originated from an incorrect request, or can be originated in the code called from the request. Both exception types will be sent to prolog using the main socket stream, leaving the prolog program manage the exception. However, the first kind of exceptions are prefixed, so the user program can distinguish from the second type of exceptions.

In order to handle exceptions properly using the prolog to java and java to prolog interfaces simultaneously, in both sides of the interface will be filtered those exceptions coming from their own side: this avoids an endless loop of exceptions bouncing from one side to another.

108.4 Usage and interface (`javart`)

- **Library usage:**

  `:- use_module(library(javart)).`
- **Exports:**
  - *Predicates:*

    `java_create_object/2`, `java_delete_object/1`, `java_invoke_method/2`, `java_get_value/2`, `java_set_value/2`, `java_connection/0`, `java_remote_connection/1`, `java_disconnection/0`, `java_add_listener/3`, `java_remove_listener/3`.
  - *Regular Types:*

    `machine_name/1`, `java_constructor/1`, `java_object/1`, `java_method/1`, `java_field/1`, `java_event/1`, `prolog_goal/1`.
- **Other modules used:**
  - *Application modules:*

    `javasock`.
  - *System library modules:*

    `format`, `lists`, `read`, `write`, `system`.

108.5 Documentation on exports (`javart`)

java_create_object/2: PREDICATE

Usage: `java_create_object(+java_constructor,-(java_object))`

- *Description:* New java object creation. The constructor must be a compound term as defined by its type, with the full class name as functor (e.g., 'java.lang.String'), and the parameters passed to the constructor as arguments of the structure.
- *Call and exit should be compatible with:*

  `+java_constructor` is a java constructor (structure with functor as constructor full name, and arguments as constructor arguments). (`javart:java_constructor/1`)

  `-(java_object)` is a java object (a structure with functor '$java_object', and argument an integer given by the java side). (`javart:java_object/1`)

machine_name/1: REGTYPE

Usage: `machine_name(X)`

- *Description:* `X` is the network name of a machine.

java_constructor/1: REGTYPE

Usage: `java_constructor(X)`

- *Description:* `X` is a java constructor (structure with functor as constructor full name, and arguments as constructor arguments).

java_object/1: REGTYPE

Usage: `java_object(X)`

- *Description:* `X` is a java object (a structure with functor '$java_object', and argument an integer given by the java side).

java_delete_object/1: PREDICATE

Usage: `java_delete_object(+java_object)`

- *Description:* Java object deletion. It removes the object given as argument from the Java object table.
- *Call and exit should be compatible with:*

  `+java_object` is a java object (a structure with functor '$java_object', and argument an integer given by the java side). (`javart:java_object/1`)

java_invoke_method/2: PREDICATE

Usage: `java_invoke_method(+java_object,+java_method)`

- *Description:* Invokes a java method on an object. Given a Java object reference, invokes the method represented with the second argument.
- *Call and exit should be compatible with:*

  `+java_object` is a java object (a structure with functor '$java_object', and argument an integer given by the java side). (`javart:java_object/1`)

  `+java_method` is a java method (structure with functor as method name, and arguments as method ones, plus a result argument. This result argument is unified with the atom 'Yes' if the java method returns void). (`javart:java_method/1`)

java_method/1: REGTYPE

Usage: `java_method(X)`

- *Description:* `X` is a java method (structure with functor as method name, and arguments as method ones, plus a result argument. This result argument is unified with the atom 'Yes' if the java method returns void).

java_get_value/2: PREDICATE

Usage: `java_get_value(+java_object,+java_field)`

- *Description:* Gets the value of a field. Given a Java object as first argument, it instantiates the variable given as second argument. This field must be uninstantiated in the java_field functor, or this predicate will fail.
- *Call and exit should be compatible with:*

  `+java_object` is a java object (a structure with functor '$java_object', and argument an integer given by the java side). (`javart:java_object/1`)

  `+java_field` is a java field (structure on which the functor name is the field name, and the single argument is the field value). (`javart:java_field/1`)

java_set_value/2: PREDICATE

Usage: `java_set_value(+java_object,+java_field)`

- *Description:* Sets the value of a Java object field. Given a Java object reference, it assigns the value included in the java_field compound term. The field value in the java_field structure must be instantiated.
- *Call and exit should be compatible with:*

  `+java_object` is a java object (a structure with functor '$java_object', and argument an integer given by the java side). (`javart:java_object/1`)

  `+java_field` is a java field (structure on which the functor name is the field name, and the single argument is the field value). (`javart:java_field/1`)

java_field/1: REGTYPE

Usage: `java_field(X)`

- *Description:* `X` is a java field (structure on which the functor name is the field name, and the single argument is the field value).

java_connection/0: PREDICATE

Usage:

- *Description:* Starts the Java server on the local machine, connects it through the data and event sockets, and starts the event handling thread.

java_remote_connection/1: PREDICATE

Usage: `java_remote_connection(+machine_name)`

- *Description:* Starts the Java server in machine_name, connects it through the data and event sockets, and starts the event handling thread.

java_disconnection/0: PREDICATE

Usage:

- *Description:* Closes the connection with the java process.

java_add_listener/3: PREDICATE

Meta-predicate with arguments: `java_add_listener(?,?,goal)`.

Usage: `java_add_listener(+java_object,+java_event,+prolog_goal)`

- *Description:* Adds a listener to an event on an object. Given a Java object reference, it registers the goal received as third argument to be launched when the Java event raises.
- *Call and exit should be compatible with:*

  `+java_object` is a java object (a structure with functor '$java_object', and argument an integer given by the java side). (`javart:java_object/1`)

  `+java_event` is a java event represented as an atom with the full event constructor name (e.g., 'java.awt.event.ActionListener'). (`javart:java_event/1`)

  `+prolog_goal` is a prolog predicate. Prolog term that represents the goal that must be invoked when the event raises on the object. The predicate arguments can be java objects, or even the result of java methods. These java objects will be evaluated when the event raises (instead of when the listener is added). The arguments that represent java objects must be instantiated to already created objects. The variables will be kept uninstantiated when the event raises and the predicate is called. (`javart:prolog_goal/1`)

java_event/1: REGTYPE

Usage: `java_event(X)`

- *Description:* `X` is a java event represented as an atom with the full event constructor name (e.g., 'java.awt.event.ActionListener').

prolog_goal/1: REGTYPE

Usage: `prolog_goal(X)`

- *Description:* `X` is a prolog predicate. Prolog term that represents the goal that must be invoked when the event raises on the object. The predicate arguments can be java objects, or even the result of java methods. These java objects will be evaluated when the event raises (instead of when the listener is added). The arguments that represent java objects must be instantiated to already created objects. The variables will be kept uninstantiated when the event raises and the predicate is called.

java_remove_listener/3: PREDICATE

Usage: `java_remove_listener(+java_object,+java_event,+prolog_goal)`

- *Description:* It removes a listener from an object event queue. Given a Java object reference, goal registered for the given event is removed.
- *Call and exit should be compatible with:*

  `+java_object` is a java object (a structure with functor '$java_object', and argument an integer given by the java side). (`javart:java_object/1`)

  `+java_event` is a java event represented as an atom with the full event constructor name (e.g., 'java.awt.event.ActionListener'). (`javart:java_event/1`)

  `+prolog_goal` is a prolog predicate. Prolog term that represents the goal that must be invoked when the event raises on the object. The predicate arguments can be java objects, or even the result of java methods. These java objects will be evaluated when the event raises (instead of when the listener is added). The arguments that represent java objects must be instantiated to already created objects. The variables will be kept uninstantiated when the event raises and the predicate is called. (`javart:prolog_goal/1`)

109 Low-level Java to Prolog interface

Author(s): Jesús Correas.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.5#40 (2000/2/8, 16:32:42 CET)

This module defines a low level Java to Prolog interface. This Prolog side of the Java to Prolog interface only has one public predicate: a server that listens at the socket connection with Java, and executes the commands received from the Java side.

In order to evaluate the goals received from the Java side, this module can work in two ways: executing them in the same engine, or starting a thread for each goal. The easiest way is to launch them in the same engine, but the goals must be evaluated sequentially: once a goal provides the first solution, all the subsequent goals must be finished before this goal can backtrack to provide another solution. The Prolog side of this interface works as a top-level, and the goals partially evaluated are not independent.

The solution of this goal dependence is to evaluate the goals in a different prolog engine. Although Ciao includes a mechanism to evaluate goals in different engines, the approach used in this interface is to launch each goal in a different thread.

The decision of what kind of goal evaluation is selected is done by the Java side. Each evaluation type has its own command terms, so the Java side can choose the type it needs.

A Prolog server starts by calling the `prolog_server/0` predicate. The user predicates and libraries to be called from Java must be included in the executable file, or be accesible using the built-in predicates dealing with code loading.

109.1 Usage and interface (`jtopl`)

- **Library usage:**
  `:- use_module(library(jtopl)).`
- **Exports:**
  - *Predicates:*
    `prolog_server/0`.
- **Other modules used:**
  - *Application modules:*
    `javasock`.
  - *System library modules:*
    `system`, `read`, `write`, `dynamic`, `lists`, `format`, `dynmods`, `atom2term`.

109.2 Documentation on exports (`jtopl`)

prolog_server/0: PREDICATE

Usage:

- *Description:* Prolog server entry point. Reads from the standard input the node name and port number where the java client resides, and starts the prolog server listening at the data socket. This predicate acts as a server: it includes an endless read-process loop until the `prolog_exit` command is received.

109.3 Documentation on internals (`jtopl`)

command/1: REGTYPE

Usage: `command(X)`

– *Description:* `X` is a command received from the java client, to be executed by the Prolog process. The command is represented as an atom or a functor with arity 1. The command to be executed must be one of the following types:

  - `prolog_launch_query(Q)` Compound term to create a new query, received as single argument of this structure. A reference to the new query is returned to Java.
  - `prolog_launch_query_on_thread(Q)` Compound term to evaluate a new query using a separate thread. A reference to the new query is returned to Java.
  - `prolog_next_solution(ID)` Compound term to get the next solution of a goal identified by the single argument of the structure. A term representing the goal instantiated with the next solution is returned to Java.
  - `prolog_terminate_query(ID)` Compound term to terminate the goal identified by the argument. If the thread option is disabled, a cut is made in the goal search tree, and the goal is removed from the goal table; if the thread option is enabled, the thread evaluating the goal is terminated.
  - `prolog_exit` Atom to terminate the current Prolog process.

answer/1: REGTYPE

Usage: `answer(X)`

– *Description:* `X` is a response sent from the prolog server. Is represented as an atom or a functor with arity 1 or 2, depending on the functor name.

prolog_query/1: REGTYPE

Usage: `prolog_query(X)`

– *Description:* `X` is a query to be launched from the prolog server.

shell_s/0: PREDICATE

Usage:

– *Description:* Command execution loop. This predicate is called when the connection to Java is established, and performs an endless loop dealing with the following tasks:

  - reads a command from the socket. All the commands are explained in the `command` type description,
  - processes the command using the `process_first_command/1` and `process_next_command/1` predicates, and
  - returns the results of the command execution.

process_first_command/1: PREDICATE

Usage: `process_first_command(+Command)`

- *Description:* Processes the first command of a query. Using the threads option, it processes all the commands received from the prolog server.
- *Call and exit should be compatible with:*

  `+Command` is a command received from the java client, to be executed by the Prolog process. The command is represented as an atom or a functor with arity 1. The command to be executed must be one of the following types:

  - `prolog_launch_query(Q)` Compound term to create a new query, received as single argument of this structure. A reference to the new query is returned to Java.
  - `prolog_launch_query_on_thread(Q)` Compound term to evaluate a new query using a separate thread. A reference to the new query is returned to Java.
  - `prolog_next_solution(ID)` Compound term to get the next solution of a goal identified by the single argument of the structure. A term representing the goal instantiated with the next solution is returned to Java.
  - `prolog_terminate_query(ID)` Compound term to terminate the goal identified by the argument. If the thread option is disabled, a cut is made in the goal search tree, and the goal is removed from the goal table; if the thread option is enabled, the thread evaluating the goal is terminated.
  - `prolog_exit` Atom to terminate the current Prolog process.

  (`jtopl:command/1`)

process_next_command/2: PREDICATE

Usage: `process_next_command(+Id,+Command)`

- *Description:* Process the commands received from the prolog server when a query is being handled.
- *Call and exit should be compatible with:*

  `undefined:prolog_query_id(+Id)` (undefined property)

  `+Command` is a command received from the java client, to be executed by the Prolog process. The command is represented as an atom or a functor with arity 1. The command to be executed must be one of the following types:

  - `prolog_launch_query(Q)` Compound term to create a new query, received as single argument of this structure. A reference to the new query is returned to Java.
  - `prolog_launch_query_on_thread(Q)` Compound term to evaluate a new query using a separate thread. A reference to the new query is returned to Java.
  - `prolog_next_solution(ID)` Compound term to get the next solution of a goal identified by the single argument of the structure. A term representing the goal instantiated with the next solution is returned to Java.
  - `prolog_terminate_query(ID)` Compound term to terminate the goal identified by the argument. If the thread option is disabled, a cut is made in the goal search tree, and the goal is removed from the goal table; if the thread option is enabled, the thread evaluating the goal is terminated.
  - `prolog_exit` Atom to terminate the current Prolog process.

  (`jtopl:command/1`)

solve/1: PREDICATE

Usage: `solve(+Query)`

- *Description:* Runs the query processing the commands received from the java side, and handles query nesting. It is used only with the thread option disabled.
- *Call and exit should be compatible with:*

  `+Query` is a query to be launched from the prolog server. (`jtopl:prolog_query/1`)

solve_on_thread/1: PREDICATE

Usage: `solve_on_thread(+Query)`

- *Description:* Runs the query on a separate thread and stores the solutions on the `query_solutions/2` data predicate.
- *Call and exit should be compatible with:*

  `+Query` is a query to be launched from the prolog server. (`jtopl:prolog_query/1`)

get_query_id/1: PREDICATE

Usage: `get_query_id(-(Id))`

- *Description:* Produces the query id for the next query, when is invoked the threads option disabled.
- *Call and exit should be compatible with:*

  `undefined:prolog_query_id(-(Id))` (undefined property)

prolog_parse/2: PREDICATE

Usage: `prolog_parse(+String,-(Term))`

- *Description:* Parses the string received as first argument and returns the prolog term as second argument. **Important:** This is a private predicate but could be called from java side, to parse strings to Prolog terms.
- *Call and exit should be compatible with:*

  `+String` is a string (a list of character codes). (`basic_props:string/1`)

  `-(Term)` is any term. (`basic_props:term/1`)

read_command/1: PREDICATE

Usage: `read_command(-(Command))`

- *Description:* Reads from the input stream a new prolog server command.
- *Call and exit should be compatible with:*

  `-(Command)` is a command received from the java client, to be executed by the Prolog process. The command is represented as an atom or a functor with arity 1. The command to be executed must be one of the following types:

  - `prolog_launch_query(Q)` Compound term to create a new query, received as single argument of this structure. A reference to the new query is returned to Java.
  - `prolog_launch_query_on_thread(Q)` Compound term to evaluate a new query using a separate thread. A reference to the new query is returned to Java.

- `prolog_next_solution(ID)` Compound term to get the next solution of a goal identified by the single argument of the structure. A term representing the goal instantiated with the next solution is returned to Java.
- `prolog_terminate_query(ID)` Compound term to terminate the goal identified by the argument. If the thread option is disabled, a cut is made in the goal search tree, and the goal is removed from the goal table; if the thread option is enabled, the thread evaluating the goal is terminated.
- `prolog_exit` Atom to terminate the current Prolog process.

(`jtopl:command/1`)

write_answer/1: PREDICATE

Usage: `write_answer(+Answer)`

- *Description:* writes to the output socket stream the given answer.
- *Call and exit should be compatible with:*

  `+Answer` is a response sent from the prolog server. Is represented as an atom or a functor with arity 1 or 2, depending on the functor name.

  (`jtopl:answer/1`)

110 Low-level Prolog to Java socket connection

Author(s): Jesús Correas.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.5#49 (2000/2/8, 16:35:38 CET)

This module defines a low level socket interface, to be used by javart and jtopl. Includes all the code related directly to the handling of sockets. This library should not be used by any user program, because is a very low-level connection to Java. Use `javart` (Prolog to Java low-level interface) or `jtopl` (Java to Prolog interface) libraries instead.

110.1 Usage and interface (`javasock`)

- **Library usage:**
  `:- use_module(library(javasock)).`
- **Exports:**
  - *Predicates:*
    `socket_connection/2`, `socket_disconnection/0`, `java_fast_read/2`, `java_fast_write/2`, `java_stream/3`.
- **Other modules used:**
  - *System library modules:*
    `fastrw`, `read`, `sockets/sockets`, `dynamic`, `format`.

110.2 Documentation on exports (`javasock`)

socket_connection/2: PREDICATE

Usage: `socket_connection(+node,+stream)`

- *Description:* Given a stream connected to a node, it gets the socket port from the given stream and creates the sockets to connect to the java process.
- *Call and exit should be compatible with:*
  `+node` is currently instantiated to an atom. (`term_typing:atom/1`)
  `+stream` is an open stream. (`streams_basic:stream/1`)

socket_disconnection/0: PREDICATE

Usage:

- *Description:* It closes the sockets to disconnect from the java process.

java_fast_read/2: PREDICATE

Usage: `java_fast_read(+type,-(term))`

- *Description:* It reads from the given stream type one term and unifies it with the term received as second argument. This is the basic predicate used to receive data from the Java side. The first argument reflects the socket type: event or data. The second argument is unified with the data received from the socket.

- *Call and exit should be compatible with:*
  `+type` is currently instantiated to an atom. (`term_typing:atom/1`)
  `-(term)` is any term. (`basic_props:term/1`)

java_fast_write/2: PREDICATE

Usage: `java_fast_write(+type,+term)`

- *Description:* It writes on the given stream type the term received as second argument. This is the basic predicate used to send data to the Java side. The first argument reflects the socket type: event or data. The second argument is the term to be sent to the Java side.
- *Call and exit should be compatible with:*
  `+type` is currently instantiated to an atom. (`term_typing:atom/1`)
  `+term` is any term. (`basic_props:term/1`)

java_stream/3: PREDICATE

The predicate is of type *dynamic*.

Usage: `java_stream(DataStream,EventStream,Address)`

- *Description:* Stores the identifiers of the streams used. A fact is asserted when the connection to the Java process is established. It Contains the data and event streams, and the network address where the Java process is running.
- *Call and exit should be compatible with:*
  `DataStream` is an atom. (`basic_props:atm/1`)
  `EventStream` is an integer. (`basic_props:int/1`)
  `undefined:machine_name(Address)` (undefined property)

111 Calling emacs from Prolog

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.5#54 (2000/2/10, 21:32:23 CET)

This library provides a *prolog-emacs interface*. This interface is complementary to (and independent from) the emacs mode, which is used to develop programs from within the `emacs` editor/environment. Instead, this library allows calling `emacs` from a running Prolog program. This facilitates the use of `emacs` as a "user interface" for a Prolog program. Emacs can be made to:

- Visit a file, which can then be edited.
- Execute arbitrary *emacs lisp* code, sent from Prolog.

In order for this library to work correctly, the following is needed:

- You should be running the `emacs` editor on the same machine where the executable calling this library is executing.
- This `emacs` should be running the *emacs server*. This can be done by including the following line in your `.emacs` file:

  ```
  ;; Start a server that emacsclient can connect to.
  (server-start)
  ```

  Or typing `M-x server-start` within `emacs`.

This suffices for using `emacs` to edit files. For running arbitrary code the following also needs to be added to the `.emacs` file:

`(setq enable-local-eval t)`
  Allows executing lisp code without asking.

`(setq enable-local-eval nil)`
  Does not allow executing lisp code without asking.

`(setq enable-local-eval 'maybe)`
  Allows executing lisp code only if user agrees after asking (asks interactively for every invocation).

Examples:

Assuming that a `.pl` file loads this library, then:

`..., emacs_edit('foo'), ...`
  Opens file `foo` for editing in `emacs`.

`..., emacs_eval_nowait("(run-ciao-toplevel)"), ...`
  Starts execution of a Ciao top-level within `emacs`.

111.1 Usage and interface (emacs)

- **Library usage:**
  `:- use_module(library(emacs)).`
- **Exports:**
  - *Predicates:*
    `emacs_edit/1`, `emacs_edit_nowait/1`, `emacs_eval/1`, `emacs_eval_nowait/1`.
  - *Regular Types:*
    `elisp_string/1`.
- **Other modules used:**
  - *System library modules:*
    `metaterms`, `lists`, `terms`, `system`.

111.2 Documentation on exports (emacs)

emacs_edit/1: PREDICATE

Usage: `emacs_edit(+filename)`

- *Description:* Opens the given file for editing in `emacs`. Waits for editing to finish before continuing.

emacs_edit_nowait/1: PREDICATE

Usage: `emacs_edit_nowait(+filename)`

- *Description:* Opens the given file for editing in `emacs` and continues without waiting for editing to finish.

emacs_eval/1: PREDICATE

Usage: `emacs_eval(+elisp_string)`

- *Description:* Executes in emacs the lisp code given as argument. Waits for the command to finish before continuing.

emacs_eval_nowait/1: PREDICATE

Usage: `emacs_eval_nowait(+elisp_string)`

- *Description:* Executes in emacs the lisp code given as argument and continues without waiting for it to finish.

elisp_string/1: REGTYPE

Usage: `elisp_string(L)`

- *Description:* `L` is a string containing `emacs` lisp code.

112 linda (library)

Version: 0.9#66 (1999/4/29, 12:28:0 MEST)

This is a SICStus-like linda package. Note that this is essentially quite obsolete, and provided mostly in case it is needed for compatibility, since Ciao now supports all Linda functionality (and more) through the concurrent fact database.

112.1 Usage and interface (`linda`)

- **Library usage:**
  `:- use_module(library(linda)).`
- **Exports:**
  - *Predicates:*
    `linda_client/1`, `close_client/0`, `in/1`, `in/2`, `in_noblock/1`, `out/1`, `rd/1`, `rd/2`, `rd_noblock/1`, `rd_findall/3`, `linda_timeout/2`, `halt_server/0`, `open_client/2`, `in_stream/2`, `out_stream/2`.
- **Other modules used:**
  - *System library modules:*
    `read`, `fastrw`, `sockets/sockets`.

112.2 Documentation on exports (`linda`)

linda_client/1: PREDICATE

No further documentation available for this predicate.

close_client/0: PREDICATE

No further documentation available for this predicate.

in/1: PREDICATE

No further documentation available for this predicate.

in/2: PREDICATE

No further documentation available for this predicate.

in_noblock/1: PREDICATE

No further documentation available for this predicate.

out/1: PREDICATE

No further documentation available for this predicate.

rd/1: PREDICATE

No further documentation available for this predicate.

rd/2: PREDICATE

No further documentation available for this predicate.

rd_noblock/1: PREDICATE

No further documentation available for this predicate.

rd_findall/3: PREDICATE

No further documentation available for this predicate.

linda_timeout/2: PREDICATE

No further documentation available for this predicate.

halt_server/0: PREDICATE

No further documentation available for this predicate.

open_client/2: PREDICATE

No further documentation available for this predicate.

in_stream/2: PREDICATE

No further documentation available for this predicate.

out_stream/2: PREDICATE

No further documentation available for this predicate.

PART IX - Abstract data types

This part includes libraries which implement some generic data structures (abstract data types) that are used frequently in programs or in the Ciao system itself.

113 atom2term (library)

Author(s): Francisco Bueno, Daniel Cabeza.
Version: 1.5#118 (2000/4/19, 18:13:43 CEST)
Version of last change: 0.9#78 (1999/5/3, 19:19:27 MEST)

113.1 Usage and interface (`atom2term`)

- **Library usage:**
  `:- use_module(library(atom2term)).`
- **Exports:**
  - *Predicates:*
    `atom2term/2`, `string2term/2`, `parse_term/3`.

113.2 Documentation on exports (`atom2term`)

atom2term/2: PREDICATE

Usage: `atom2term(+Atom,-(Term))`

- *Description:* Convert an atom into a term. `Atom` is an atom, but must have term syntax. `Term` is a term resulting from parsing `Atom` char by char. Note: this is just a simple version written for parsing daVinci's messages. There should be a call to the standard reader to do this!

string2term/2: PREDICATE

Usage: `string2term(+String,-(Term))`

- *Description:* Same as `atom2term/2` but first argument is a string (containing a term).

parse_term/3: PREDICATE

No further documentation available for this predicate.

114 counters (library)

Version: 0.4#5 (1998/2/24)

114.1 Usage and interface (counters)

- **Library usage:**
  `:- use_module(library(counters)).`
- **Exports:**
  - *Predicates:*
    `setcounter/2`, `getcounter/2`, `inccounter/2`.

114.2 Documentation on exports (counters)

setcounter/2: PREDICATE
No further documentation available for this predicate.

getcounter/2: PREDICATE
No further documentation available for this predicate.

inccounter/2: PREDICATE
No further documentation available for this predicate.

115 Identity lists

Author(s): Francisco Bueno.

Version: 0.4#5 (1998/2/24)

The operations in this module handle lists by performing equality checks via identity instead of unification.

115.1 Usage and interface (`idlists`)

- **Library usage:**
  `:- use_module(library(idlists)).`
- **Exports:**
  - *Predicates:*
    `member_0/2`, `memberchk/2`, `list_insert/2`, `add_after/4`, `add_before/4`, `delete/3`, `subtract/3`, `union_idlists/3`.

115.2 Documentation on exports (`idlists`)

member_0/2: PREDICATE

No further documentation available for this predicate.

memberchk/2: PREDICATE

`memberchk(X,Xs)`

Checks that `X` is an element of (list) `Xs`.

list_insert/2: PREDICATE

Usage: `list_insert(-(List),+Term)`

- *Description:* Adds `Term` to the end of (tail-opened) `List` if there is not an element in `List` identical to `Term`.

add_after/4: PREDICATE

Usage: `add_after(+L0,+E0,+E,-(L))`

- *Description:* Adds element `E` after the first element identical to `E0` (or at end) of list `L0`, returning in `L` the new list.

add_before/4: PREDICATE

Usage: `add_before(+L0,+E0,+E,-(L))`

- *Description:* Adds element `E` before the first element identical to `E0` (or at start) of list `L0`, returning in `L` the new list.

delete/3: PREDICATE

Usage: `delete(+List,+Element,-(Rest))`

– *Description:* `Rest` has the same elements of `List` except for all the occurrences of elements identical to `Element`.

subtract/3: PREDICATE

Usage: `subtract(+Set,+Set0,-(Difference))`

– *Description:* `Difference` has the same elements of `Set` except those which have an identical occurrence in `Set0`.

union_idlists/3: PREDICATE

Usage: `union_idlists(+List1,+List2,-(List))`

– *Description:* `List` has the elements which are in `List1` but are not identical to an element in `List2` followed by the elements in `List2`.

116 Term manipulation

Version: 1.5#43 (2000/2/4, 19:53:51 CET)

116.1 Usage and interface (`metaterms`)

- **Library usage:**
  `:- use_module(library(metaterms)).`
- **Exports:**
  - *Predicates:*
    `varset/2`, `varsbag/3`, `ask/2`, `instance/2`, `variant/2`.
- **Other modules used:**
  - *System library modules:*
    `lists`, `idlists`, `sort`, `write`.

116.2 Documentation on exports (`metaterms`)

varset/2: PREDICATE

`varset(Term,Xs)`

`Xs` is the sorted list of all the variables in `Term`.

varsbag/3: PREDICATE

`varsbag(Term,Vs,Xs)`

`Vs` is the list of all the variables in `Term` ordered as they appear in `Term` left-to-right depth-first (including duplicates) plus `Xs`.

ask/2: PREDICATE

`ask(Term1,Term2)`

`Term1` and `Term2` unify without producing bindings for the variables of `Term1`. I.e., `instance(Term1,Term2)` holds.

instance/2: PREDICATE

`instance(Term1,Term2)`

`Term1` is an instance of `Term2`.

variant/2: PREDICATE

`variant(Term1,Term2)`

`Term1` and `Term2` are identical up to renaming.

117 Lists of numbers

Version: 0.9#99 (1999/5/26, 11:33:15 MEST)

117.1 Usage and interface (numlists)

- **Library usage:**
  `:- use_module(library(numlists)).`
- **Exports:**
  - *Predicates:*
    `get_primes/2`, `sum_list/2`, `sum_list/3`, `sum_list_of_lists/2`, `sum_list_of_lists/3`.
  - *Regular Types:*
    `intlist/1`, `numlist/1`.
- **Other modules used:**
  - *System library modules:*
    `lists`.

117.2 Documentation on exports (numlists)

get_primes/2: PREDICATE

Usage: `get_primes(N,Primes)`

- *Description:* Computes the `N`th first prime numbers in ascending order.
- *The following properties should hold at call time:*
  `N` is an integer. (`basic_props:int/1`)
- *The following properties should hold upon exit:*
  `Primes` is a list of integers. (`numlists:intlist/1`)

intlist/1: REGTYPE

Usage: `intlist(X)`

- *Description:* `X` is a list of integers.

numlist/1: REGTYPE

Usage: `numlist(X)`

- *Description:* `X` is a list of numbers.

sum_list/2: PREDICATE

Usage: `sum_list(List,N)`

- *Description:* `N` is the total sum of the elements of `List`.

- *The following properties should hold at call time:*
  `List` is a list of numbers. (`numlists:numlist/1`)
- *The following properties should hold upon exit:*
  `N` is a number. (`basic_props:num/1`)

sum_list/3: PREDICATE

Usage: `sum_list(List,N0,N)`

- *Description:* `N` is the total sum of the elements of `List` plus `N0`.
- *The following properties should hold at call time:*
  `List` is a list of numbers. (`numlists:numlist/1`)
  `N0` is a number. (`basic_props:num/1`)
- *The following properties should hold upon exit:*
  `N` is a number. (`basic_props:num/1`)

sum_list_of_lists/2: PREDICATE

Usage: `sum_list_of_lists(Lists,N)`

- *Description:* `N` is the total sum of the elements of the lists of `Lists`.
- *The following properties should hold at call time:*
  `List` is a list of `numlists`. (`basic_props:list/2`)
- *The following properties should hold upon exit:*
  `N` is a number. (`basic_props:num/1`)

sum_list_of_lists/3: PREDICATE

Usage: `sum_list_of_lists(Lists,N0,N)`

- *Description:* `N` is the total sum of the elements of the lists of `Lists` plus `N0`.
- *The following properties should hold at call time:*
  `List` is a list of `numlists`. (`basic_props:list/2`)
  `N0` is a number. (`basic_props:num/1`)
- *The following properties should hold upon exit:*
  `N` is a number. (`basic_props:num/1`)

118 Pattern (regular expression) matching

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.3#49 (1999/9/8, 21:8:9 MEST)

This library provides facilities for matching strings and terms against *patterns* (i.e., *regular expressions*).

118.1 Usage and interface (patterns)

- **Library usage:**
  `:- use_module(library(patterns)).`
- **Exports:**
  - *Predicates:*
    `match_pattern/2`, `match_pattern/3`, `case_insensitive_match/2`, `letter_match/2`, `match_pattern_pred/2`.
  - *Regular Types:*
    `pattern/1`.
- **Other modules used:**
  - *System library modules:*
    `lists`.

118.2 Documentation on exports (patterns)

match_pattern/2: PREDICATE

Usage: `match_pattern(Pattern,String)`

- *Description:* Matches `String` against `Pattern`. For example, `match_pattern("*.pl","foo.pl")` succeeds.
- *The following properties should hold at call time:*
  `Pattern` is a pattern to match against. (`patterns:pattern/1`)
  `String` is a string (a list of character codes). (`basic_props:string/1`)

match_pattern/3: PREDICATE

Usage: `match_pattern(Pattern,String,Tail)`

- *Description:* Matches `String` against `Pattern`. `Tail` is the remainder of the string after the match. For example, `match_pattern("??*","foo.pl",Tail)` succeeds, instantiating `Tail` to `"o.pl"`.
- *The following properties should hold at call time:*
  `Pattern` is a pattern to match against. (`patterns:pattern/1`)
  `String` is a string (a list of character codes). (`basic_props:string/1`)
  `Tail` is a string (a list of character codes). (`basic_props:string/1`)

case_insensitive_match/2: PREDICATE

Usage: `case_insensitive_match(Pred1,Pred2)`

- *Description:* Tests if two predicates `Pred1` and `Pred2` match in a case-insensitive way.

letter_match/2: PREDICATE

No further documentation available for this predicate.

pattern/1: REGTYPE

Special characters for `Pattern` are:

* Matches any string, including the null string.

? Matches any single character.

[...] Matches any one of the enclosed characters. A pair of characters separated by a minus sign denotes a range; any character lexically between those two characters, inclusive, is matched. If the first character following the [is a ^ then any character not enclosed is matched. No other character is special inside this construct. To include a] in a character set, you must make it the first character. To include a '-', you must use it in a context where it cannot possibly indicate a range: that is, as the first character, or immediately after a range.

`|` specifies an alternative. Two regular expressions A and B with `|` in between form an expression that matches anything that either A or B will match.

`{...}` groups alternatives inside larger patterns.

\ Quotes a special character (including itself).

Usage: `pattern(P)`

- *Description:* `P` is a pattern to match against.

match_pattern_pred/2: PREDICATE

Usage: `match_pattern_pred(Pred1,Pred2)`

- *Description:* Tests if two predicates `Pred1` and `Pred2` match using regular expressions.

119 Graphs

Author(s): F. Bueno.
Version: 1.5#118 (2000/4/19, 18:13:43 CEST)
Version of last change: 1.5#83 (2000/3/23, 19:23:58 CET)

119.1 Usage and interface (graphs)

- **Library usage:**
  `:- use_module(library(graphs)).`
- **Exports:**
  - *Predicates:*
    `dgraph_to_ugraph/2`, `dlgraph_to_lgraph/2`, `edges_to_ugraph/2`, `edges_to_lgraph/2`.
  - *Regular Types:*
    `dgraph/1`, `dlgraph/1`.
- **Other modules used:**
  - *System library modules:*
    `sort`, `graphs/ugraphs`, `graphs/lgraphs`.

119.2 Documentation on exports (graphs)

dgraph/1: REGTYPE

`dgraph(Graph)`

A directed graph is a term `graph(V,E)` where `V` is a list of vertices and `E` is a list of edges (none necessarily sorted). Edges are pairs of vertices which are directed, i.e., `(a,b)` represents `a->b`. Two vertices `a` and `b` are equal only if `a==b`.

Usage: `dgraph(Graph)`

- *Description:* `Graph` is a directed graph.

dlgraph/1: REGTYPE

`dlgraph(Graph)`

A labeled directed graph is a directed graph where edges are triples of the form `(a,l,b)` where `l` is the label of the edge `(a,b)`.

Usage: `dlgraph(Graph)`

- *Description:* `Graph` is a directed labeled graph.

dgraph_to_ugraph/2: PREDICATE

Usage: `dgraph_to_ugraph(+Graph,-(UGraph))`

- *Description:* Converts `Graph` to `UGraph`.

- *The following properties should hold at call time:*
  +Graph is a directed graph. (graphs:dgraph/1)
  -(UGraph) is a free variable. (term_typing:var/1)
- *The following properties should hold upon exit:*
  +Graph is a directed graph. (graphs:dgraph/1)
  -(UGraph) is an ugraph. (ugraphs:ugraph/1)

dlgraph_to_lgraph/2: PREDICATE

Usage: `dlgraph_to_lgraph(+Graph,-(LGraph))`

- *Description:* Converts `Edges` to `LGraph`.
- *The following properties should hold at call time:*
  +Graph is a directed labeled graph. (graphs:dlgraph/1)
  -(LGraph) is a free variable. (term_typing:var/1)
- *The following properties should hold upon exit:*
  +Graph is a directed labeled graph. (graphs:dlgraph/1)
  -(LGraph) is a labeled graph of `term` terms. (lgraphs:lgraph/2)

edges_to_ugraph/2: PREDICATE

Usage: `edges_to_ugraph(+Edges,-(UGraph))`

- *Description:* Converts `Graph` to `UGraph`.
- *The following properties should hold at call time:*
  +Edges is a list of `pairs`. (basic_props:list/2)
  -(UGraph) is a free variable. (term_typing:var/1)
- *The following properties should hold upon exit:*
  +Edges is a list of `pairs`. (basic_props:list/2)
  -(UGraph) is an ugraph. (ugraphs:ugraph/1)

edges_to_lgraph/2: PREDICATE

Usage: `edges_to_lgraph(+Edges,-(LGraph))`

- *Description:* Converts `Edges` to `LGraph`.
- *The following properties should hold at call time:*
  +Edges is a list of `triples`. (basic_props:list/2)
  -(LGraph) is a free variable. (term_typing:var/1)
- *The following properties should hold upon exit:*
  +Edges is a list of `triples`. (basic_props:list/2)
  -(LGraph) is a labeled graph of `term` terms. (lgraphs:lgraph/2)

119.3 Documentation on internals (graphs)

pair/1: REGTYPE

Usage: `pair(P)`

– *Description:* `P` is a pair `(_,_)`.

triple/1: REGTYPE

Usage: `triple(P)`

– *Description:* `P` is a triple `(_,_,_)`.

120 Unweighted graph-processing utilities

Author(s): M. Carlsson, adapted from shared code written by Richard A O'Keefe. Mods by F.Bueno and M.Carro..

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 0.9#105 (1999/6/4, 12:24:49 MEST)

An unweighted directed graph (ugraph) is represented as a list of (vertex-neighbors) pairs, where the pairs are in standard order (as produced by keysort with unique keys) and the neighbors of each vertex are also in standard order (as produced by sort), and every neighbor appears as a vertex even if it has no neighbors itself.

An undirected graph is represented as a directed graph where for each edge `(U,V)` there is a symmetric edge `(V,U)`.

An edge `(U,V)` is represented as the term `U-V`.

A vertex can be any term. Two vertices are distinct iff they are not identical (`==/2`).

A path is represented as a list of vertices. No vertex can appear twice in a path.

120.1 Usage and interface (ugraphs)

- **Library usage:**
  `:- use_module(library(ugraphs)).`
- **Exports:**
  - *Predicates:*
    `vertices_edges_to_ugraph/3`, `neighbors/3`, `edges/2`, `del_vertices/3`, `vertices/2`, `add_vertices/3`, `add_edges/3`, `transpose/2`, `point_to/3`.
  - *Regular Types:*
    `ugraph/1`.
- **Other modules used:**
  - *System library modules:*
    `sets`, `sort`.

120.2 Documentation on exports (ugraphs)

vertices_edges_to_ugraph/3: PREDICATE

No further documentation available for this predicate.

neighbors/3: PREDICATE

Usage: `neighbors(+Vertex,+Graph,-(Neighbors))`

- *Description:* Is true if `Vertex` is a vertex in `Graph` and `Neighbors` are its neighbors.

edges/2: PREDICATE

Usage: `edges(+Graph,-(Edges))`

– *Description:* Unifies `Edges` with the edges in `Graph`.

del_vertices/3: PREDICATE

Usage: `del_vertices(+Graph1,+Vertices,-(Graph2))`

– *Description:* Is true if `Graph2` is `Graph1` with `Vertices` and all edges to and from `Vertices` removed from it.

vertices/2: PREDICATE

Usage: `vertices(+Graph,-(Vertices))`

– *Description:* Unifies `Vertices` with the vertices in `Graph`.

add_vertices/3: PREDICATE

Usage: `add_vertices(+Graph1,+Vertices,-(Graph2))`

– *Description:* Is true if `Graph2` is `Graph1` with `Vertices` added to it.

add_edges/3: PREDICATE

Usage: `add_edges(+Graph1,+Edges,-(Graph2))`

– *Description:* Is true if `Graph2` is `Graph1` with `Edges` and their 'to' and 'from' vertices added to it.

transpose/2: PREDICATE

Usage: `transpose(+Graph,-(Transpose))`

– *Description:* Is true if `Transpose` is the graph computed by replacing each edge `(u,v)` in `Graph` by its symmetric edge `(v,u)`. It can only be used one way around. The cost is O(N^2).

point_to/3: PREDICATE

Usage: `point_to(+Vertex,+Graph,-(Point_to))`

– *Description:* Is true if `Point_to` is the list of nodes which go directly to `Vertex` in `Graph`.

ugraph/1: REGTYPE

Usage: `ugraph(Graph)`

– *Description:* `Graph` is an ugraph.

121 wgraphs (library)

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)
Version of last change: 0.4#5 (1998/2/24)

121.1 Usage and interface (`wgraphs`)

- **Library usage:**
  `:- use_module(library(wgraphs)).`
- **Exports:**
  - *Predicates:*
    `vertices_edges_to_wgraph/3`.
- **Other modules used:**
  - *System library modules:*
    `sets`, `sort`.

121.2 Documentation on exports (`wgraphs`)

vertices_edges_to_wgraph/3: PREDICATE

No further documentation available for this predicate.

122 Labeled graph-processing utilities

Author(s): F. Bueno.
Version: 1.5#118 (2000/4/19, 18:13:43 CEST)
See the comments for the `ugraphs` library.

122.1 Usage and interface (`lgraphs`)

- **Library usage:**
  `:- use_module(library(lgraphs)).`
- **Exports:**
  - *Predicates:*
    `vertices_edges_to_lgraph/3`.
  - *Regular Types:*
    `lgraph/2`.
- **Other modules used:**
  - *System library modules:*
    `sort`, `sets`.

122.2 Documentation on exports (`lgraphs`)

lgraph/2: REGTYPE
Usage: `lgraph(Graph,Type)`
- *Description:* `Graph` is a labeled graph of `Type` terms.

vertices_edges_to_lgraph/3: PREDICATE
No further documentation available for this predicate.

123 queues (library)

Version: 0.4#5 (1998/2/24)

123.1 Usage and interface (queues)

- **Library usage:**
  `:- use_module(library(queues)).`
- **Exports:**
  - *Predicates:*
    `q_empty/1`, `q_insert/3`, `q_member/2`, `q_delete/3`.

123.2 Documentation on exports (queues)

q_empty/1: PREDICATE
No further documentation available for this predicate.

q_insert/3: PREDICATE
No further documentation available for this predicate.

q_member/2: PREDICATE
No further documentation available for this predicate.

q_delete/3: PREDICATE
No further documentation available for this predicate.

124 Random numbers

Author(s): Daniel Cabeza.
Version: 1.5#118 (2000/4/19, 18:13:43 CEST)
Version of last change: 1.5#26 (1999/12/29, 15:7:59 CET)
This module provides predicates for generating pseudo-random numbers

124.1 Usage and interface (`random`)

- **Library usage:**
  `:- use_module(library(random)).`
- **Exports:**
  - *Predicates:*
    `random/1`, `random/3`, `srandom/1`.

124.2 Documentation on exports (`random`)

random/1: PREDICATE

`random(Number)`

`Number` is a (pseudo-) random number in the range [0.0,1.0]

random/3: PREDICATE

`random(Low,Up,Number)`

`Number` is a (pseudo-) random number in the range [`Low`, `Up`]

Usage 1: `random(+int,+int,-(int))`

- *Description:* If `Low` and `Up` are integers, `Number` is an integer.

srandom/1: PREDICATE

`srandom(Seed)`

Changes the sequence of pseudo-random numbers according to `Seed`. The stating sequence of numbers generated can be duplicated by calling the predicate with `Seed` unbound (the sequence depends on the OS).

125 sets (library)

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 0.9#17 (1999/3/23, 20:57:20 MET)

This module implements set operations. Sets are just ordered lists.

125.1 Usage and interface (sets)

- **Library usage:**
  `:- use_module(library(sets)).`
- **Exports:**
  - *Predicates:*
    `insert/3`, `ord_delete/3`, `ord_member/2`, `ord_test_member/3`, `ord_subtract/3`, `ord_intersection/3`, `ord_intersection_diff/4`, `ord_intersect/2`, `ord_subset/2`, `ord_subset_diff/3`, `ord_union/3`, `ord_union_diff/4`, `ord_union_symdiff/4`, `ord_union_change/3`, `merge/3`, `ord_disjoint/2`, `setproduct/3`.
- **Other modules used:**
  - *System library modules:*
    `sort`.

125.2 Documentation on exports (sets)

insert/3: PREDICATE

Usage: `insert(+Set1,+Element,-(Set2))`

- *Description:* It is true when `Set2` is `Set1` with `Element` inserted in it, preserving the order.

ord_delete/3: PREDICATE

Usage: `ord_delete(+Set0,+X,-(Set))`

- *Description:* It succeeds if `Set` is `Set0` without element `X`.

ord_member/2: PREDICATE

Usage: `ord_member(+X,+Set)`

- *Description:* It succeeds if `X` is member of `Set`.

ord_test_member/3: PREDICATE

Usage: `ord_test_member(+Set,+X,-(Result))`

- *Description:* If `X` is member of `Set` then `Result=yes`. Otherwise `Result=no`.

ord_subtract/3: PREDICATE

Usage: `ord_subtract(+Set1,+Set2,?(Difference))`

- *Description:* It is true when `Difference` contains all and only the elements of `Set1` which are not also in `Set2`.

ord_intersection/3: PREDICATE

Usage: `ord_intersection(+Set1,+Set2,?(Intersection))`

- *Description:* It is true when `Intersection` is the ordered representation of `Set1` and `Set2`, provided that `Set1` and `Set2` are ordered lists.

ord_intersection_diff/4: PREDICATE

Usage: `ord_intersection_diff(+Set1,+Set2,-(Intersect),-(NotIntersect))`

- *Description:* `Intersect` contains those elements which are both in `Set1` and `Set2`, and `NotIntersect` those which are in `Set1` but not in `Set2`.

ord_intersect/2: PREDICATE

Usage: `ord_intersect(+Xs,+Ys)`

- *Description:* Succeeds when the two ordered lists have at least one element in common.

ord_subset/2: PREDICATE

Usage: `ord_subset(+Xs,+Ys)`

- *Description:* Succeeds when every element of `Xs` appears in `Ys`.

ord_subset_diff/3: PREDICATE

Usage: `ord_subset_diff(+Set1,+Set2,-(Difference))`

- *Description:* It succeeds when every element of `Set1` appears in `Set2` and `Difference` has the elements of `Set2` which are not in `Set1`.

ord_union/3: PREDICATE

Usage: `ord_union(+Set1,+Set2,?(Union))`

- *Description:* It is true when `Union` is the union of `Set1` and `Set2`. When some element occurs in both sets, `Union` retains only one copy.

ord_union_diff/4: PREDICATE

Usage: `ord_union_diff(+Set1,+Set2,-(Union),-(Difference))`

- *Description:* It succeeds when `Union` is the union of `Set1` and `Set2`, and `Difference` is `Set2` set-minus `Set1`.

ord_union_symdiff/4: PREDICATE

Usage: `ord_union_symdiff(+Set1,+Set2,-(Union),-(Diff))`

- *Description:* It is true when Diff is the symmetric difference of Set1 and Set2, and Union is the union of Set1 and Set2.

ord_union_change/3: PREDICATE

Usage: `ord_union_change(+Set1,+Set2,-(Union))`

- *Description:* `Union` is the union of `Set1` and `Set2` and `Union` is different from `Set2`.

merge/3: PREDICATE

Usage: `merge(+Set1,+Set2,?(Union))`

- *Description:* See `ord_union/3`.

ord_disjoint/2: PREDICATE

Usage: `ord_disjoint(+Set1,+Set2)`

- *Description:* `Set1` and `Set2` have no element in common.

setproduct/3: PREDICATE

Usage: `setproduct(+Set1,+Set2,-(Product))`

- *Description:* `Product` has all two element sets such that one element is in `Set1` and the other in `set2`, except that if the same element belongs to both, then the corresponding one element set is in `Product`.

126 Variable name dictionaries

Author(s): Francisco Bueno.
Version: 1.5#118 (2000/4/19, 18:13:43 CEST)
Version of last change: 0.8#19 (1998/12/3, 12:53:16 MET)

126.1 Usage and interface (vndict)

- **Library usage:**
  `:- use_module(library(vndict)).`
- **Exports:**
  - *Predicates:*
    `create_dict/2`, `complete_dict/3`, `complete_vars_dict/3`, `prune_dict/3`, `dict2varnamesl/2`, `varnamesl2dict/2`, `find_name/4`, `rename/2`, `vars_names_dict/3`.
  - *Regular Types:*
    `null_dict/1`, `varname/1`, `varnamesl/1`, `varnamedict/1`.
- **Other modules used:**
  - *System library modules:*
    `idlists`, `metaterms`, `sets`, `sort`.

126.2 Documentation on exports (vndict)

null_dict/1: REGTYPE
Usage: `null_dict(D)`
- *Description:* `D` is an empty dictionary.

create_dict/2: PREDICATE
Usage: `create_dict(Term,Dict)`
- *Description:* `Dict` has names for all variables in `Term`.
- *The following properties should hold at call time:*
  `Term` is any term. (`basic_props:term/1`)
- *The following properties should hold upon exit:*
  `Dict` is a dictionary of variable names. (`vndict:varnamedict/1`)

complete_dict/3: PREDICATE
Usage: `complete_dict(+Dict,+Term,-(NewDict))`
- *Description:* `NewDict` is `Dict` augmented with the variables of `Term` not yet in `Dict`.

complete_vars_dict/3: PREDICATE

Usage: `complete_vars_dict(+Dict,+Vars,-(NewDict))`

- *Description:* `NewDict` is `Dict` augmented with the variables of the list `Vars` not yet in `Dict`.

prune_dict/3: PREDICATE

Usage: `prune_dict(+Term,+Dict,-(NewDict))`

- *Description:* `NewDict` is `Dict` reduced to just the variables of `Term`.

dict2varnamesl/2: PREDICATE

Usage: `dict2varnamesl(Dict,VNs)`

- *Description:* Translates `Dict` to `VNs`.
- *The following properties should hold at call time:*
  `Dict` is a dictionary of variable names. (`vndict:varnamedict/1`)
- *The following properties should hold upon exit:*
  `VNs` is a list of `Name=Var`, for a variable `Var` and its name `Name`. (`vndict:varnamesl/1`)

varnamesl2dict/2: PREDICATE

Usage: `varnamesl2dict(VNs,Dict)`

- *Description:* Translates `VNs` to `Dict`.
- *The following properties should hold at call time:*
  `VNs` is a list of `Name=Var`, for a variable `Var` and its name `Name`. (`vndict:varnamesl/1`)
- *The following properties should hold upon exit:*
  `Dict` is a dictionary of variable names. (`vndict:varnamedict/1`)

find_name/4: PREDICATE

`find_name(Vars,Names,V,Name)`

Given that `vars_names_dict(Dict,Vars,Names)` holds, it acts as `rename(X,Dict)`, but the name of `X` is given as `Name` instead of unified with it.

rename/2: PREDICATE

Usage: `rename(Term,Dict)`

- *Description:* Unifies each variable in `Term` with its name in `Dict`. If no name is found, a new name is created.
- *The following properties should hold at call time:*
  `Dict` is a dictionary of variable names. (`vndict:varnamedict/1`)

varname/1: REGTYPE

Usage: `varname(N)`

– *Description:* `N` is term representing a variable name.

varnamesl/1: REGTYPE

Usage: `varnamesl(D)`

– *Description:* `D` is a list of `Name=Var`, for a variable `Var` and its name `Name`.

varnamedict/1: REGTYPE

Usage: `varnamedict(D)`

– *Description:* `D` is a dictionary of variable names.

vars_names_dict/3: PREDICATE

Usage: `vars_names_dict(Dict,Vars,Names)`

– *Description:* `Varss` is a sorted list of variables, and `Names` is a list of their names, which correspond in the same order.

– *Call and exit should be compatible with:*

`Dict` is a dictionary of variable names. (`vndict:varnamedict/1`)

`Vars` is a list. (`basic_props:list/1`)

`Names` is a list. (`basic_props:list/1`)

PART X - Miscellaneous standalone utilities

Author(s): clip@dia.fi.upm.es, http://www.clip.dia.fi.upm.es/, The CLIP Group, School of Computer Science, Technical University of Madrid.

This is the documentation for a set of miscellaneous standalone utilities contained in the `etc` directory of the Ciao distribution.

127 Printing the declarations and code in a file

Author(s): Manuel Hermenegildo.

Version: 0.5#6 (1999/4/15, 20:33:6 MEST)

A simple program for printing assertion information (predicate declarations, property declarations, type declarations, etc.) and printing code-related information (imports, exports, libraries used, etc.) on a file. The file should be a single CIAO or Prolog source file. It uses the CIAO compiler's pass one to do it. This program is specially useful for example to check what the compiler is actually seeing after syntactic expansions, as well as for checking what assertions the assertion normalizer is producing from the original assertions in the file.

127.1 Usage (fileinfo)

```
fileinfo -asr <filename.asr>
   : pretty prints the contents of <filename.asr>

fileinfo [-v] [-m] <-a|-c|-e> <filename> [libdir1] ... [libdirN]
-v : verbose output (e.g., lists all files read)
-m : restrict info to current module
-a : print assertions
-c : print code and interface (imports/exports, etc.)
-e : print only errors - useful to check syntax of assertions in file

fileinfo -h
   : print this information
```

127.2 More detailed explanation of options (fileinfo)

- If the `-a` option is selected, `fileinfo` prints the assertions (only code-oriented assertions – not comment-oriented assertions) in the file *after normalization*. If the `-c` option is selected `fileinfo` prints the file interface, the declarations contained in the file, and the actual code. If the `-e` option is selected `fileinfo` prints only any sintactic and import-export errors found in the file, including the assertions.
- `filename` must be the name of a Prolog or CIAO source file.
- This filename can be followed by other arguments which will be taken to be library directory paths in which to look for files used by the file being analyzed.
- If the `-m` option is selected, only the information related to the current module is printed.
- The `-v` option produces verbose output. This is very useful for debugging, since all the files accessed during assertion normalization are listed.
- In the `-asr` usage, `fileinfo` can be used to print the contents of a `.asr` file in human-readable form.

128 Printing the contents of a bytecode file

Author(s): Daniel Cabeza.

Version: 0.5#2 (1999/11/11, 19:20:50 MET)

This simple program takes as an argument a bytecode (.po) file and prints out in symbolic form the information contained in the file. It uses compiler and engine builtins to do so, so that it keeps track with changes in bytecode format.

128.1 Usage (viewpo)

```
viewpo <file1>.po
   : print .po contents in symbolic form

viewpo -h
   : print this information
```

129 Crossed-references of a program

Author(s): Francisco Bueno.

The `xrefs` crossed-references Ciao library includes several modules which allow displaying crossed-references of the code in a program. Crossed-references identify modules which import code from other modules, or files (be them modules or not) which use code in other files. Crossed-references can be obtained as a term representing a graph, displayed graphically (using daVinci, a graph displayer developed by U. of Bremen, Germany), or printed as a list.

The libraries involved are as follows:

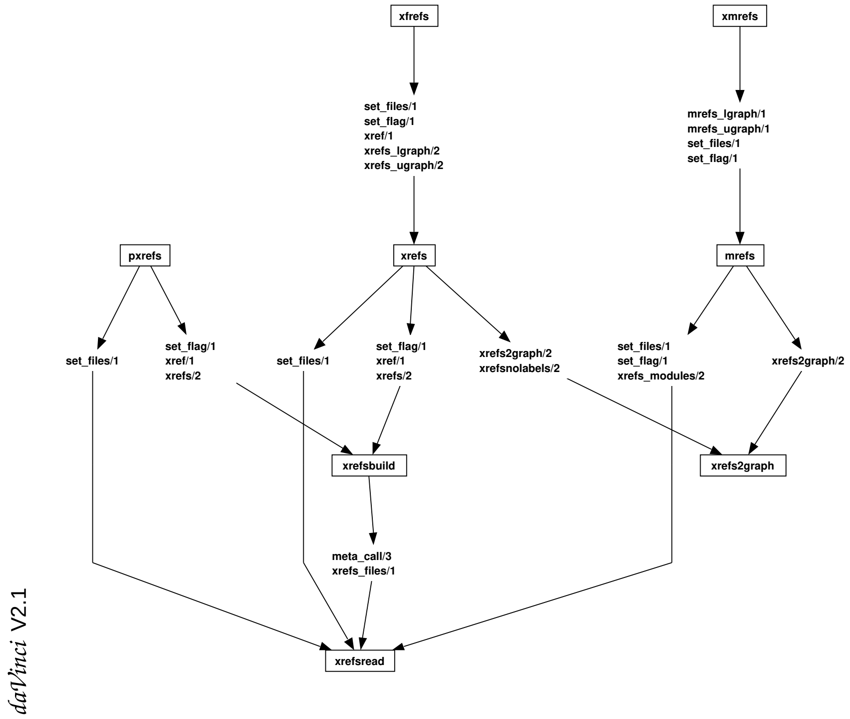
- `etc(xmrefs)` displays a graph of crossed-references between modules using daVinci,
- `etc(xfrefs)` displays a graph of crossed-references between files using daVinci,
- `library(xrefs)` obtains a graph of crossed-references between files,
- `library('xrefs/mrefs')` obtains a graph of crossed-references between modules,
- `library('xrefs/pxrefs')` prints a list of crossed-references between files.

The first two are intended to be used by loading in `ciaosh`. The other three are intended to be used as modules within an application.

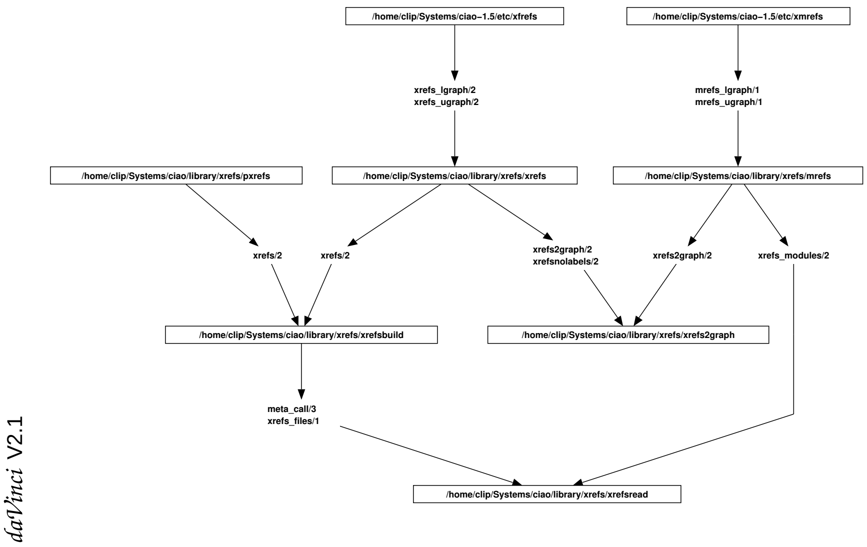
The following is an example graph of the library modules involved in the crossed-references application. It has been obtained with:

```
[ciao/etc]> ciaosh
Ciao-Prolog 1.5 #24: Tue Dec 28 14:12:11 CET 1999
?- use_module(xmrefs).

yes
?- set_flag(X).

X = 3 ?

yes
?- set_files([xfrefs, xmrefs,
              library(xrefs),
              library('xrefs/mrefs'),
              library('xrefs/pxrefs'),
              library('xrefs/xrefs2graph'),
              library('xrefs/xrefsbuild'),
              library('xrefs/xrefsread')
    ]).

yes
?- xmrefs.
```

so that it is displayed by daVinci as:



The following is an example graph of the same module files, where crossed-references have been obtained with `xfrefs:xfrefs(whodefs)` instead of `xmrefs:xmrefs`:



For more information refer to the xrefs documentation (`xrefs_doc.dvi`) in the source library of the Ciao distribution.

130 Gathering the dependent files for a file

Author(s): Daniel Cabeza, Manuel Hermenegildo.

Version: 1.0#6 (1998/11/5, 13:56:58 MET)

This simple program takes a single Ciao or Prolog source filename (which is typically the main file of an application). It prints out the list of all the dependent files, i.e., all files needed in order to build the application, including those which reside in libraries. This is particularly useful in Makefiles, for building standalone distributions (e.g., .tar files) automatically.

The filename should be followed by other arguments which will be taken to be library directory paths in which to look for files used by the file being analyzed.

130.1 Usage (get_deps)

```
get_deps [-u <filename>] <filename> [lib_dir1] ... [lib_dirN]
          : return dependent files for <filename>
            found in [lib_dir1] ... [lib_dirN]

get_deps -h
          : print this information
```

131 Finding differences between two Prolog files

Author(s): Francisco Bueno.

This simple program works like the good old diff but for files that contain Prolog code. It prints out the clauses that it finds are different in the files. Its use avoids textual differences such as different variable names and different formatting of the code in the files.

131.1 Usage (pldiff)

```
pldiff <file1> <file2>
   : find differences

pldiff -h
   : print this information
```

but you can also use the program as a library and invoke the predicate:

```
pldiff( <filename> , <filename> )
```

131.2 Known bugs and planned improvements (pldiff)

- Currently uses variant/2 to compare clauses. This is useful, but there should be an option to select the way clauses are compared, e.g., some form of equivalence defined by the user.

132 The Ciao lpmake scripting facility

Author(s): Manuel Hermenegildo.

Note: `lpmake` and the `make` library are still under active development, and they may change substantially in future releases.

`lpmake` is a Ciao application which uses the Ciao `make` library to implement a dependency-driven scripts in a similar way to the Unix `make` facility.

The original purpose of the Unix `make` utility was to determine automatically which pieces of a large program needed to be recompiled, and issue the commands to recompile them. In fact, `make` is often used for many other purposes: it can be used to describe any task where some files must be updated automatically from others whenever the others change.

To prepare to use `lpmake`, you must write a file (typically called `Makefile.pl`) that describes the relationships among files in your program or application, and states the commands for updating each file. In a program, typically the executable file is updated from object files, which are in turn made by compiling source files.

Once a suitable makefile exists, each time you change some source files, simply typing `lpmake` suffices to perform all necessary recompilations. The `make` program uses the makefile data base and the last modification times of the files to decide which of the files need to be updated. For each of those files, it issues the commands recorded in the data base.

`lpmake` executes commands in the `Makefile.pl` to update one or more target *names*, where *name* is typically a program, but can also be a file to be generated or even a "virtual" target. If no `-l` or `-m` options are present, `lpmake` will look for the makefile `Makefile.pl`.

`lpmake` updates a target if it depends on prerequisite files that have been modified since the target was last modified, or if the target does not exist.

You can provide command line arguments to `lpmake` to control which files should be regenerated, or how.

Note: if you use `make` and `functions`, then `make` should appear before `functions` in the list of packages.

133 Find out which architecture we are running on

Author(s): Manuel Carro, Robert Manchek.

Version: 0.0 (1998/8/12, 19:53:41 MET DST)

The architecure and operating system the engine is compiled for determines whether we can use or not certain libraries. This script, taken from a PVM distribution, uses a heuristic (which may need to be tuned from time to time) to find out the platform. It returns a string which is used throughout the engine (in #ifdefs) to enable/disable certain characteristics.

133.1 Usage (ciao_get_arch)

```
Usage: ciao_get_arch
```

133.2 More details

Look at the script itself...

PART XI - Contributed libraries

This part includes a number of libraries which have contributed by users of the Ciao system. Over time, some of these libraries are moved to the main library directories of the system.

134 mycin (library)

134.1 Usage and interface (`mycin`)

- **Library usage:**
  `:- use_package(mycin).`
  or
  `:- module(...,...,[mycin]).`
- **Exports:**
  - *Predicates:*
    `mycin/2`.
- **New operators defined:**
  `cf/2` [700,xfy], `extern/1` [1150,fx].
- **Other modules used:**
  - *System library modules:*
    `aggregates`, `dynamic`, `iso_misc`, `iso_byte_char`, `iso_incomplete`, `operators`, `read`, `streams`, `write`, `mycin/mycin_rt`.

134.2 Documentation on exports (`mycin`)

mycin/2: PREDICATE

No further documentation available for this predicate.

PART XII - Appendices

These appendices describe the installation of the Ciao environment on different systems and some other issues such as reporting bugs, signing up on the Ciao user's mailing list, downloading new versions, limitations, etc.

135 Installing Ciao from the source distribution

Author(s): Manuel Carro, Daniel Cabeza, Manuel Hermenegildo.

Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

Version of last change: 1.5#104 (2000/4/3, 10:41:2 CEST)

This describes the installation procedure for the Ciao Prolog system, including libraries and manuals, from a *source* distribution. This applies primarily to Unix-type systems (Linux, SunOS, Solaris, etc.). However, the sources can also be compiled on Windows NT/95/98 systems – see Section 135.6 [Installation and compilation under Windows], page 514 for details.

If you find any problems during installation, please refer to Section 135.8 [Troubleshooting (nasty messages and nifty workarounds)], page 515. See also Section 137.3 [Downloading new versions], page 521 and Section 137.4 [Reporting bugs], page 522.

135.1 Un*x installation summary

Note: it is recommended that you read the full installation instructions (specially if the installation will be shared by different architectures). However, in many cases it suffices to follow this summary:

1. Uncompress and unpackage (using `gunzip` and `tar -xpf`) the distribution. This will put everything in a new directory whose name reflects the Ciao version.
2. Enter the newly created directory (`SRC`). Edit `SETTINGS` and check/set the variables `SRC`, `BINROOT` (where the executables will go), `LIBROOT` (where the libraries will go), and `DOCROOT` (where the documentation will go, preferably a directory *accessible via* WWW).
3. Type `gmake install`. This will build executables, compile libraries, and install everything in a directory `LIBROOT/ciao` and in `BINROOT`.

   Note that `gmake` refers to the GNU implementation of the make Un*x command, which is available in many systems (including all Linux systems) simply as `make`. I.e., you can try simply typing `make install` if `gmake install` does not work. If typing `make` stops right away with error messages it is probably an older version and you need to install `gmake`.
4. Make the following modifications in your startup scripts. This will make the documentation accessible, set the correct mode when opening Ciao source files in `emacs`, etc. (`<LIBROOT>` must be replaced with the appropriate value):
   - For users a *csh-compatible shell* (`csh`, `tcsh`, ...), add to `~/.cshrc`:

     ```
     if ( -e <LIBROOT>/ciao/DOTcshrc ) then
        source <LIBROOT>/ciao/DOTcshrc
     endif
     ```
   - For users of an *sh-compatible shell* (`sh`, `bash`, ...), add to `~/.profile`:

     ```
     if [ -f <LIBROOT>/ciao/DOTprofile ]; then
        . <LIBROOT>/ciao/DOTprofile
     fi
     ```

     This will set up things so that the Ciao executables are found and you can access the Ciao system manuals using the `info` command. Note that, depending on your shell, *you may have to log out and back in* for the changes to take effect.
   - Also, if you use `emacs` (highly recommended) add this line to your `~/.emacs` file:

     ```
     (load-file "<LIBROOT>/ciao/DOTemacs.el")
     ```

   If you are installing Ciao globally in a multi-user machine, make sure that you instruct all users to do the same. If you are the system administrator, the previous steps can be done once and for all, and globally for all users by including the lines above in the central startup scripts (e.g., in Linux `/etc/bashrc`, `/etc/csh.login`, `/etc/csh.cshrc`, `/etc/skel`, `/usr/share/emacs/.../lisp/site-init.pl`, etc.).

5. Finally, if the (freely available) `emacs` editor/environment is not installed in your system, we *highly recommend* that you also install it at this point (see Section 135.2 [Un*x full installation instructions], page 510 for instructions). While it is easy to use Ciao with any editor of your choice, the Ciao distribution includes a very powerful *application development environment* which is based on `emacs` and which enables, e.g., source-level debugging, syntax coloring, context-sensitive on-line help, etc.
6. You may want now want to check your installation (see Section 135.3 [Checking for correct installation on Un*x], page 512) and read the documentation, which is stored in `DOCROOT` (copied from `SRC/doc/reference`) and can be easily accessed as explained in that same section. There are special "getting started" sections at the beginning of the manual.
7. If you have any problems you may want to check the rest of the instructions. The system can be *uninstalled* by typing `gmake uninstall`.

135.2 Un*x full installation instructions

1. **Uncompress and unpackage:** (using `gunzip` and `tar -xpf`) the distribution in a suitable directory. This will create a new directory called `ciao-X.Y`, where `X.Y` is the version number of the distribution. The `-p` option in the `tar` command ensures that the relative dates of the files in the package are preserved, which is needed for correct operation of the Makefiles.
2. **Select installation options:** Edit the file `SETTINGS` and set the following variables:
   - `SRC`: directory where the sources are stored.
   - `BINROOT`: directory where the Ciao executables will go. For example, if `BINROOT=/usr/local/bin`, then the Ciao compiler (`ciaoc`) will be stored at `/usr/local/bin/ciaoc`. Actually, it will be a link to `ciaoc-`*VersionNumber*. This applies also to other executables below and is done so that several versions of Ciao can coexist on the same machine. Note that the *version installed latest* will be the one started by default when typing `ciao`, `ciaoc`, etc.
   - `LIBROOT`: directory where the run-time libraries will be installed. The Ciao installation procedure will create a new subdirectory `ciao` below `LIBROOT` and a subdirectory below this one for each Ciao version installed. For example, if `LIBROOT=/usr/local/lib` and you have Ciao version `x.y`, then the libraries will be installed under `/usr/local/lib/ciao/ciao-x.y`. This allows you to install site-specific programs under `/usr/local/lib/ciao` and they will not be overwritten if a new version of Ciao is installed. It also again allows having several Ciao versions installed simultaneously.
   - `DOCROOT`: directory where the manuals will be installed. It is often convenient if this directory is accessible via WWW (`DOCROOT=/home/httpd/html/ciao`, or something like that).

   For network-based installations, it is of *utmost importance* that the paths given be reachable in all the networked machines. Different machines with different architectures can share the same physical `SRC` directory during installation, since compilations for different architectures take place in dedicated subdirectories. Also, different machines/architectures can share the same `LIBROOT` directory. This saves space since the architecture-independent libraries will be shared. See Section 135.5 [Multiarchitecture support], page 513 below.
3. **Compile Ciao:** At the ciao top level directory type `gmake all`.

   Important: use GNU make (`gmake`), not the standard UNIX make, as the latter does not support some features used during the compilation. It does not matter if the name of the executable is `make` or `gmake`: only make sure that it is GNU make.

   This will:

- Build an engine in `$(SRC)/bin/$(CIAOARCH)`, where `$(CIAOARCH)` depends on the architecture. The engine is the actual interpreter of the low level code into which Ciao Prolog programs are compiled.
- Build a new Ciao standalone compiler (`ciaoc`), with the default paths set for your local configuration (nonetheless, these can be overridden by environment variables, as described below).
- Precompile all the libraries under `$(SRC)/lib` and `$(SRC)/library` using this compiler.
- Compile a toplevel Prolog shell and a shell for Prolog scripts, under the `$(SRC)/shell` directory.
- Compile some small, auxiliary applications (contained in the `etc` directory, and documented in the part of the manual on 'Miscellaneous Standalone Utilities').

This step can be repeated successively for several architectures in the same source directory. Only the engine and some small parts of the libraries (those written in C) differ from one architecture to the other. Standard Ciao Prolog code compiles into bytecode object files (`.po`) and/or executables which are portable among machines of different architecture, provided there is an executable engine accessible in every such machine. See more details below under Section 135.5 [Multiarchitecture support], page 513.

4. **Check compilation:** If the above steps have been satisfactorily finished, the compiler has compiled itself and all the distribution modules, and very probably everything is fine. To further check this before final installation you can type `gmake test`, which will compile and execute a small program containing several tests.
5. **Install Ciao:** To install Ciao in the directories selected in the file `SETTINGS` during step 2 above, type `gmake justinstall`. This will:
   - Install the executables of the Ciao program development tools (i.e., the general driver/top-level `ciao`, the standalone compiler `ciaoc`, the script interpreter `ciao-shell`, miscellaneous utilities, etc.) in `BINROOT` (see below). In order to use these tools, the `PATH` environment variable of users needs to contain the path `BINROOT`.
   - Install the Ciao libraries under `LIBROOT/ciao` (these will be automatically found).
   - Install under `DOCROOT` the Ciao manuals in several formats (such as GNU `info`, `html`, `postscript`, etc.), depending on the distribution. In order for these manuals to be found when typing `M-x info` within `emacs`, or by the standalone `info` and `man` commands, the `MANPATH` and `INFOPATH` environment variables of users both need to contain the path `DOCROOT`.
   - Install under `LIBROOT/ciao` the Ciao GNU `emacs` interface (`ciao.el`, which provides an interactive interface to the Ciao program development tools, as well as some other auxiliary files) and a file `DOTemacs` containing the `emacs` initialization commands which are needed in order to use the Ciao `emacs` interface.
6. **Set up user environments:** In order to automate the process of setting the variables above, the installation process leaves the files `LIBROOT/ciao/DOTcshrc` (for `csh`-like shells), `LIBROOT/ciao/DOTprofile` (for `sh`-like shells), and `LIBROOT/ciao/DOTemacs` (for emacs) with appropriate definitions which will take care of all needed environment variable definitions and emacs mode setup. Make the following modifications in your startup scripts, so that these files are used (`<LIBROOT>` must be replaced with the appropriate value):
   - For users a *csh-compatible shell* (`csh`, `tcsh`, ...), add to `~/.cshrc`:

```
if ( -e <LIBROOT>/ciao/DOTcshrc ) then
   source <LIBROOT>/ciao/DOTcshrc
endif
```

   - For users of an *sh-compatible shell* (`sh`, `bash`, ...), add to `~/.profile`:

```
if [ -f <LIBROOT>/ciao/DOTprofile ]; then
   . <LIBROOT>/ciao/DOTprofile
fi
```

   This will set up things so that the Ciao executables are found and you can access the Ciao system manuals using the `info` command. Note that, depending on your shell, *you may have to log out and back in* for the changes to take effect.

   - Also, if you use `emacs` (highly recommended) add this line to your `~/.emacs` file:

```
(load-file "<LIBROOT>/ciao/DOTemacs.el")
```

   If you are installing Ciao globally in a multi-user machine, make sure that you instruct all users to do the same. If you are the system administrator, the previous steps can be done once and for all, and globally for all users by including the lines above in the central startup scripts (e.g., in Linux `/etc/bashrc`, `/etc/csh.login`, `/etc/csh.cshrc`, `/etc/skel`, `/usr/share/emacs/.../lisp/site-init.pl`, etc.).

7. **Download and install Emacs (highly recommended):** If the (freely available) `emacs` editor is not installed in your system, its installation is *highly recommended* (if you are installing in a multi-user machine, you may want to do it in a general area so that it is available for other users, even if you do not use it yourself). While it is easy to use Ciao with any editor of your choice, the Ciao distribution includes a very powerful *application development environment* which is based on `emacs` and which enables, e.g., source-level debugging, syntax coloring, context-sensitive on-line help, etc.

   The emacs editor (in all its versions: Un*x, Windows, etc.) can be downloaded from, for example, `http://www.emacs.org/`, and also from the many GNU mirror sites worldwide (See `http://www.gnu.org/` for a list), in the `gnu/emacs` and `gnu/windows/emacs` directories. For example, at UPM it can be found at `ftp://ftp.fi.upm.es/pub/mirrors/gnu/emacs` and `ftp://ftp.fi.upm.es/pub/mirrors/gnu/windows/emacs`. You can find answers to frequently asked questions (FAQ) about `emacs` in general at `http://www.gnu.org/software/emacs/emacs-faq.text` and about the Windows version at `http://www.gnu.org/software/emacs/windows/ntemacs.html` (despite the `ntemacs` name it runs fine also as is on Win9X and Win2000 machines).

8. **Check installation / read documentation:** You may now want to check your installation (see Section 135.3 [Checking for correct installation on Un*x], page 512) and read the documentation, which is stored in `DOCROOT` (copied from `SRC/doc/reference`) and can be easily accessed as explained that same section. There are special "getting started" sections at the beginning of the manual.

Other useful `make` targets are listed at the beginning of `$(SRC)/Makefile`.

If you have any problems you may want to check Section 135.8 [Troubleshooting (nasty messages and nifty workarounds)], page 515.

The system can be *uninstalled* by typing `gmake uninstall` in the top directory (the variables in `SETTINGS` should have the same value as when the install was performed, so that the same directories are cleaned).

135.3 Checking for correct installation on Un*x

If everything has gone well, several applications and tools should be available to a normal user. Try the following while logged in as a *normal user* (important in order to check that permissions are set up correctly):

- Typing `ciao` (or `ciaosh`) should start the typical Prolog top-level shell.
- In the top-level shell, Prolog library modules should load correctly. Type for example `use_module(library(dec10_io))` –you should get back a prompt with no errors reported.
- To exit the top level shell, type `halt.` as usual, or ⟨^D⟩.

- Typing `ciaoc` should produce the help message from the Ciao standalone compiler.
- Typing `ciao-shell` should produce a message saying that no code was found. This is a Ciao application which can be used to write scripts written in Prolog, i.e., files which do not need any explicit compilation to be run.

Also, the following documentation-related actions should work:

- If the `info` program is installed, typing `info` should produce a list of manuals which *should include Ciao manual(s) in a separate area* (you may need to log out and back in so that your shell variables are reinitialized for this to work).
- Opening with a WWW browser (e.g., `netscape`) the directory or `URL` corresponding to the `DOCROOT` setting should show a series of Ciao-related manuals. Note that *style sheets* should be activated for correct formatting of the manual.
- Typing `man ciao` should produce a man page with some very basic general information on Ciao (and pointing to the on-line manuals).
- The `DOCROOT` directory should contains the manual also in the other formats such as `postscript` or `pdf` which specially useful for printing. See Section 2.2.7 [Printing manuals (Un*x)], page 17 for instructions.

Finally, if `emacs` is installed, after starting it (typing `emacs`) the following should work:

- Typing ⟨^H⟩ ⟨i⟩ (or in the menus `Help->Manuals->Browse Manuals with Info`) should open a list of manuals in info format in which the Ciao manual(s) should appear.
- When opening a Prolog file, i.e., a file with `.pl` or `.pls` ending, using ⟨^X⟩⟨^F⟩`filename` (or using the menus) the code should appear highlighted according to syntax (e.g., comments in red), and `Ciao/Prolog` menus should appear in the menu bar on top of the `emacs` window.
- Loading the file using the `Ciao/Prolog` menu (or typing ⟨^C⟩ ⟨l⟩) should start in another emacs buffer the Ciao toplevel shell and load the file. You should now be able to switch the the toplevel shell and make queries from within `emacs`.

Note: when using `emacs` it is *very convenient* to swap the locations of the (normally not very useful) ⟨Caps Lock⟩ key and the (very useful in `emacs`) ⟨Ctrl⟩ key on the keyboard. How to do this is explained in the `emacs` frequently asked questions FAQs (see the `emacs` download instructions for their location).

135.4 Cleaning up the source directory

After installation, the source directory can be cleaned up in several ways:

- `gmake uninstall` removes the installation but does not touch the source directories.
- `gmake totalclean` leaves the distribution is its original form, throwing away any intermediate files (as well as any unneeded files left behind by the Ciao developers), while still allowing recompilation.

Other useful `make` targets are listed at the beginning of `$(SRC)/Makefile`.

135.5 Multiarchitecture support

As mentioned before, Ciao applications (including the compiler and the top level) can run on several machines with different architectures without any need for recompiling, provided there is one Ciao engine (compiled for the corresponding architecture) accessible in each machine. Also, the Ciao libraries (installed in `LIBROOT`, which contain also the engines) and the actual binaries (installed in `BINROOT`) can themselves be shared on several machines with different architectures, saving disk space.

For example, assume that the compiler is installed as:

`/usr/local/share/bin/ciaoc`

and the libraries are installed under

`/usr/local/share/lib`

Assume also that the `/usr/local/share` directory is mounted on, say, a number of Linux and a number of Solaris boxes. In order for `ciaoc` to run correctly on both types of machines, the following is needed:

1. Make sure you that have done `gmake install` on one machine of each architecture (once for Linux and once for Solaris in our example). This recompiles and installs a new engine and any architecture-dependent parts of the libraries for each architecture. The engines will have names such as `ciaoengine.LINUXi86`, `ciaoengine.SolarisSparc`, and so on.
2. In multi-architecture environments it is even more important to make sure that users make the modifications to their startup scripts using `<LIBROOT>/ciao/DOTcshrc` etc. The selection of the engine (and architecture-dependent parts of libraries) is done in these scripts by setting the environment variable `CIAOARCH`, using the `ciao_get_arch` command, which is installed automatically when installing Ciao. This will set `CIAOARCH` to, say, `LINUXi86`, `SolarisSparc`, respectively, and `CIAOENGINE` will be set to `ciaoengine.`*CIAOARCH*.

   However, note that this is not strictly necessary if running on only one architecture: if `CIAOARCH` is not set (i.e., undefined), the Ciao executables will look simply for `ciaoengine`, which is always a link to the latest engine installed in the libraries. But including the initialization files provided has the advantage of setting also paths for the manuals, etc.

135.6 Installation and compilation under Windows

There are two possibilities in order to install Ciao Prolog on Windows NT/95/98 machines:

- Installing from the Windows *precompiled* distribution. This is the easiest since it requires no compilation and is highly recommended. This is described in Chapter 136 [Installing Ciao from a Win32 binary distribution], page 519.
- Installing the standard Ciao Prolog (Un*x) system source distribution and compiling it under Windows. This is somewhat more complex and currently requires the (freely available) Cygnus Win32 development libraries –described below.

In order to compile Ciao Prolog for Win32 environments you need to have the (public domain) *Cygnus Win32* and development libraries installed in your system. Compilation should be performed preferably under Windows NT.

- Thus, the first step, if Cygnus Win32 is not installed in your system, is to download it (from, e.g., `http://www.cygnus.com/misc/gnu-win32`) and install it. The compilation process also requires that the executables `rm.exe`, `sh.exe`, and `uname.exe` from the Cygnus distribution be copied under `/bin` prior to starting the process (if these executables are not available under `/bin` the compilation process will produce a number of errors and eventually stop prematurely).
- Assuming all of the above is installed, type `make allwin32`. This will compile both the engine and the Prolog libraries. In this process, system libraries that are normally linked dynamically under Un*x (i.e., those for which `.so` dynamically loadable files are generated) are linked statically into the engine (this is done instead of generating `.dll`s because of a limitation in the current version of the Cygnus Win32 environment). No actual installation is made at this point, i.e., this process leaves things in a similar state as if you had just downloaded and uncompressed the precompiled distribution. Thus, in order to complete the installation you should now:
- Follow now the instructions in Chapter 136 [Installing Ciao from a Win32 binary distribution], page 519.

A further note regarding the executables generated by the Ciao compiler and top-level: the same considerations given in Chapter 136 [Installing Ciao from a Win32 binary distribution], page 519 apply regarding `.bat` files, etc. However, in a system in which Cygnus Win32 is installed these executables can also be used in a very simple way. In fact, the executables can be run as in Un*x by simply typing their name at the `bash` shell command line without any associated `.bat` files. This only requires that the `bash` shell which comes with Cygnus Win32 be installed and accessible: simply, make sure that `/bin/sh.exe` exists.

135.7 Porting to currently unsupported operating systems

If you would like to port Ciao to a currently unsupported platform, there are several issues to take into account. The main one is to get the *engine* to compile in that platform, i.e., the C code under the `engine` directory. To this end, look under the directory `$(SRC)/makefile-sysdep`: there is a `mkf-CIAOARCH` file for every architecture/OS under which Ciao is known to compile. You can start from the closest one and customize it for your local distribution or installed tools. But please keep the original files! Once a working engine is achieved, it should be possible to continue with the standard installation procedure, which will try to use a completely static version of the standalone compiler (`ciaoc.sta` in the `ciaoc` directory) to compile the interactive top-level (`ciaosh`) and a new version of the standalone compiler (`ciaoc`). These in turn should be able to compile the Prolog libraries. You may also need to look at some libraries (such as, for example, `sockets`) which contain C code. Porting to other Un*x flavours should be easy while other operating system families may require more work. If you do succeed in porting to a platform that is currently unsupported please send the `mkf-CIAOARCH` and any patches to `ciao@clip.dia.fi.upm.es`, and we will include them (with due credit, of course) in the next distribution.

135.8 Troubleshooting (nasty messages and nifty workarounds)

The following a list of common installation problems reported by users:

- **Problem:** Compilation errors appear when trying a new installation/compilation after the previous one was aborted (e.g., because of errors).

  Possible reason and solution: It is a good idea to clean up any leftovers from the previous compilation using `make engclean` before restarting the installation or compilation process.
- **Problem:**

  During engine compilation, messages such as the following appear: `tasks.c:102:PTHREAD_CANCEL_ASYNCHRONOUS undeclared (first use of this function)`.

  Possible reason and solution:

  Your (Linux?) system does not have (yet) the Posix threads library installed. You can upgrade to one which does have it, or download the library from

  `http://pauillac.inria.fr/~xleroy/linuxthreads/index.html`

  and install it, or disable the use of threads in Linux: for this, edit the `SETTINGS` file and specify `USE_THREADS=no`, which will avoid linking against thread libraries (it will disable the use of thread-related primitives as well). Clean the engine with `make engclean` and restart compilation.

  If you have any alternative threads library available, you can tinker with `engine/threads.h` and the files under `makefile-sysdep` in order to get the task managing macros right for your system. Be sure to link the right library. If you succeed, we (`ciao@clip.dia.fi.upm.es`) will be happy of knowing about what you have done.
- **Problem:**

  `-lpthread: library not found` (or similar)

Possible reason and solution:

Your (Linux?) system seems to have Posix threads installed, but there is no threads library in the system. In newer releases (e.g., RedHat 5.0), the Posix threads system calls have been included in `glibc.so`, so specifying `-lpthread` in `makefile-sysdep/mkf-LINUX` is not needed; remove it. `make engclean` and restart installation.

Alternatively, you may have made a custom installation of Posix threads in a non-standard location: be sure to include the flag `-L/this/is/where/the/posix/libraries/are` **before** `-lpthread`, and to update `/etc/ld.so.conf` (see `man ldconfig`).

- **Problem:**

  `Segmentation Violation` (when starting the first executable)

  Possible reason and solution:

  This has been observed with certain older versions of `gcc` which generated erroneous code under full optimization. The best solution is to upgrade to a newer version of `gcc`. Alternatively, lowering the level of optimization (by editing the `SETTINGS` file in the main directory of the distribution) normally solves the problem, at the cost of reduced execution speed.

- **Problem:** `ciaoc: /home/clip/lib/ciao/ciao-X.Y/engine/ciaoengine: not found`

  Possible reason and solution:

  - The system was not fully installed and the variable `CIAOENGINE` was not set.
  - The system was installed, the variable `CIAOENGINE` is set, but it is does not point to a valid ciaoengine.

  See the file `LIBROOT/ciao/DOTcshrc` for user settings for environment variables.

- **Problem:**

  `ERROR: File library(compiler) not found - aborting...` (or any other library is not found)

  Possible reason and solution:

  - The system was not installed and the variable `CIAOLIB` was not set.
  - The system is installed and the variable `CIAOLIB` is wrong.

  See the file `LIBROOT/ciao/DOTcshrc` for user settings for environment variables.

- **Problem:**

  `ERROR: File <some_directory>/<some_file>.itf not found - aborting...`

  Possible reason and solution:

  Can appear when compiling `.pl` files. The file to compile (<some_file>.pl) is not in the directory <some_directory>. You gave a wrong file name or you are in the wrong directory.

- **Problem:**

  `*ERROR*: /(write_option,1) is not a regular type` (and similar ones)

  Possible reason and solution:

  This is not a problem, but rather the type checker catching some minor inconsistencies which may appear while compiling the libraries. Bug us to remove it, but ignore it for now.

- **Problem:**

  `WARNING: Predicate <some_predicate>/<N> undefined in module <some_module>`

  Possible reason and solution:

  It can appear when the compiler is compiling Ciao library modules. If so, ignore it (we will fix it). If it appears when compiling user programs or modules, you may want to check your program for those undefined predicates.

- **Problem:**

  `gmake[1]: execve: /home/clip/mcarro/ciao-0.7p2/etc/collect_modules: No such file or directory`

Possible reason and solution:

Check if collect_modules is in $(SRC)/etc and is executable. If it is not here, your distribution is incorrect: please let us know.

- **Problem:**

  `make: Fatal error in reader: COMMON, line 12: Unexpected end of line seen`

  Possible reason and solution:

  You are using standard Un*x make, not GNU's make implementation (gmake).

- **Problem:**

  `WARNING`s or `ERROR`s while compiling the Ciao libraries during installation.

  Possible reason and solution:

  It is possible that you will see some such errors while compiling the Ciao libraries during installation. This is specially the case if you are installing a Beta or Alpha release of Ciao. These releases (which have "odd" version numbers such as 1.5 or 2.1) are typically snapshots of the development directories, on which many developers are working simultaneously, which may include libraries which have typically not been tested yet as much as the "official" distributions (those with "even" version numbers such as 1.6 or 2.8). Thus, minor warnings may not have been eliminated yet or even errors can sneak in. These warnings and errors should not affect the overall operation of the system (e.g., if you do not use the affected library).

136 Installing Ciao from a Win32 binary distribution

Author(s): Daniel Cabeza, Manuel Carro, Manuel Hermenegildo.
Version: 1.5#128 (2000/5/2, 18:4:14 CEST)
Version of last change: 1.5#92 (2000/3/28, 17:41:25 CEST)

137 Beyond installation

Author(s): Manuel Carro, Daniel Cabeza, Manuel Hermenegildo.
Version: 1.5#118 (2000/4/19, 18:13:43 CEST)

137.1 Architecture-specific notes and limitations

Ciao makes use of advanced characteristics of modern architectures and operating systems such as multithreading, shared memory, sockets, locks, dynamic load libraries, etc., some of which are sometimes not present in a given system and others may be implemented in very different ways across the different systems. As a result, currently not all Ciao features are available in all supported operating systems. Sometimes this is because not all the required features are present in all the OS flavors supported and sometimes because we simply have not had the time to port them yet.

The current state of matters is as follows:

LINUX: multithreading, shared DB access, and locking working.

Solaris: multithreading, shared DB access, and locking working.

SunOS 4: multithreading, shared DB access, and locking NOT working.

Win 95/NT:
multithreading working. Shared DB access, locking, and dynamic linking of object code (C) libraries NOT working.

The features that do not work are disabled at compile time. If problems appear, please disable them explicitly in the `SETTINGS` file. We will make this more automatic sometime in the future.

137.2 Keeping up to date with the Ciao users mailing list

We recommend that you join the Ciao *users mailing list* (`ciao-users@clip.dia.fi.upm.es`), in order to receive information on new versions and solutions to problems. Simply send a message to `ciao-users-request@clip.dia.fi.upm.es`, containing in the body only the word:

```
subscribe
```

alone in one line. Messages in the list are strictly limited to issues directly related to Ciao Prolog and your email address will of course be kept strictly confidential. You mail also want to subscribe to the `comp.lang.prolog` newsgroup.

There is additional info available on the Ciao system, other CLIP group software, publications on the technology underlying these systems, etc. in the CLIP group's WWW site `http://clip.dia.fi.upm.es`.

137.3 Downloading new versions

Ciao and its related libraries and utilities are under constant improvement, so you should make sure that you have the latest versions of the different components, which can be dowloaded from:

```
http://clip.dia.fi.upm.es/Software
```

137.4 Reporting bugs

If you still have problems after downloading the latest version and reading the installation instructions you can send a message to `ciao-bug@clip.dia.fi.upm.es`. Please be as informative as possible in your messages, so that we can reproduce the bug.

- For *installation problems* we typically need to have the version and patch number of the Ciao package (e.g., the name of the file downloaded), the output produced by the installation process (you can capture it by redirecting the output into a file or cutting and pasting with the mouse), and the exact version of the Operating System you are using (as well as the C compiler, if you took a source distribution).
- For *problems during use* we also need the Ciao and OS versions and a small example of code which we can run to reproduce the bug.

References

[AAF91] J. Almgren, S. Andersson, L. Flood, C. Frisk, H. Nilsson, and J. Sundberg. *Sicstus Prolog Library Manual.* Po Box 1263, S-16313 Spanga, Sweden, October 1991.

[AKNL86] Hassan Ait-Kaci, Roger Nasr, and Pat Lincoln. E An Overview. Technical Report AI-420-86-P, Microelectronics and Computer Technology Corporation, 9430 Research Boulevard, Austin, TX 78759, December 1986.

[Apt97] K. Apt, editor. *From Logic Programming to Prolog.* Prentice-Hall, Hemel Hempstead, Hertfordshire, England, 1997.

[BA82] M. Ben-Ari. *Principles of Concurrent Programming.* Prentice Hall International, 1982.

[BBP81] D.L. Bowen, L. Byrd, L.M. Pereira, F.C.N. Pereira, and D.H.D. Warren. Decsystem-10 prolog user's manual. Technical report, Department of Artificial Intelligence, University of Edinburgh, October 1981.

[BdlBH99] F. Bueno, M.˜Garc\'\ia de˜la Banda, and M. Hermenegildo. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. *ACM Transactions on Programming Languages and Systems*, 21(2):189–238, March 1999.

[BLGPH99] F. Bueno, P. L\'opez-Garc\'\ia, G. Puebla, and M. Hermenegildo. The Ciao Prolog Preprocessor. Technical Report CLIP8/95.0.7.20, Technical University of Madrid (UPM), Facultad de Inform\'atica, 28660 Boadilla del Monte, Madrid, Spain, November 1999.

[Bue95] F. Bueno. The CIAO Multiparadigm Compiler: A User's Manual. Technical Report CLIP8/95.0, Facultad de Inform\'atica, UPM, June 1995.

[Byr80] L. Byrd. Understanding the Control Flow of Prolog Programs. In S.-A. T\"arnlund, editor, *Workshop on Logic Programming*, Debrecen, 1980.

[Car87] M. Carlsson. Freeze, Indexing, and Other Implementation Issues in the Wam. In *Fourth International Conference on Logic Programming*, pages 40–58. University of Melbourne, MIT Press, May 1987.

[Car88] M. Carlsson. *Sicstus Prolog User's Manual.* Po Box 1263, S-16313 Spanga, Sweden, February 1988.

[CCG98] I. Caballero, D. Cabeza, S. Genaim, J.M. Gomez, and M. Hermenegildo. persdb_sql: SQL Persistent Database Interface. Technical Report D3.1.M2-A2 CLIP10/98.0, RADIOWEB Project, December 1998.

[CGH93] M. Carro, L. G\'omez, and M. Hermenegildo. Some Paradigms for Visualizing Parallel Execution of Logic Programs. In *1993 International Conference on Logic Programming*, pages 184–201. MIT Press, June 1993.

[CH97] D. Cabeza and M. Hermenegildo.
WWW Programming using Computational Logic Systems (and the PILLOW/CIAO Library).
In *Proceedings of the Workshop on Logic Programming and the WWW at WWW6*, San Francisco, CA, April 1997.

[CH99a] D. Cabeza and M. Hermenegildo.
A New Module System for Prolog.
In *ICLP'99 WS on Parallelism and Implementation of (C)LP Systems*, pages 110–128. N.M. State U., December 1999.

[CH99b] D. Cabeza and M. Hermenegildo.
The Ciao Modular Compiler and Its Generic Program Processing Library.
In *ICLP'99 WS on Parallelism and Implementation of (C)LP Systems*, pages 147–164. N.M. State U., December 1999.

[CH00a] D. Cabeza and M. Hermenegildo.
The Ciao Modular, Standalone Compiler and Its Generic Program Processing Library.
In *Special Issue on Parallelism and Implementation of (C)LP Systems. To appear*, Electronic Notes in Theoretical Computer Science. Elsevier - North Holland, 2000.

[CH00b] D. Cabeza and M. Hermenegildo.
The Ciao Module System: A New Module System for Prolog.
In *Special Issue on Parallelism and Implementation of (C)LP Systems. To appear*, Electronic Notes in Theoretical Computer Science. Elsevier - North Holland, 2000.

[CH00c] M. Carro and M. Hermenegildo.
Tools for Constraint Visualization: The VIFID/TRIFID Tool.
In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*. Springer-Verlag, 2000.
To appear.

[CH00d] M. Carro and M. Hermenegildo.
Tools for Search Tree Visualization: The APT Tool.
In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*. Springer-Verlag, 2000.
To appear.

[CHGT98] D. Cabeza, M. Hermenegildo, S. Genaim, and C. Taboch.
Design of a Generic, Homogeneous Interface to Relational Databases.
Technical Report D3.1.M1-A1, CLIP7/98.0, RADIOWEB Project, September 1998.

[CHV96a] D. Cabeza, M. Hermenegildo, and S. Varma.
The PiLLoW/CIAO Library for INTERNET/WWW Programming using Computational Logic Systems.
In *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications*, pages 72–90, JICSLP'96, Bonn, September 1996.
Text and code available from \htmladdnormallink\tt http://www.clip.dia.fi.upm.es/miscdocs/pillow/pillow.html http://www.clip.dia.fi.upm.es/miscdocs/pillow/pillow.html.

[CHV96b] D. Cabeza, M. Hermenegildo, and S. Varma.
The \sf P\em i\sf LL\em o\sf W/CIAO Library for INTERNET/WWW Programming using Computational Logic Systems.
In *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications*, JICSLP'96, Bonn, September 1996.
Available from \htmladdnormallink\tt http://clement.info.umoncton.ca/\~lpnet http://clement.info.umoncton.ca/\~lpnet.

[CLI95] The CLIP Group.
CIAO Compiler: Distributed Execution and Low Level Support Subsystem.
Public Software, ACCLAIM Deliverable D4.3/2-A3, Facultad de Inform\'atica, UPM, June 1995.

[CM81] W.F. Clocksin and C.S. Mellish.
Programming in Prolog.
Springer-Verlag, 1981.

[Col78] A. Colmerauer.
Metamorphosis grammars.
In *Natural language communication with computers*, pages 133–189. Springer LNCS 63, 1978.

[Col82] A. Colmerauer et al.
Prolog II: Reference Manual and Theoretical Model.
Groupe D'intelligence Artificielle, Facult\'e Des Sciences De Luminy, Marseille, 1982.

[DEDC96] P. Deransart, A. Ed-Dbali, and L. Cervoni.
Prolog: The Standard.
Springer-Verlag, 1996.

[Dij65] E.W. Dijkstra.
Co-operating sequential processes.
In F. Genuys, editor, *Programming Languages.* Academic Press, London, 1965.

[DL93] S.K. Debray and N.W. Lin.
Cost analysis of logic programs.
ACM Transactions on Programming Languages and Systems, 15(5):826–875, November 1993.

[DLGH97] S.K. Debray, P. L\'opez-Garc\'\ia, and M. Hermenegildo.
Non-Failure Analysis for Logic Programs.
In *1997 International Conference on Logic Programming*, pages 48–62, Cambridge, MA, June 1997. MIT Press, Cambridge, MA.

[DLGHL97]
S.K. Debray, P. L\'opez-Garc\'\ia, M. Hermenegildo, and N.-W. Lin.
Lower Bound Cost Estimation for Logic Programs.
In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[GCH98] J.M. Gomez, D. Cabeza, and M. Hermenegildo.
WebDB: A Database WWW Interface.
Technical Report D3.1.M2-A3 CLIP11/98.0, RADIOWEB Project, December 1998.

[GdW94] J.P. Gallagher and D.A. de Waal.
Fast and precise regular approximations of logic programs.
In Pascal Van~Hentenryck, editor, *Proc.~of the 11th International Conference on Logic Programming*, pages 599–613. MIT Press, 1994.

[HBC96] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. Garc\'\ia de la Banda, P. L\'opez-Garc\'\ia, and G. Puebla.
The CIAO Multi-Dialect Compiler and System: A Demo and Status Report.
In *Proceedings of the JICSLP'96 Workshop on Parallelism and Implementation Technology.* Computer Science Department, Technical University of Madrid, September 1996.
Available from \htmladdnormallink\tt http://www.clip.dia.fi.upm.es/Projects/COMPULOG/meeting96/papers/PS/clip.ps.%

gz
http://www.clip.dia.fi.upm.es/Projects/COMPULOG/meeting96/papers/PS/clip.ps%
.gz.

[HBC99] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M.~Garc\'\ia de la Banda, P. L\'opez-Garc\'\ia, and G. Puebla.
The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems.
In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, April 1999.

[HBGP95] M. Hermenegildo, F. Bueno, M. Garc\'\ia de la Banda, and G. Puebla.
The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems.
In *Proceedings of the ILPS'95 Workshop on Visions for the Future of Logic Programming*, Portland, Oregon, USA, December 1995.
Available from \htmladdnormallink\tt http://www.clip.dia.fi.upm.es/ http://www.clip.dia.fi.upm.es/.

[HBPLG99]
M. Hermenegildo, F. Bueno, G. Puebla, and P. L\'opez-Garc\'\ia.
Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor.
In *1999 International Conference on Logic Programming*, pages 52–66, Cambridge, MA, November 1999. MIT Press.

[HC93] M. Hermenegildo and The CLIP Group.
Towards CIAO-Prolog – A Parallel Concurrent Constraint System.
In *Proc. of the Compulog Net Area Workshop on Parallelism and Implementation Technologies*. FIM/UPM, Madrid, Spain, June 1993.

[HC94] M. Hermenegildo and The CLIP Group.
Some Methodological Issues in the Design of CIAO - A Generic, Parallel, Concurrent Constraint System.
In *Principles and Practice of Constraint Programming*, number 874 in LNCS, pages 123–133. Springer-Verlag, May 1994.

[HC97] M. Hermenegildo and The CLIP Group.
An Automatic Documentation Generator for (C)LP – Reference Manual.
The Ciao System Documentation Series–TR CLIP5/97.3, Facultad de Inform\'atica, UPM, August 1997.

[HCC95] M. Hermenegildo, D. Cabeza, and M. Carro.
Using Attributed Variables in the Implementation of Concurrent and Parallel Logic Programming Systems.
In *Proc. of the Twelfth International Conference on Logic Programming*, pages 631–645. MIT Press, June 1995.

[Her86] M. Hermenegildo.
An Abstract Machine for Restricted AND-parallel Execution of Logic Programs.
In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 25–40. Imperial College, Springer-Verlag, July 1986.

[Her96] M. Hermenegildo.
Writing "Shell Scripts" in SICStus Prolog, April 1996.
Posting in \tt comp.lang.prolog. Available from \htmladdnormallink\tt http://www.clip.dia.fi.upm.es/ http://www.clip.dia.fi.upm.es/.

[**Her99a**] M. Hermenegildo.
A Documentation Generator for Logic Programming Systems.
Technical Report CLIP10/99.0, Facultad de Inform\'atica, UPM, September 1999.

[**Her99b**] M. Hermenegildo.
A Documentation Generator for Logic Programming Systems.
In *ICLP'99 Workshop on Logic Programming Environments*, pages 80–97. N.M. State University, December 1999.

[**HG90**] M. Hermenegildo and K. Greene.
\&-Prolog and its Performance: Exploiting Independent And-Parallelism.
In *1990 International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.

[**HG91**] M. Hermenegildo and K. Greene.
The \&-Prolog System: Exploiting Independent And-Parallelism.
New Generation Computing, 9(3,4):233–257, 1991.

[**Hog84**] C.~J. Hogger.
Introduction to Logic Programming.
Academic Press, London, 1984.

[**Hol90**] C. Holzbaur.
Specification of Constraint Based Inference Mechanisms through Extended Unification.
PhD thesis, University of Vienna, 1990.

[**Hol92**] C. Holzbaur.
Metastructures vs. Attributed Variables in the Context of Extensible Unification.
In *1992 International Symposium on Programming Language Implementation and Logic Programming*, pages 260–268. LNCS631, Springer Verlag, August 1992.

[**Hol94**] C. Holzbaur.
SICStus 2.1/DMCAI Clp 2.1.1 User's Manual.
University of Vienna, 1994.

[**JL88**] D. Jacobs and A. Langen.
Compilation of Logic Programs for Restricted And-Parallelism.
In *European Symposium on Programming*, pages 284–297, 1988.

[**Knu84**] D. Knuth.
Literate programming.
Computer Journal, 27:97–111, 1984.

[**LGHD96**] P. L\'opez-Garc\'\ia, M. Hermenegildo, and S.K. Debray.
A Methodology for Granularity Based Control of Parallelism in Logic Programs.
Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation, 22:715–734, 1996.

[**MH89**] K. Muthukumar and M. Hermenegildo.
Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation.
In *1989 North American Conference on Logic Programming*, pages 166–189. MIT Press, October 1989.

[**Nai85**] L. Naish.
The MU-Prolog 3.2 Reference Manual.
TR 85/11, Dept. of Computer Science, U. of Melbourne, October 1985.

[**Nai91**] L. Naish.
Adding equations to NU-Prolog.

In *Symp. on Progr. Language Impl. and Logic Progr (PLILP'91)*, LNCS 528, pages 15–26. Springer Verlag, 1991.

[Par97] The RADIOWEB˜Project Partners.
RADIOWEB EP25562: Automatic Generation of Web Sites for the Radio Brodcasting Industry – Project Description / Technical Annex.
Technical Report, RADIOWEB Project, July 1997.

[PBH97] G. Puebla, F. Bueno, and M. Hermenegildo.
An Assertion Language for Debugging of Constraint Logic Programs.
In *Proceedings of the ILPS'97 Workshop on Tools and Environments for (Constraint) Logic Programming*, October 1997.
Available from \htmladdnormallink\tt ftp://clip.dia.fi.upm.es/pub/papers/assert_lang_tr_discipldeliv.ps.gz ftp://clip.dia.fi.upm.es/pub/papers/assert_lang_tr_discipldeliv.ps.gz as technical report CLIP2/97.1.

[PBH00] G. Puebla, F. Bueno, and M. Hermenegildo.
An Assertion Language for Debugging of Constraint Logic Programs.
In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, LNCS. Springer-Verlag, 2000.
To appear.

[PH99] G. Puebla and M. Hermenegildo.
Some Issues in Analysis and Specialization of Modular Ciao-Prolog Programs.
In *ICLP'99 Workshop on Optimization and Implementation of Declarative Languages*, pages 45–61. U. of Southampton, U.K, November 1999.

[PW80] F.C.N. Pereira and D.H.D. Warren.
Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks.
Artificial Intelligence, 13:231–278, 1980.

[SS86] L. Sterling and E. Shapiro.
The Art of Prolog.
MIT Press, 1986.

[Swe95] Swedish Institute of Computer Science, P.O. Box 1263, S-16313 Spanga, Sweden.
Sicstus Prolog V3.0 User's Manual, 1995.

Library/Module Definition Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Predicate/Method Definition Index

!

$

&

,

-

.

;

=

|

>

^

\

<

A

B

C

D

E

F

G

H

I

J

P

Q

R

S

T

Property Definition Index

=

@

\

A

C

D

E

F

G

H

I

L

M

N

P

R

S

T

V

W

Regular Type Definition Index

A

B

C

D

E

F

G

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

Declaration Definition Index

Concept Definition Index

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Global Index

This is a global index containing pointers to places where concepts, predicates, modes, properties, types, applications, etc., are referred to in the text of the document. Note that due to limitations of the `info` format unfortunately only the first reference will appear in online versions of the document.

@

|

~

+

>

^

\

<

A

B

C

D

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X