ECL^{i}PS^e

User Manual

Release 4.2

Abderrahamane Aggoun (ECRC)
David Chan (ECRC)
Pierre Dufresne (ECRC)
Eamon Falvey (ICL-ITC)
Hugh Grant (ICL-ITC)
Alexander Herold (ECRC)
Geoffrey Macartney (ECRC)
Micha Meier (ECRC)
David Miller (ICL-ITC)
Shyam Mudambi (ECRC)
Stefano Novello (ECRC and IC-Parc)
Bruno Perez (ECRC)
Emmanuel van Rossum (ECRC)
Joachim Schimpf (ECRC and IC-Parc)
Kish Shen (IC-Parc)
Periklis Andreas Tsahageas (ECRC)
Dominique Henry de Villeneuve (ECRC)

August 6, 1999

Trademarks

Contents

Chapter 1

Introduction

1.1 What is ECL^{i}PS^e ?

ECL^{i}PS^e (ECL^{i}PS^e Common Logic Programming System) is a Prolog based system whose aim is to serve as a platform for integrating various Logic Programming extensions, in particular Constraint Logic Programming (CLP). The kernel of ECL^{i}PS^e is an efficient implementation of standard (Edinburgh-like) Prolog as described in basic Prolog texts [2]. It is built around an incremental compiler which compiles the ECL^{i}PS^e source into WAM-like code [14], and an emulator of this abstract code.

1.2 Overview

The ECL^{i}PS^e logic programming system is (so far) an integration of ECRC's SEPIA, MegaLog and (parts of the) CHIP system and newly developed libraries. This combination is now the default configuration of the system. The documentation is organised as follows:

The User Manual describes the functionality of the ECL^{i}PS^e kernel (this document).

The Library Manual describes the major ECL^{i}PS^e libraries, in particular the ones implementing constraint solvers.

The Interfacing and Embedding Manual describes how to interface ECL^{i}PS^e to other programming languages, and in particular how to embed it into an application as a component.

Apart from these manuals, there are detailed descriptions of all built-in and most library predicates. They can be obtained either from the development system via the help/1 command, or with an html browser (refer to the eclipse installation directory under doc/index.html).

1.3 Further Information

ECL^{i}PS^e has initially been developed at the European Computer-Industry Research Centre (ECRC) in Munich, and is now being further developed and maintained at IC-Parc (Centre for Planning and Resource Control at the Imperial College in London) with the support of ICL and the CHIC-2 ESPRIT project. Up-to-date information, including ordering information can be obtained from the ECL^{i}PS^e web site `http://www.icparc.ic.ac.uk/eclipse`

or by sending email to

`eclipse-request@icparc.ic.ac.uk`

There is also an ECL^{i}PS^e user group mailing list. Contributions to this list can be sent to

`eclipse-users@icparc.ic.ac.uk`

and requests for being added to or removed from this list to `majordomo@icparc.ic.ac.uk`.

1.4 Reporting Problems

In order to make ECL^{i}PS^e as useful and reliable as possible, we would like to encourage users to send problem reports by e-mail to

`eclipse-bugs@icparc.ic.ac.uk`

A bug report form can be found in the `doc` subdirectory of your ECL^{i}PS^e installation.

Chapter 2

Terminology

This chapter defines the terminology which is used throughout the manual and in related documentation.

+X This denotes an input argument. Such an argument must be instantiated before a built-in is called.

−X This denotes an output argument. Such an argument must be not instantiated before a built-in is called.

?X This denotes an input or an output argument. Such an argument may be either instantiated or not when a built-in is called.

Arity Arity is the number of arguments to a term. Atoms are considered as functors with zero arity. The notation *Name/Arity* is used to specify a functor of name *Name* with arity *Arity*.

Atom An arbitrary name chosen by the user to represent objects from the problem domain. A Prolog *atom* corresponds to an identifier in other languages.

Atomic An atom, string or a number. A terms which does not contain other terms.

Body A clause *body* can either be of the form

```
Goal_1, Goal_2, ..., Goal_k
```

or simply

```
Goal
```

Each *Goal_i* must be a callable term.

Built-in Procedures These are predicates provided for the user by the ECL^{i}PS^e system, they are either written in Prolog or in the implementation language (usually “C”).

Clause See program clause or goal.

Callable Term A *callable term* is either a compound term or an atom.

Compound Term Compound terms are of the form

```
f(t_1, t_2, ..., t_n)
```

where *f* is the *functor* of the compound term and *t_i* are terms, n is its arity. Lists and Pairs are also compound terms.

DID Each atom created within ECL^{i}PS^e is assigned a unique identifier called the *dictionary identifier* or *DID*.

Difference List A difference list is a special kind of a list. Instead of being ended by *nil*, a difference list has an uninstantiated tail so that new elements can be appended to it in constant time. A difference list is written as *List - Tail* where *List* is the beginning of the list and *Tail* is its uninstantiated tail. Programs that use difference lists are usually more efficient and always much less readable than programs without them.

Dynamic Procedures These are procedures which can be modified clause-wise, by adding or removing one clause at a time. Note that this class of procedure is equivalent to interpreted procedures in other Prolog systems. See also *static procedures*.

ElemSpec An *ElemSpec* specifies a global variable (an atom) or an array element (a ground compound term with as much arguments (integers) as the number of dimensions of the array).

External Procedures These are procedures which are defined in a language other than Prolog, and explicitly connected to Prolog predicates by the user.

Fact A fact or *unit clause* is a term of the form:

```
Head.
```

where *Head* is a structure or an atom. A fact may be considered to be a rule whose body is always *true*.

Functor A functor is characterised by its name which is an atom, and its arity which is its number of arguments.

Goal Clause See *query*.

Ground A term is ground when it does not contain any uninstantiated variables.

Head A head is a structure or an atom.

Instantiated A variable is instantiated when it has been bound to an atomic or a compound term as opposed to being *uninstantiated* or *free*. See also *ground*.

List A list is a special type of term within Prolog. It is a recursive data structure consisting of *pairs* (whose tails are lists). A `list` is either the atom `[]` called `nil` as in LISP, or a pair whose tail is a list. The notation :

```
[a , b , c]
```

is shorthand for:

```
[a | [b | [c | []]]]
```

Name/Arity The notation `Name/Arity` is used to specify a functor of name **Name** with arity **Arity**.

Pair A pair is a compound term with the functor `./2` (`dot`) which is written as :

```
[H|T]
```

`H` is the `head` of the pair and `T` its `tail`.

Predicate A predicate is another term for a *procedure*.

PredSpec This is similar to the notation `Name/Arity`. Some built-ins allow the arity to be omitted and to specify `Name` only. This stands for all (visible) predicates with that name and any arity.

Program Clause A *program clause* or *clause* is either the term

```
Head :- Body.
```

i.e. a compound term with the functor *:-/2*, or only a fact.

Query A query has the same form as *Body* and is also called a *goal*. Such clauses occur mainly as input to the top level Prolog loop and in files being compiled, then they have the form

```
:- Goal_1, ..., Goal_k.
```

or

```
?- Goal_1, ..., Goal_k.
```

Regular Prolog Procedure A *regular (Prolog) procedure* is a sequence of user clauses whose heads have the same functor, which then identifies the user procedure.

Simple Procedures Apart from regular procedures ECL^{i}PS^e recognises *simple procedures* which are written not in Prolog but in the implementation language, i.e. C and which are deterministic. There is a functor associated with each simple procedure, so that any procedure recognisable by ECL^{i}PS^e is identified by a functor, or a compound term with this functor (or atom).

SpecList The SpecList notation means a sequence of terms of the form:

```
name_1/a_1, name_2/a_2, ..., name_k/a_k.
```

The SpecList notation is used in many built-ins, for example, to specify a list of procedures in the **global/1** predicate.

Static Procedures These are procedures which can only be changed as a whole unit, i.e. removed or replaced.

Stream This is an I/O channel identifier and can be a physical stream number, one of the reserved stream identifiers or a user defined stream name (defined using **set_stream/2** or **open/3**). The reserved stream identifiers are:

```
input, output, error, toplevel_input, toplevel_output,
answer_output, debug_input, debug_output, user, null,
stdin, stdout, stderr.
```

Structures Compound terms which are not pairs are also called *structures*.

Term A *term* is the basic data type in Prolog. It is either a *variable*, a *constant*, i.e. an *atom*, a *number* or a *string*, or a *compound term*.

The notation **Pred/N1, N2** is often used in this documentation as a shorthand for **Pred/N1, Pred/N2**.

Chapter 3

Getting started with ECL^{i}PS^e

3.1 Entering the ECL^{i}PS^e System

ECL^{i}PS^e can be installed in several configurations, according to the installation notes. Enter one of the following commands after the operating system prompt:

- **eclipse**: The basic variant of ECL^{i}PS^e with a a command-line based user interface, as described in this manual.
- **tkeclipse**: ECL^{i}PS^e with a Graphical Development Interface. This is overviewed in chapter 4, and comes with its own online help.
- **peclipse**: The parallel ECL^{i}PS^e variant as described in chapter 8.

When one of these commands is invoked, the ECL^{i}PS^e system will display the initial header:

```
% eclipse
ECLiPSe Constraint Logic Programming System [kernel]
Version X.Y.Z, Copyright IC-Parc and ICL, DAY MONTH DD HH:MM YYYY
[eclipse 1]:
```

The list in square brackets specifies the configuration of the running system, i.e. the language extensions that are present. This is followed by the prompt **[eclipse 1]:**, which tells the user that the top-level loop is waiting for a user query in the module **eclipse**. The predicate **help/0** gives a general help and **help/1** gives help about specific built-in predicates.

3.2 ECL^{i}PS^e Command Line Options

The following command line options may be specified

−b bootfile Compile the file *bootfile* before starting the session or before saving the state using the −s option.

−e goal Instead of starting an interactive toplevel-loop, the system will execute the goal **goal**. **goal** is given in normal Prolog syntax, and has to be quoted if it contains any characters that would normally be interpreted by the shell. The -e option can be used together with the -b option and is executed afterwards.

The exit status of the ECL^{i}PS^e process reflects success or failure of the executed Prolog goal (0 for success, 1 for failure).

−g **size** This option specifies to which limit the memory consumption of the ECL^{i}PS^e global/trail stack can grow. The size is specified in kilobytes, or in megabytes when the number is followed by the letter M. The default is 128M, ie. 128 Megabytes. (On machines that do not support memory mapping, the stacks are pre-allocated and the default size is only 750K).

−l **size** This option specifies to which limit the memory consumption of the ECL^{i}PS^e local/control stack can grow. The size is specified in kilobytes, or in megabytes when the number is followed by the letter M. The default is 128M, ie. 128 Megabytes. (On machines that do not support memory mapping, the stacks are pre-allocated and the default size is only 200K).

−**h size** This option specifies to which limit the memory consumption of the ECL^{i}PS^e private heap can grow. The size is specified in kilobytes, or in megabytes when the number is followed by the letter M. The default is 32M, ie. 32 Megabytes.

−s **size** This option specifies to which limit the memory consumption of the ECL^{i}PS^e shared heap can grow. The size is specified in kilobytes, or in megabytes when the number is followed by the letter M. The default is 64M, ie. 64 Megabytes.

−**p size** The size of the page buffer area for the database handling is set to *size* kbytes. The default value is 400 kbytes.

− − The ECL^{i}PS^e system will ignore this argument and everything that follows on the commmand line. The Prolog program will only see the part of the command line that follows this argument.

More command line options are described in chapter 8.

3.3 The `.eclipserc` file

Before displaying the prompt, the system checks whether there is a file called `.eclipserc` in the current directory and if not, in the user's home directory and if this file is found, ECL^{i}PS^e compiles it first. Thus it is possible to put various initialisation commands into this file. ECL^{i}PS^e has many possibilities to change its default behaviour and setting up a `.eclipserc` file is a convenient way to achieve this. A different name for the initialisation file can be specified in the environment variable ECLIPSEINIT. If ECLIPSEINIT is set to an empty string, no initialisation is done. If the system is started with a -e option, then the `.eclipserc` file is ignored.

3.4 Interaction with the Toplevel Loop

3.4.1 Entering Goals

The ECL^{i}PS^e prompt **[eclipse 1]:** indicates that ECL^{i}PS^e is at the top level and the opened module is **eclipse**. The *top level loop* is a Prolog procedure which repetitively prompts the user for a query, executes it and reports its result, i.e. either the answer variable bindings or the failure message. There is always exactly one module opened in the top level and its name is printed in the prompt. From this point it is possible to enter Prolog goals, e.g. to pose queries, to enter a Prolog program from the keyboard or to compile a program from a file. Goals are entered after the prompt and are terminated by fullstop and newline.

3.4.2 Exiting from the Toplevel

The ECL^{i}PS^e system may be exited by typing CTRL-D (UNIX) or CTRL-Z + RETURN (Windows) at the top level prompt, or by calling the predicate **halt/0** or **exit/1**.

3.4.3 Entering Programs from the Terminal

To enter Prolog code at the terminal, type **[user].** or **compile(user).** in response to the top level prompt. (The closing bracket must be followed by a "full stop" just like any other query.) The system then displays the compiler prompt (which is a blank by default) and waits for a sequence of Prolog clauses. Each of the clauses is terminated by a fullstop. (If the fullstop is omitted the system just sits waiting, because it supposes the clause is not terminated. If you omit the stop by accident simply type it in on the following line, and then proceed to type in the program clauses, each followed by a full stop and carriage return.) To return to the top level prompt, type CTRL-D (UNIX), CTRL-Z + RETURN (Windows) or enter the atom **end_of_file** followed by fullstop and RETURN.

```
[eclipse 1]: [user].
 father(abraham, isaac).
 father(isaac, jacob).
 father(jacob, joseph).
 ancestor(X, Y) :- father(X, Y).
 ancestor(X, Y) :- ancestor(X, Z), ancestor(Z, Y).
 ^D
  user       compiled traceable 516 bytes in 0.00 seconds

yes.
[eclipse 2]:
```

The two predicates father/2 and ancestor/2 are now compiled and can be used.

3.4.4 Querying Programs

Once a set of clauses has been compiled into the database, it may be queried in the usual Prolog manner. If there are no uninstantiated variables in the query, the system replies 'yes' or 'no' and prompts for another query, for example:

```
[eclipse 1]: father(jacob, joseph).
yes.
[eclipse 2]:
```

If there are uninstantiated variables in the query, the system will attempt to find an instantiation of them which will satisfy the query, and if successful it will display one such instantiation. It will then wait for a further instruction: either a `<CR>` ("newline" or "return") or a semi-colon ';'. A return will end the query successfully. A semi-colon will initiate backtracking in an attempt to find another solution to the query. Note that it is not necessary to type a new line after the semicolon — one keystroke is enough. When the top level loop can detect that there are no further solutions, it does not wait for the semicolon or newline, but it displays directly the next prompt. For example in a query on a family database:

```
[eclipse 2]: father(X, Y).
X = abraham
Y = isaac   More? (;)   (';' typed)

X = isaac
Y = jacob

yes.
[eclipse 3]:
```

Queries may be extended over more than one line. When this is done the prompt changes to a tabulation character, ie. the input is indented to indicate that the query is not yet completed. The fullstop marks the end of the input.

3.4.5 Syntax errors

If an error occurs while reading input from the terminal, the system prints an error message after the next newline and waits for input of a correct Prolog term.

```
[eclipse 3]: a b      <return>
syntax error: postfix/infix operator expected
| a b
|    ^ here
[eclipse 3]:
```

During compilation, clauses with syntax errors cause an error message and are ignored, but all other clauses are compiled normally.

3.4.6 Interrupting the execution

If a program is executing, it may be interrupted by typing **CTRL-C** (interrupt in the UNIX environment). This will invoke the corresponding interrupt handler (see section 14.3). By default, the system prints a menu offering some alternatives:

```
^C
interruption: type a, b, c, e, or h for help : ? help
        a : abort
        b : break level
        c : continue
        e : exit
        h : help

interruption: type a, b, c, e, or h for help : ?
```

The `a` option returns to the toplevel, `b` starts a nested toplevel, `c` continues the interrupted execution, `d` switches the debugger to creep mode provided it is running, and `e` is an emergency exit of the whole ECL^{i}PS^e session.
The execution of ECL^{i}PS^e may be suspended by typing **CTRL-Z** (suspend) or by calling **pause/0**. This will suspend the ECL^{i}PS^e process and return the UNIX prompt. Entering the BSD-UNIX C-shell command `fg` will return to ECL^{i}PS^e Note that this feature may not be available on all systems.

3.4.7 History Mechanism

The ECL^{i}PS^e toplevel loop provides a simple history mechanism which allows to examine and to repeat previous queries. The history list is printed with command **h**. A previous query is invoked by typing its absolute number or its relative negative offset from the current query number (i.e. –1 will execute the previous query). The current query number is displayed in the toplevel prompt.
The history is initialized from the file *.eclipse_history* in the current directory or in the home directory. This file contains the history goals, each ended by a fullstop. The current history can be written using the predicate **write_history/0** from the **util** library.

3.4.8 Getting Help

The contents of the ECL^{i}PS^e BIP book, i.e. the detailed descriptions of all built-in predicates, can be accessed with the help-facility. It has two modes of operation. First, when a fragment of a built-in name is specified, a list of short descriptions of all built-ins whose name contains the specified string is printed, .e.g.

```
:- help(write).
```

will print one-line descriptions about write/1, writeclause/2 etc. When a unique specification is given, the full description of the specified built-in is displayed, e.g. in

```
:- help(write/1).
```

3.4.9 Global Flags and Settings

ECL^{i}PS^e has a number of flags to control options and modes of operation. They will be described in detail in the appropriate places. To get an overview of the existing flags, call **env/0** which will print a list like

```
[eclipse 1]: env.
all_dynamic:            off             last_errno:             0
break_level:            0               macro_expansion:        on
coroutine:              off             max_global_trail:       134217728
debug_compile:          on              max_local_control:      134217728
debugger_model:         eclipse         max_predicate_arity:    255
debugging:              nodebug         object_suffix:          "so"
dfid_compile:           off             occur_check:            off
enable_interrupts:      on              output_mode:            "QPm"
float_precision:        double          pid:                    21660
gc:                     on              ppid:                   14029
gc_interval:            1048576         prefer_rationals:       off
gc_interval_dict:       960             print_depth:            20
goal_expansion:         on              toplevel_module:        eclipse
hostarch:               "sparc_sunos5"  unix_time:              919720465
hostid:                 "9999999999"    version:                '4.1'
hostname:               "breeze"        wm_window:              off
ignore_eof:             off             worker:                 0
```

```
cwd:                    "/homes/john/"
extension:              development occur_check dfid
installation_directory: "/usr/local/eclipse"
library_path:           ["/usr/local/eclipse/lib", ...]
loaded_library:         lists pdb idb tracer_tty environment array
                        development_support tracer setof suspend sorts io
prolog_suffix:          ["", ".sd", ".ecl", ".pl"]
variable_names:         on
workerids:              "breeze" : [0] + []
workers:                "breeze" : 1
```

The values of individual flags can be retrieved using **get_flag/2**. Some of these flags can be set using **set_flag/2**. The built-in **statistics/0** displays various figures about time and memory usage. Refer to chapter 19 for details.

3.5 More about compilation

3.5.1 Optimised Compilation

Note that the code above was compiled as **traceable**, which means that it can be traced using the built-in debugger or using the programmable debugger OPIUM. To obtain maximum efficiency, the directive **nodbgcomp** should be used, which will set some flags to produce a more efficient and shorter code

```
[eclipse 2]: nodbgcomp.

yes.
[eclipse 3]: [user].
 father(abraham, isaac).
 father(isaac, jacob).
 father(jacob, joseph).
 ancestor(X, Y) :- father(X, Y).
 ancestor(X, Y) :- ancestor(X, Z), ancestor(Z, Y).
  user      compiled optimized 396 bytes in 0.02 seconds

yes.
[eclipse 4]:
```

3.5.2 Compiling from a File

The square brackets **[...]** or **compile/1** are also used to compile Prolog source from a file. If the goal

```
compile(myfile).
```

or the short-hand notation:

```
[myfile].
```

is called, either as a query at the top level or within another goal, the system looks for the file **myfile** or for a file called **myfile.pl** and compiles it into the Prolog database. The short-hand notation may also be used to compile several files in sequence

```
[ file_1, file_2, ...   file_n ]
```

The **compile/2** predicate may be used to compile a file or list of files into a module specified in the second argument.
It is a recommended programming practice to give the Prolog source programs the suffix **.pl** or **.ecl** if it contains ECL^{i}PS^e specific code. It is not enforced by the system, but it simplifies managing the source programs. The **compile/1** predicate automatically adds the suffix to the filename, so that it does not need to be specified; only if the literal filename can not be found, the system appends one of the valid suffixes and tries to find the resulting filename. The system's list of valid Prolog suffixes is in the global flag **prolog_suffix** and can be examined and modified using **get_flag/2** and **set_flag/2**. For example, to add the new suffix ".pro" use:

```
get_flag(prolog_suffix, Old), set_flag(prolog_suffix, [".pro"|Old]).
```

3.5.3 File Queries and Directives

A file being compiled may contain queries. These are goals preceded by either the symbol "?-" or the symbol ":-". As soon as a query or command is encountered in the compilation of a file, the ECL^{i}PS^e system will try to satisfy it. In this way, in particular, a file can contain a directive to the system to compile another file, and so large programs can be split between files. When this happens, ECL^{i}PS^e interprets the pathnames of the nested compiled files relative to the directory of the parent compiled file; if e.g. the user calls

```
[eclipse 1]: compile('src/pl/prog').
```

and the file src/pl/prog.pl contains a query

```
:- [part1, part2].
```

then the system searches for the files `part1.pl` and `part2.pl` in the directory `src/pl` and not in the current directory. Usually larger Prolog programs have one main file which contains only commands to compile all the subfiles. In ECL^{i}PS^e it is possible to compile this main file from any directory, whereas in other Prolog systems it might be necessary to make the directory of the main file the current one, or to specify in the main file the full pathnames for all compiled subfiles.
If the **compile/1** predicate is called and the system is unable to find or open the required file, it will issue an error:

```
[eclipse 1]: [file].
File does not exist in compile('/user/lp/eclipse/src/file')
yes.
[eclipse 2]:
```

If in compilation of a file `file_a` the compiler is directed to open or find a file `file_b`, but cannot do so, it will raise an exception whose default action is to write an error message and continue to compile `file_a`.

3.5.4 Compiling Procedures as Dynamic or Static

If it is intended that a procedure be altered through the use of **assert/1** and **retract/1**, the system should be informed that the procedure will be dynamic, since these predicates are designed to work on dynamic procedures. If **assert/1** is applied on a non-existing procedure, an error is raised, however the default error handler for this error only declares the procedure as dynamic and then makes the assertion.

A procedure is by default static unless it has been specifically declared as dynamic. Clauses of static procedures must always be consecutive, they may not be separated in one or more source files or by the user from the top level. If the static procedure clauses are not consecutive, each of the consecutive parts is taken as a separate procedure which redefines the previous occurrence of that procedure, and so only the last one will remain. However, whenever the compiler encounters nonconsecutive clauses of a static procedure in one file, it raises an exception whose default handler prints a warning but it continues to compile the rest of the file.

If a procedure is to be dynamic the ECL^{i}PS^e system should be given a specific *dynamic declaration* A dynamic declaration takes the form

```
:- dynamic SpecList.
```

The predicate **is_dynamic/1** may be used to check if a procedure is dynamic:

```
is_dynamic(Name/Arity).
```

When the goal

```
compile(Somefile)
```

is executed and `Somefile` contains clauses for procedures that have already been defined in the Prolog database, those procedures are treated in one of two ways: If such a procedure is dynamic, its clauses compiled from `Somefile` are added to the database (just as would happen if they were asserted), and the existing clauses are not affected. For example, if the following clauses have already been compiled:

```
:- dynamic city/1.

city(london).
city(paris).
```

and the file `Somefile` contains the following Prolog code:

```
city(munich).
city(tokyo).
```

then compiling `Somefile` will cause adding the clauses for **city/1** to those already compiled, as **city/1** has been declared dynamic. Thus the query **city(X)** will give:

```
[eclipse 5]: city(X).
X = london    More? (;)

X = paris    More? (;)

X = munich    More? (;)
```

```
X = tokyo
yes.
```

If, however, the compiled procedure is static, the new clauses in `Somefile` replace the old procedure. Thus, if the following clauses have been compiled:

```
city(london).
city(paris).
```

and the file `Somefile` contains the following Prolog code:

```
city(munich).
city(tokyo).
```

when `Somefile` is compiled, then the procedure **city/1** is redefined. Thus the query `city(X)` will give:

```
[eclipse 5]: city(X).
X = munich    More? (;)

X = tokyo
yes.
```

When the **dynamic/1** declaration is used on a procedure that is already dynamic, which may happen for instance by recompiling a file with this declaration inside, the system raises the error 64, 'procedure already dynamic'. The default handler for this error, however, will only erase all existing clauses for the specified procedure, so that when such a file is recompiled several times during its debugging, the system behaves as expected, the existing clauses are always replaced. The handler for this error can of course be changed if required. If it is set to **true/0**, for instance, the **dynamic/1** declaration is be just silently accepted without erasing any clauses and without printing an error message.

3.5.5 Altering Programs

The Prolog database can be updated during the execution of a program. ECL^{i}PS^e allows the user to modify procedures dynamically by adding new clauses via **assert/1** and by removing some clauses via **retract/1**. These predicates operate on dynamic procedures; if it is required that the definition of a procedure be altered through assertion and retraction, the procedure should therefore first be declared dynamic (see the previous section). The effect of **assert/1** and **retract/1** on static procedures is explained below.
The effect of the goal

```
assert(ProcClause)
```

where `ProcClause`[1] is a clause of the procedure `Proc`, is as follows.

1. If `Proc` has not been previously defined, the assertion raises an exception, however the default handler for this exception just declares the given procedure silently as dynamic and executes the assertion.

---

[1] It should be remembered that because of the definition of the syntax of a term, to assert a procedure of the form p :- q,r it is necessary to enclose it in parentheses: `assert((p:-q,r))`.

2. If `Proc` is already defined as a dynamic procedure, the assertion adds *ProcClause* to the database after any clauses already existing for `Proc`.

3. If `Proc` is already defined as a static procedure, then the assertion raises an exception.

The goal

```
retract(Clause)
```

will unify `Clause` with a clause on the dynamic database and remove it. If `Clause` does not specify a dynamic procedure, an exception is raised.
ECL^{i}PS^e's dynamic database features the so-called *logical update semantics.* This means that any change in the database that occurs as a result of executing one of the builtins of the abolish, assert or retract family affects only those goals that start executing afterwards. For every call to a dynamic procedure, the procedure is virtually frozen at call time.

3.6 Using Libraries

A number of files containing library predicates are issued with the ECL^{i}PS^e system. These predicates provide utility functions for general use. They are usually installed in a ECL^{i}PS^e library directory (or directories). These predicates are either loaded automatically by ECL^{i}PS^e or may be loaded "by hand".
During the execution of an ECL^{i}PS^e program, the system may dynamically load files containing library predicates. When this happens, the user is informed by a compilation or loading message. It is possible to explicitly force this loading to occur by use of the **lib/1** or **use_module/1** predicates. E.g. to load the library called `lists`, use one of the following directives

```
:- lib(lists)
:- use_module(library(lists))
```

will load the library file unless it has been already loaded. The library file is found by searching the library path and by appending a suffix to the library name.
The search path used when loading libraries is specified by the global flag **library_path** using the **get_flag/2** and **set_flag/2** predicates. This flag contains a list of strings containing the pathnames of the directories to be searched when loading a library file. User libraries may be be added to the system simply by copying the desired file into the ECL^{i}PS^e library directory. Alternatively the **library_path** flag may be updated to point at a number of user specific directories. The following example illustrates how a directive may be added to a file to add a user-defined library in front of any existing system libraries.

```
?- get_flag(library_path,Path),
   set_flag(library_path, ["/home/myuser/mylibs" | Path]).
```

The UNIX environment variable ECLIPSELIBRARYPATH may also be used to specify the initial setting of the library path. The syntax is similar to the syntax of the UNIX PATH variable, i.e. a list of directory names separated by colons. The directories will be prepended to the standard library path in the given order.

3.7 Redefining Built In Predicates

Any ECL^{i}PS^e built-in predicate can be redefined (i.e. hidden by a local predicate of the same name, cf. 9.6.3) by the user. To remind the user of what is happening, a warning is given if the redefinition is done by just defining a new predicate of the same name. To avoid the warning and to clarify the meaning, an explicit **local/1** declaration should be provided for the builtin that is to be redefined.

Some builtins are classified as **protected** (see Appendix E). Anyway, this does not mean that they can not be redefined. They just require the explicit **local/1** declaration to appear in the source *before* any occurrence as a subgoal. The same is true for redefining existing predicates with predicates that use a different calling convention (this is signaled by the "inconsistent redefinition" error). Different calling convention are used for normal Prolog predicates, for C externals and for Prolog tool predicates.

Chapter 4

Tkeclipse Development Environment

Tkeclipse is a graphical user interface to ECL^{i}PS^e. It is an alternative to the traditional textual line-based user interface, providing multiple windows, menus and buttons for the user to interact with ECL^{i}PS^e. It consists of two major components:

- A graphical top-level.
- A suite of development tools for aiding the development of ECL^{i}PS^e code.

Tkeclipse is implemented in the Tcl/Tk scripting language/graphical toolkit [13], using the new ECL^{i}PS^e Tcl/Tk interface [12]. The development tools are designed to be independent of the top-level, so the user can develop their own applications with a graphical front end written in Tcl/Tk, replacing the tkeclipse top-level, but still using the developments tools.

This chapter will provide an overview of tkeclipse, but will not describe its functionality in detail. More detailed information on the functionality of tkeclipse is available via tkeclipse's own on-line help.

4.1 Starting and obtaining help

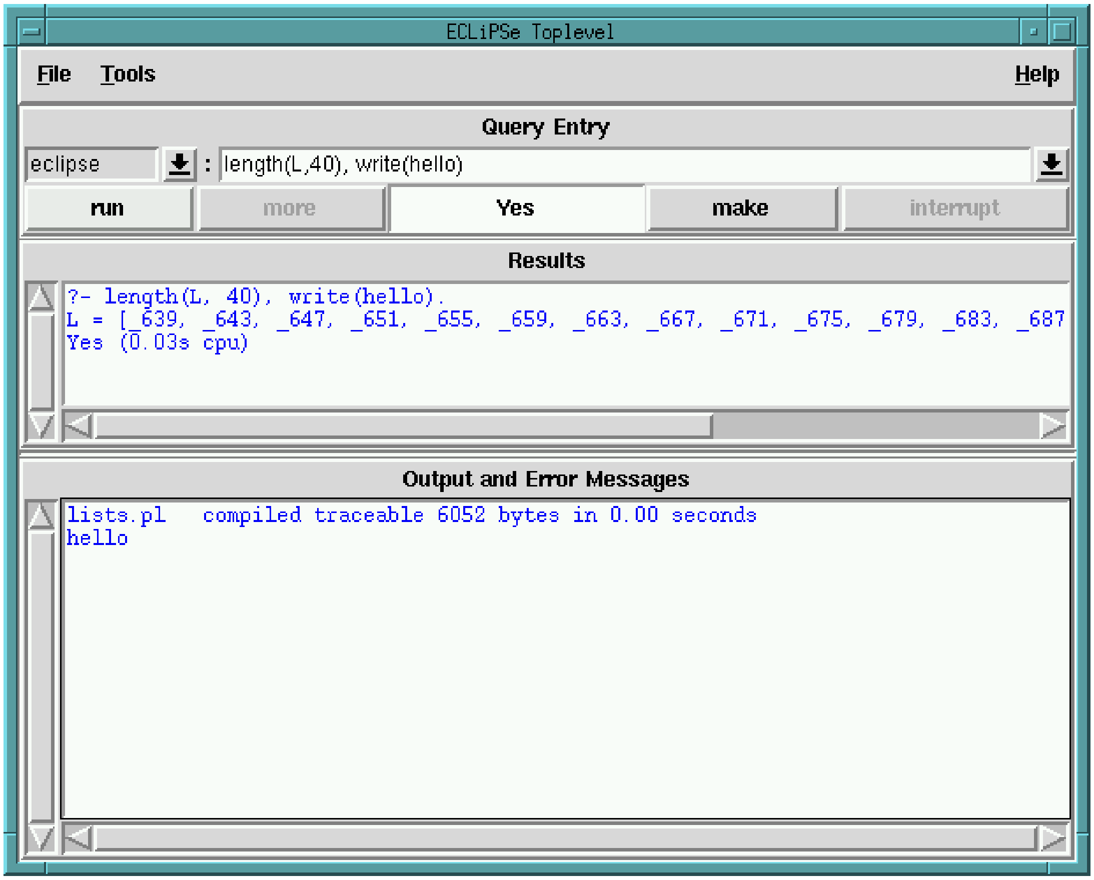
To start tkeclipse, type the command tkeclipse after the operating system prompt, or click on the tkeclipse icon in a window/icons based operating system. This will bring up the tkeclipse top-level, which is shown in Figure 4.1.

Help for tkeclipse can be obtained from the Help menu – the user can obtain on-line documentation on the development tools from this menu, along with turning the balloon help mode on. When this mode is on, a pop-up text 'balloon' will appear when the cursor is left on an item for a short while. The balloon will provide a brief explanation of the particular item, and it will disappear when the cursor is moved off the item.

Help on a particular development tool can also be obtained when that tool is being used. The on-line documentation for a tool can be obtained by typing Alt-H on the windows associated with the tool In addition, if a menu bar is available for the tool, the documentation can be obtained via the help menu.

4.2 Using tkeclipse

The user can interact with ECL^{i}PS^e by entering a query at the Query entry window at the top-level, or through the menus and buttons in the window. The query entry window acts similar to



```
eclipse : length(L,40), write(hello)
```

```
?- length(L, 40), write(hello).
L = [_639, _643, _647, _651, _655, _659, _663, _667, _671, _675, _679, _683, _687
Yes (0.03s cpu)
```

```
lists.pl    compiled traceable 6052 bytes in 0.00 seconds
hello
```

Figure 4.1: Tkeclipse top-level

the command line at the prompt of the tty interface, with the addition of a history mechanism. Another difference from the tty interface is that output streams are sent to different windows – the results of a query (top-level variable bindings, state after executing query, and time taken to execute query) is presented in the Results window, and the output to standard output and error are sent to the Output and Error Messages window. Outputs to the `debug_output` stream is sent to the tracer's trace log window. Reading from standard input will cause a window to pop-up to read input from the user. Figure 4.1 illustrates this division of the outputs – the bindings to the top-level variable `L` is shown in the Results window, whereas the `write(hello)` to standard output is shown in the Output and Error Messages window.
All built-ins which are available in ECL^{i}PS^e are available under tkeclipse. Some commands from the tty interface are not present in tkeclipse: `trace/0` and `debug/0` which invoke the debugger; and `[user]`, which allow the user to type in simple ECL^{i}PS^e code. The equivalent functionality is provided via the tkeclipse tracer and compile scratch-pad tools respectively.
The Tools menu allow the user to launch the tools of the development tools suite. The following tools are available from the menu:

- Compile scratch-pad
- Source file manager

- Predicate browser
- Source viewer
- Delay goals
- Tracer
- Inspector
- Global settings
- Statistics
- Simple Query
- ECLiPSe help

Note that one tool, the **display matrix** tool, is not available from the menu. This tool is be invoked from ECL^{i}PS^e code, and is described in more detail in section 4.2.1.

The File menu provides some common file related operations such as compile, edit and make, as well as exiting from tkeclipse. Note that the file browser defaults to the `.ecl` extension, so that only files with such an extension are shown in the browser initially.

If the name of a menu button on the menu-bar has an underlined character (as in File), then the pressing Alt with the underlined letter (either upper or lower case) will popup the associated menu without using the mouse. The arrow keys can be used to navigate the menu, and return to select.

4.2.1 The Tkeclipse tool suite

The tkeclipse tools will not be covered in detail here – they are best tried by hands-on experimentation with the aid of the on-line help.

Compile scratch-pad

This tool replaces the `[user]` facility of the tty interface, and allows the user to type in short program code and compile it. The code is sent to ECL^{i}PS^e as a string, and ECL^{i}PS^e will respond that 'string' has been compiled. Note that the window's content is forgotten when the window is closed – larger programs should be written using a text editor, source files and the source file manager.

Source File Manager

This tool provides an interface to the *make* facility of ECL^{i}PS^e – it displays the information used by make to track file statuses. All files that have been compiled by ECL^{i}PS^e in the current session would be listed, and the user can also use this to select files that needs editing or be compiled individually. Files which have not yet been compiled can be added to this list, but they must be compiled first before make will recompile them.

Predicate Browser

This tool allows the user to browse through the defined modules and predicates of the current session, showing the user the properties associated with the selected predicate. The modifiable properties can be changed.

Source Viewer

This tool tries to display the source of a selected predicate. A predicate is selected in other tools, which will launch the source viewer window. The tool itself does not provide a selection mechanism. Note that the tool may be unable to display the source, because it may not be accessible to the user, and in addition, the tool uses a simple algorithm to try and find the predicate, so it may be unable to locate the predicate even if it is accessible.

Delayed Goals

This tool displays the currently delayed goals. It is possible to filter out uninteresting goals, e.g. by displaying only goals that are traceable or have a spy point set.

The user can select specific goals in the window and perform operations on the goals such as viewing its source, inspecting it, and putting a spy-point on the goal.

Tracer

This tool is the debugger for tkeclipse, and replaces the tty-based tracer in functionality. One difference from the tty tracer is the display of the Call Stack, which shows the ancestors of the current goal. The user can select goals in this stack to perform operations such as viewing the source and inspecting them. The trace log window shows output similar to the output from the traditional tracer. Some of the most common debugger commands are available as buttons, and their keystroke equivalent can also be used to invoke the commands. The 'Continue Until' option provides a more sophisticated means of controlling which port the debugger should stop at.

Note that the tracer can be invoked to start tracing at a particular predicate. This is done by turning the 'start_tracing' and 'spy' predicate properties on for the predicate (this can be done from the predicate browser).

Inspector

This tool provides a graphical browser for inspecting terms. Goals and data terms are displayed as a tree structure. Sub-trees can be collapsed and expanded by double-clicking. A navigation panel can be launched which provides arrow buttos as an alternative way to navigate the tree.

The Inspector tool can be invoked from within other tools such as the tracer and the delayed goals viewer, but it can also be invoked on its own, in which case the term being inspected is the current goal. Note that when the Inspector is active, interactions with the other tkeclipse windows are disallowed. This prevents the term from changing while being inspected. To continue tkeclipse, the inspector window must be closed.

Global Settings

This tool shows the settings of some global flags, which can be accessed via the `set_flag/2` and `get_flag/2` predicates.

Statistics

The tool displays some of the statistics on the current memory usage and timings, information which can also be obtained using `statistics/0,2`. However, the information is displayed in a graphical form, and is also updated automatically at regular intervals, allowing the user to monitor the changing statistics as a program is executing.

Simple Query

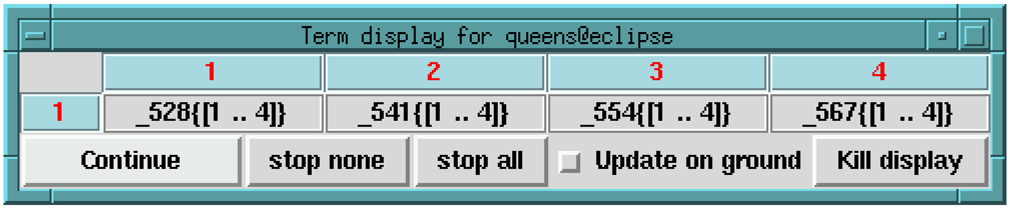
This tool allows the user to send a simple query to ECL^{i}PS^e even while ECL^{i}PS^e is running some program and the Toplevel Query Entry window is unavailable. Note that the reply is shown in EXDR format (see the ECL^{i}PS^e Embedding and Interfacing Manual).

ECL^{i}PS^e Help

This tool provides an interface to the `help/1` facility of ECL^{i}PS^e. A simple form of 'hypertext' facility is provided in that the user can double click on any word in the window to select the word in the entry window.

Display Matrix

This tool provides a method to display the values of terms in a matrix form. It is particularly useful because it can display the attributes of an attributed variable[1]. The tool is invoked from ECL^{i}PS^e code with just one predicate. This predicate is considered a no-op in the tty based ECL^{i}PS^e, and so the same code can be run without modification in either environment.



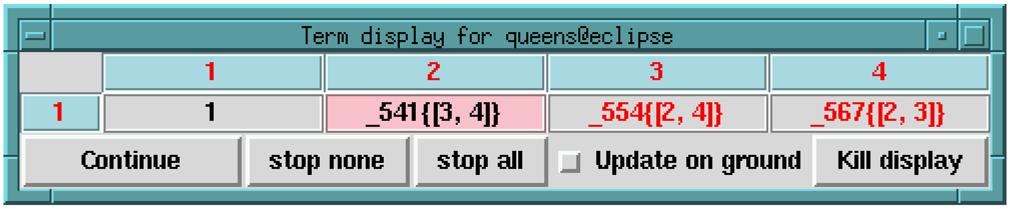
	1	2	3	4
1	_528{[1 .. 4]}	_541{[1 .. 4]}	_554{[1 .. 4]}	_567{[1 .. 4]}

Figure 4.2: Display Matrix Tool for 4-Queens (Initial)

This tool must be invoked from ECL^{i}PS^e code, using the `make_display_matrix/2,5` predicates. Only this one predicate needs to be added, and no other changes need to be made to the code. For example, in the following fragment of a N-queens program, only one extra line has been added to invoke a display matrix:

---

[1] The display matrix tools is similar to the variable display of **Grace**. The main differences are: it can display all attributes, not just the finite domain attribute; it only allows observation of the attributes, but cannot change the attribute or the labelling strategy



	1	2	3	4
1	1	_541{[3, 4]}	_554{[2, 4]}	_567{[2, 3]}

Figure 4.3: Display Matrix Tool for 4-Queens (During execution)

```
queens(N, List) :-
    length(List, N),
    List :: 1..N,
    make_display_matrix(List/0, queens),
    % sets up a matrix with all variables in 1 row. This is the only
    % extra goal that has to be added to enable monitoring
    alldistinct(List),
    constrain_queens(List),
    labeling(List).
```

Figures 4.2 and 4.3 show the tool invoked with the example N-Queens programs for 4 Queens, at the start initially and during the execution of the program. The name of the display window is specified by the second argument of `make_display_matrix/2`, along with the module it is in. The values of the terms are shown in the matrix, which can be one dimensional (as in this case), or two dimensional. Break-points can be set on each individual cell of the matrix so that execution will stop when the cell is updated. The matrix can be killed using the 'Kill display' button. Left-clicking on a cell will bring up a menu which shows the current and previous value of the term in the cell (the current value is shown because the space available in the cell may be too small to fully display the term), and allow the user to inspect the term using the inspector. Note the display matrix can be used independently of, or in conjunction with, the tracer. Multiple display matrices can be created to view different terms.
The following predicates are available in conjunction with the display matrix:

make_display_matrix(+Terms, +Name)
make_display_matrix(+Terms, +Prio, +Type, +CondList, +Name) These predicates create a display matrix of terms that can be monitored under tkeclipse. The two argument form is a simplification of the five argument form, with defaults settings for the extra arguments. Terms is a list or array of terms to be displayed. A List can be specified in the form List/N, where N is the number of elements per row of the matrix. If N is 0, then the list will be displayed in one row (it could also be omitted in this case). The extra arguments are used to control how the display is updated.
The terms are monitored by placing a demon suspension on the variables in each term. When a demon wakes, the new value of the term it is associated with is sent to the display matrix (and possibly updated, depending on the interactive settings on the matrix). When the new value is backtracked, the old value is sent to the display matrix. The other arguments in this predicate is used to control when the demon wakes, and what sort of information is monitored. Prio is the priority that the demon should be suspended at, Type is designed to specify the

attributes that is being monitored (currently all attributes are monitored, and Type is a dummy argument), CondList is the suspension list that the demon should be added to. Depending on these arguments, the level of monitoring can be controlled. Note that it is possible for the display matrix to show values that are out of date because the change was not monitored.

The display matrix will be removed on backtracking. However, it will not be removed if make_display_matrix has been cut – `kill_display_matrix/1` can be used to explicitly remove the matrix in this case.

kill_display_matrix(+Name) This predicate destroys an existing display matrix. Name is an atomic term which identifies the matrix.

Destroys an existing display matrix. The display matrix is removed from being displayed, and from ECL^{i}PS^e so that the name can be reused. This is the only way to remove a display matrix if it is not removed normally when the original make_display_matrix used to create the matrix was unable to remove the matrix on backtracking because of cuts. It can also be used to remove a display matrix at any other time, but there may be less need for this.

The Name can be specified as `Name@Module`, where Name was the original name given to the display matrix, and Module the module in which the display matrix was created in (this is the format that the name appears in the title bar of the display matrix). This allows the display matrix to be killed from any module.

4.3 Using the Development tools in applications

The user can develop their own ECL^{i}PS^e application which uses the Tcl/Tk interface to provide a graphical front end. The development tool suite was designed to be independent of the tkeclipse top-level so that they can be used in a user's application. In effect, the user can replace the tkeclipse top-level with their own alternative top-level. Two simple examples in which this is done is provided in the `lib_tcl` library as `example.tcl` and `example1.tcl`. In addition, `tkeclipse` itself, in the file `tkeclipse.pl`, can be seen as a more complex example usage of the interface.

In order to use the Tcl/Tk interface, the system must be initialised as described in the Embedding manual. In addition, the user's Tcl code should probably also be provided as a package using Tcl's package facility, in order to allow the program to run in a different directory. See the Embedding manual and the example programs for more details on the initialisation needed.

The user should most likely provide a connection for the output stream of ECL^{i}PS^e so that output from ECL^{i}PS^e will go somewhere in the GUI. In addition, especially during the development, it is also useful to connect the error stream to some window so that errors (such as ECL^{i}PS^e compilation errors) are seen by the user. This can be done using the `ec_queue_connect` Tcl command described in the embedding manual.

Output from ECL^{i}PS^e need not be sent to a Tk window directly. The Tcl/Tk code which receives the output can operate on it before displaying it. It is intended that all such graphical operations should be performed on the Tcl side, rather than having some primitives provided on the ECL^{i}PS^e side.

The user can also provide balloon-help to his/her own application. The balloon help package is part of the Megawidget developed by Jeffrey Hobbs and used in tkeclipse. In order to define a balloon help for a particular widget, the following Tcl code is needed:

```
balloonhelp <path> <text>
```

where `<path>` is the pathname of the widget, and `<text>` is the text that the user wants to display in the balloon.

Chapter 5

Porting Applications to ECL^{i}PS^e

The ECL^{i}PS^e system is to a large extent compatible with Prolog systems of the Edinburgh family, and one of the requirements during the development of ECL^{i}PS^e was to minimise the effort required to port programs written in other dialects to ECL^{i}PS^e. However, there are some differences. When you want to run an existing Prolog application on the ECL^{i}PS^e system, you have basically two choices: Using a compatibility package, or modifying your program.

5.1 Using the compatibility packages

The ECL^{i}PS^e compatibility packages are the fastest way to get a program running that was originally written for a different system. The packages contain the necessary code to make ECL^{i}PS^e emulate the behaviour of the other system to a large extent. Compatibility packages exist for:

- ISO Standard Prolog, use use_module(library(iso)) (cf. appendix A.5)
- C-Prolog, use use_module(library(cprolog)) (cf. appendix A.6)
- Quintus Prolog, use use_module(library(quintus)) (cf. appendix A.11)
- SICStus Prolog, use use_module(library(sicstus)) (cf. appendix A.13)

Note that every package makes use of the preceding ones. To run SICStus applications, it is often enough to use the quintus mode. The source code of the compatibility packages is provided in the ECL^{i}PS^e library directory. Using this as a guideline, it should be easy to write similar packages for other systems, as long as their syntax does not deviate too much from the Edinburgh tradition.

The following problems can occur despite the use of compatibility packages:

5.1.1 Compiler versus Interpreter

If your program was written for an interpreter, e.g. C-Prolog, you have to be aware that ECL^{i}PS^e is a compiling system. There is a distinction between *static* and *dynamic* predicates. By default, a predicate is static. This means that its clauses have to be be compiled as a whole (they must not be spread over multiple files), its source code is not stored in the system, and it can not be modified (only recompiled as a whole). In contrast, a dynamic predicate may be modified by compiling or asserting new clauses and by retracting clauses. Its source code can be accessed using **clause/1,2** or **listing/0,1** A predicate is dynamic when it is explicitly declared as such or when it was created using **assert/1**. Porting programs from an interpreter usually requires the addition of some **dynamic** declarations. In the worst case, when (almost) all procedures have to be dynamic, the flag **all_dynamic** can be set instead.

5.1.2 Name clashes with global ECL^{i}PS^e builtins

Suppose you want to define a predicate named date/1, which conflicts with the ECL^{i}PS^e builtin called **date/1**. In this case the compiler will produce one of the following messages

```
warning: redefining a system predicate in date / 1
*** trying to redefine a procedure with another type: date / 1
```

depending on whether the definition or a call to the predicate was encountered first by the compiler. Both can be avoided by declaring the predicate as **local**. This declaration must be given before the first call to the predicate:

```
:- local date/1.
p(Y) :- date(Y).
date(1999).
```

Note that the **date/1** builtin is now hidden by your own definition, but only in the module where you have redefined it[1]. In all other modules, the builtin is still visible.
The same holds for the redefinition of protected predicates, see also section 3.7 and appendix E.

5.2 Porting Programs to plain ECL^{i}PS^e

If you want to use ECL^{i}PS^e to do further development of your application, it is probably advantageous to modify it such that it runs under plain ECL^{i}PS^e. In the following we summarise the main aspects that have to be considered when doing so.

- In general, it is almost always possible to add to your program a small routine that fixes the problem, rather than to modify the source of the application in many places. E.g. name clashes are easier fixed by using the **local/1** declaration rather than to rename the clashing predicate in the whole application program.

- Due to lack of standardisation, some subtle differences in the syntax exist between Prolog systems. See B.4 for details. ECL^{i}PS^e has a number of options that make it possible to configure its behaviour as desired.

- ECL^{i}PS^e has the `string` data type which is not present in Prolog of the Edinburgh family. Double-quoted items are parsed as strings in ECL^{i}PS^e, while they are lists of integers in other systems and when the compatibility packages are used (cf. chapter 6.4).

- I/O predicates of the **see** and **tell** group are not builtins in ECL^{i}PS^e, but they are provided in the **cio** library. Call `lib(cio)` in order to have them available (cf. appendix A). Similarly for **numbervars/3**.

---

[1] in case you don't use modules this is the module `eclipse`

- In ECL^{i}PS^e, some builtins raise events in cases where they just fail in other systems, e.g. `arg(1,2,X)` fails in C-Prolog, but raises a type error in ECL^{i}PS^e. If some code relies on such behaviour, it is best to modify it by adding an explicit check like

  ```
          ..., compound(T), arg(N, T, X), ...
  ```

  Another alternative is to redefine the arg/3 builtin, using call_explicit/2 to access the original version:

  ```
  :- local arg/3.
  arg(N, T, X) :-
          compound(X),
          call_explicit(arg(N, T, X), sepia_kernel).
  ```

  A third alternative, which is used in the compatibility packages, is to define an error handler which will fail the predicate whenever the event is raised. In this case:

  ```
  my_type_error(_, arg(_, _, _)) :- !, fail.
  my_type_error(E, Goal) :- error(default(E), Goal).
  :- set_error_handler(5, my_type_error/2).
  ```

- As the ECL^{i}PS^e compiler does not accept procedures whose clauses are not consecutive in a file, you have to load the library **scattered.pl** if you want to compile such procedures.

5.3 Exploiting ECL^{i}PS^e Features

When rewriting existing applications as well as when writing new programs, it is useful to bear in mind important ECL^{i}PS^e features which can make programs easier to write and/or faster:

- The maximum performance is obtained when calling **nodbgcomp/0** at the beginning of the session, before compiling any program and loading any libraries.

- ECL^{i}PS^e arrays and global variables (**setval/2, getval/2**) are usually more suitable to store permanent data than **assert/1** is, and are usually faster.

- ECL^{i}PS^e has a number of language extensions which make programming easier, see chapter 6.

- The predicates **get_flag/2**, **get_flag/3**, **get_file_info/3**, **get_stream_info/3**, **get_var_info/3** give a lot of useful information about the system and the data.

- The ECL^{i}PS^e macros often help to solve syntactic problems (see chapter 13).

- It is worth familiarising oneself with the debugger's features, see chapter 15.

- ECL^{i}PS^e is highly customizable, even problems which seemingly require modification of the ECL^{i}PS^e sources can very often be solved at the Prolog level.

Chapter 6

ECL^{i}PS^e-specific Language Features

ECL^{i}PS^e is a logic programming language derived from Prolog. This chapter describes ECL^{i}PS^e-specific language constructs that have been introduced to overcome some of the main deficiencies of Prolog.

6.1 Structure Notation

ECL^{i}PS^e structure notation provides a way to use structures with field names. It is intended to make programs more readable and easier to modify, without compromising efficiency (it is implemented by macro expansion).
A structure is declared by specifying a template like this

```
:- local struct( book(author, title, year, publisher) ).
```

Structures with the functor book/4 can then be written as

```
book with []
book with title:'tom sawyer'
book with [title:'tom sawyer', year:1886, author:twain]
```

which translate to the corresponding forms

```
book(_, _, _, _)
book(_, 'tom sawyer', _, _)
book(twain, 'tom sawyer', 1886, _)
```

This transformation is done by macro expansion, therefore it can be used in any context and is as efficient as using the structures directly.
The argument index of a field in a structure can be obtained using a term of the form

```
FieldName of StructName
```

E.g. to access (ie. unify) a single argument of a structure, use arg/3 like this:

```
arg(year of book, B, Y)
```

which is translated into

```
arg(3, B, Y)
```

When structures are printed, they are not translated back into the with-syntax by default. The reason this is not done is that this can be bulky if all fields are printed, and often it is desirable to hide some of the fields anyway.

A good way to control printing of big structures is to write special purpose write-transformations for them, for instance

```
:- functor(book with [],N,A), define_macro(N/A, tr_book_out/2, [write]).
tr_book_out(book with [author:A,title:T],
        no_macro_expansion(book with [author:A,title:T])).
```

which will cause book/4 structures to be printed like

```
book with [author:twain, title:tom sawyer]
```

while the other two arguments remain hidden.

6.1.1 Inheritance

Structures can be declared to contain other structures, in which case they inherit the base structure's field names. Consider the following declarations:

```
:- local struct(person(name,address,age)).
:- local struct(employee(p:person,salary)).
```

The `employee` structure contains a field `p` which is a `person` structure. Field names of the `person` structure can now be used as if they were field names of the `employee` structure:

```
[eclipse 1]: Emp = employee with [name:john,salary:2000].
Emp = employee(person(john, _105, _106), 2000)
yes.
```

Note that, as long as the `with` and `of` syntax is used, the `employee` structure can be viewed either as nested or as flat, depending on what is more convenient in a given situation. In particular, the embedded structure can still be accessed as a whole:

```
[eclipse 1]:
        Emp = employee with [name:john,age:30,salary:2000,address:here],
        arg(name of employee, Emp, Name),
        arg(age of employee, Emp, Age),
        arg(salary of employee, Emp, Salary),
        arg(address of employee, Emp, Address),
        arg(p of employee, Emp, Person).

Emp = employee(person(john, here, 30), 2000)
Name = john
Age = 30
Salary = 2000
Address = here
Person = person(john, here, 30)
yes.
```

Implementation note: The indices of nested structures expand into lists of integers rather than simple integers, e.g. `age of employee` expands into `[1,3]`.

6.1.2 Visibility

Structure declaration can be local to a module (when declared as above) or exported when declared as

```
:- export struct(...).
```

in the module interface part, or even global when declared as

```
:- global struct(...).
```

6.2 Loop/Iterator Constructs

Many types of simple iterations are inconvenient to write in the form of recursive predicates. ECL^{i}PS^e therefore provides a logical iteration construct **do/2**, which can be understood either by itself or by its translation to an equivalent recursion.
A simple example is the traversal of a list

```
main :-
        write_list([1,2,3]).

    write_list([]).
    write_list([X|Xs]) :-
        writeln(X),
        write_list(Xs).
```

which can be written as follows without the need for an auxliliary predicate:

```
main :-
        ( foreach(X, [1,2,3]) do
            writeln(X)
        ).
```

The general form of a do-loop is

(IterationSpecs **do** Goals)

and it corresponds to a call to an auxiliary recursive predicate of the form

```
do__n(...).
do__n(...) :- Goals, do__n(...).
```

IterationSpecs is one (or a comma-separated sequence) of the following:

fromto(First,In,Out,Last)
iterate Goals starting with In=First until Out=Last. In and Out are local variables in Goals.

foreach(X,List)
iterate Goals with X ranging over all elements of List. X is a local variable in Goals. Can also be used for constructing a list.

foreacharg(X,Struct)
iterate Goals with X ranging over all elements of Struct. X is a local variable in Goals. Cannot be used for constructing a term.

for(I,MinExpr,MaxExpr)
iterate Goals with I ranging over integers from MinExpr to MaxExpr. I is a local variable in Goals. MinExpr and MaxExpr can be arithmetic expressions. Can be used only for controlling iteration, ie. MaxExpr cannot be uninstantiated.

for(I,MinExpr,MaxExpr,Increment)
same as before, but Increment can be specified (it defaults to 1).

count(I,Min,Max)
iterate Goals with I ranging over integers from Min up to Max. I is a local variable in Goals. Can be used for controlling iteration as well as counting, ie. Max can be a variable.

param(Var1,Var2,...)
for declaring variables in Goals global, ie shared with the context. CAUTION: By default, variables in Goals are local!

Note that fromto/4 is the most general specifier, but foreach/2, foreacharg/2, count/3, for/3 and param/N are convenient shorthands.
The do-operator binds like the semicolon, ie. less than comma. That means that the whole do-construct should always be bracketed.
Unless you use :-pragma(noexpand) or :-dbgcomp, the do-construct is compiled into an efficient auxiliary predicate named do_nnn, where nnn is a unique integer.

6.2.1 Examples

Iterate over list

```
foreach(X,[1,2,3]) do writeln(X).
```

Maplist (construct a new list from an existing list)

```
(foreach(X,[1,2,3]), foreach(Y,List) do Y is X+3).
```

Sumlist

```
(foreach(X,[1,2,3]), fromto(0,In,Out,Sum) do Out is In+X).
```

Reverse list

```
(foreach(X,[1,2,3]), fromto([],In,Out,   Rev) do Out=[X|In]). % or:
(foreach(X,[1,2,3]), fromto([],In,[X|In],Rev) do true).
```

Iterate over integers from 1 up to 5

```
for(I,1,5) do writeln(I). % or:
count(I,1,5) do writeln(I).
```

Make list of integers [1,2,3,4,5]

```
(for(I,1,5), foreach(I,List) do true). % or:
(count(I,1,5), foreach(I,List) do true).
```

Make a list of length 3

```
(foreach(_,List), for(_,1,3) do true). % or:
(foreach(_,List), count(_,1,3) do true).
```

Get the length of a list

```
(foreach(_,[a,b,c]), count(_,1,N) do true).
```

Actually, the length/2 builtin is (almost)

```
length(List, N) :- (foreach(_,List), count(_,1,N) do true).
```

Filter list elements

```
(foreach(X,[5,3,8,1,4,6]), fromto(List,Out,In,[]) do
    X>3 -> Out=[X|In] ; Out=In).
```

Iterate over structure arguments

```
(foreacharg(X,s(a,b,c,d,e)) do writeln(X)).
```

Collect args in list (bad example, use =.. if you really want to do that!)

```
(foreacharg(X,s(a,b,c,d,e)), foreach(X,List) do true).
```

Collect args reverse

```
(foreacharg(X,s(a,b,c,d,e)), fromto([],In,[X|In],List) do true).
```

or like this:

```
S = s(a,b,c,d,e), functor(S, _, N),
(for(I,N,1), foreach(A,List), param(S) do arg(I,S,A)).
```

The following two are equivalent

```
foreach(X,[1,2,3])        do               writeln(X).
fromto([1,2,3],In,Out,[]) do In=[X|Out], writeln(X).
```

The following two are equivalent

```
count(I,1,5)     do            writeln(I).
fromto(0,I0,I,5) do I is I0+1, writeln(I).
```

Two examples for nested loops. Print all pairs of list elements:

```
Xs = [1,2,3,4],
( foreach(X, Xs), param(Xs) do
    ( foreach(Y,Xs), param(X) do
        writeln(X-Y)
    )
).
```

and the same without symmetries:

```
Xs = [1,2,3,4],
( fromto(Xs, [X|Xs1], Xs1, []) do
    ( foreach(Y,Xs1), param(X) do
        writeln(X-Y)
    )
).
```

6.3 Array Notation

Since our language has no type declarations, there is really no difference between a structure and an array. In fact, a structure can always be used as an array, creating it with **functor/3** and accessing elements with **arg/3**. However, this can look clumsy, especially in arithmetic expressions.

ECL^{i}PS^e therefore provides array syntax which enables the programmer to write code like

```
[eclipse 1]: Prime = a(2,3,5,7,11), X is Prime[2] + Prime[4].
X = 10
Prime = a(2, 3, 5, 7, 11)
yes.
```

Within expressions, array elements can be written as variable-indexlist or structure-indexlist sequences, e.g.

```
X[3] + M[3,4] + s(4,5,6)[3]
```

Indices run from 1 up to the arity of the array-structure. The number of dimensions is not limited.

To create multi-dimensional arrays conveniently, the built-in **dim/2** is provided (it can also be used backwards to access the array dimensions):

```
[eclipse]: dim(M,[3,4]), dim(M,D).
M = []([](_131, _132, _133, _134),
       [](_126, _127, _128, _129),
       [](_121, _122, _123, _124))
D = [3, 4]
yes.
```

Although **dim/2** creates all structures with the functor [ ], this has no significance other than reminding the programmer that these structures are intended to represent arrays.

Array notation is especially useful within loops. Here is the code for a matrix multiplication routine:

```
matmult(M1, M2, M3) :-
        dim(M1, [MaxIJ,MaxK]),
        dim(M2, [MaxK,MaxIJ]),
        dim(M3, [MaxIJ,MaxIJ]),
        (
            for(I,1,MaxIJ),
        param(M1,M2,M3,MaxIJ,MaxK)
    do
        (
            for(J,1,MaxIJ),
            param(M1,M2,M3,I,MaxK)
        do
            (
                for(K,1,MaxK),
                fromto(0,Sum0,Sum1,Sum),
                param(M1,M2,I,J)
            do
                Sum1 is Sum0 + M1[I,K] * M2[K,J]
            ),
            subscript(M3, [I,J], Sum)
        )
    ).
```

6.3.1 Implementation Note

Array syntax is implemented by parsing variable-list and structure-list sequences as terms with the functor subscript/2. For example:

```
X[3]          --->      subscript(X, [3])
M[3,4]        --->      subscript(M, [3,4])
s(4,5,6)[3]   --->      subscript(s(4,5,6), [3])
```

If such a term is then used within an arithmetic expression, a result argument is added and the built-in predicate **subscript/3** is called, which is a generalised form of **arg/3** and extracts the indicated array element.
When printed, subscript/2 terms are again printed in array notation, unless the print-option to suppress operator notation ("O") is used.

6.4 The String Data Type

In the Prolog community there have been ongoing discussions about the need to have a special string data type. The main argument against strings is that everything that can be done with strings can as well be done with atoms or with lists, depending on the application. Nevertheless, in ECL^{i}PS^e it was decided to have the string data type, so that users that are aware of the advantages and disadvantages of the different data types can always choose the most appropriate one. The system provides efficient builtins for converting from one data type to another.

6.4.1 Choosing The Appropriate Data Type

Strings, atoms and character lists differ in space consumption and in the time needed for performing operations on the data.

Strings vs. Character Lists

Let us first compare strings with character lists. The space consumption of a string is always less than that of the corresponding list. For long strings, it is asymptotically 16 times more compact. Items of both types are allocated on the global stack, which means that the space is reclaimed on failure and on garbage collection.

For the complexity of operations it must be kept in mind that the string type is essentially an array representation, ie. every character in the string can be immediately accessed via its index. The list representation allows only sequential access. The time complexity for extracting a substring when the position is given is therefore only dependent on the size of the substring for strings, while for lists it is also dependent on the position of the substring. Comparing two strings is of the same order as comparing two lists, but faster by a constant factor. If a string is to be processed character by character, this is easier to do using the list representation, since using strings involves keeping index counters and calling the **substring/4** predicate.

The higher memory consumption of lists is sometimes compensated by the property that when two lists are concatenated, only the first one needs to be copied, while the list that makes up the tail of the concatenated list can be shared. When two string are concatenated, both strings must be copied to form the new one.

Strings vs. Atoms

At a first glance, an atom does not look too different from a string. In ECL^{i}PS^e, many predicates accept both strings and atoms (e.g. the file name in open/3) and some predicates are provided in two versions, one for atoms and one for strings (e.g. concat_atoms/3 and concat_strings/3). However, internally these data types are quite different. While a string is simply stored as a character sequence, an atom is mapped into an internal constant. This mapping is done via a table called the *dictionary*. A consequence of this representation is that copying and comparing atoms is a unit time operation, while for strings both is proportional to the string length. On the other hand, each time an atom is read into the system, it has to be looked up and possibly entered into the dictionary, which implies some overhead. The dictionary is a much less dynamic memory area than the global stack. That means that once an atom has been entered there, this space will only be reclaimed by a relatively expensive dictionary garbage collection. It is therefore in general not a good idea to have a program creating new atoms dynamically at runtime.

Atoms should always be preferred when they are involved in unification and matching. As opposed to strings, they can be used for *indexing* clauses of predicates. Consider the following example:

```
[eclipse 1]: [user].
 afather(mary, george).
 afather(john, george).
 afather(sue, harry).
 afather(george, edward).

 sfather("mary", "george").
 sfather("john", "george").
 sfather("sue", "harry").
 sfather("george", "edward").
```

```
user   compiled 676 bytes in 0.00 seconds

yes.
[eclipse 2]: afather(sue,X).

X = harry
yes.
[eclipse 3]: sfather("sue",X).

X = "harry"     More? (;)

no (more) solution.
```

The predicate with atoms is *indexed*, that means that the matching clause is directly selected and the *determinacy* of the call is recognised (the system does not prompt for more solutions). When the names are instead written as strings, the system attempts to unify the call with the first clause, then the second and so on until a match is found. This is much slower than the indexed access. Moreover the call leaves a choicepoint behind (as shown by the more-prompt).

Conclusion

Atoms should be used for representing (naming) the items that a program reasons about, much like enumeration constants in PASCAL. If used like this, an atom is in fact *indivisible* and there should be no need to ever consider the atom name as a sequence of characters.
When a program deals with text processing, it should choose between string and list representation. When there is a lot of manipulation on the single character level, it is probably best to use the character list representation, since this makes it very easy to write recursive predicates walking through the text.
The string type can be viewed as being a compromise between atoms and lists. It should be used when handling large amounts of input, when the extreme flexibility of lists is not needed, when space is a problem or when handling very temporary data.

6.4.2 Builtin Support for Strings

Most ECL^{i}PS^e builtins that deliver text objects (like **getcwd/2**, **read_string/3,4** and many others) return strings. Strings can be created and their contents may be read using the string stream feature (cf. section 12.5.1). By means of the builtins **atom_string/2**, **string_list/2** and **term_string/2**, strings can easily be converted to other data types.

6.4.3 Entering Strings

For input and output, ECL^{i}PS^e strings are by default designated by the surrounding double quotes. Unfortunately, many Prologs use the double quotes as a notation for lists. In some of the compatibility modes the meaning of the quotes is therefore different:

```
[eclipse 1]: X = "text", type_of(X, T).

X = "text"
T = string
```

```
yes.
[eclipse 2]: cprolog.      % redefines the quotes (among other things)
yes.
[eclipse 3]: X = "text", type_of(X, T).

X = [116, 101, 120, 116]
T = compound
yes.
```

Note that although it is no longer possible to create a string by using double quotes, a builtin like **atom_string/2** will still deliver a true string rather than a list.
Even the user can manipulate the quotes by means of the **set_chtab/2** predicate. A quote is defined by setting the character class of the chosen character to `string_quote`, `list_quote` or `atom_quote` respectively. To create a list quote (which is not available by default) one may use:

```
[eclipse 1]: set_chtab(0'‘, list_quote).

yes.
[eclipse 2]: X = ‘text‘, Y = "text", type_of(X, TX), type_of(Y, TY).

X = [116, 101, 120, 116]
TX = compound
Y = "text"
TY = string
yes.
```

6.4.4 Matching Clauses

When Prolog systems look for clauses that match a given call, they use full unification of the goal with the clause head (but usually without the occur check). Sometimes it is useful or necessary to use *pattern matching* instead of full unification, i.e. during the matching only variables in the clause head can be bound, the call variables must not be changed. This means that the call must be an instance of the clause head.
The operator `-?->` at the beginning of the clause body specifies that one-way matching should be used instead of full unification:

```
p(f(X)) :-
    -?->
    q(X).
```

Pattern matching can be used for several purposes:

- Generic pattern matching when looking for clauses whose heads are more general than the call.

- Decomposing *attributed variables* [4]. When an attributed variable occurs in the head of a matching clause, it is not unified with the call argument (which would trigger the unification handlers) but instead, the call argument is decomposed into the variable and its attribute(s):

```
get_attr(X{A}, Attr) :-
    -?->
    A = Attr.
```

This predicate can be used to return the attribute of a given attributed variable and fail if it is not one.

- Replacing other metalogical operations, e.g. **var/1** test. Since a nonvariable in the head of a matching clause matches only a nonvariable, explicit variable tests and/or cuts may become obsolete.

If some argument positions of a matching clause are declared as **output** in a mode declaration, then they are not unified using pattern matching but normal unification, in this case then the variable is normally bound. The above example can thus be also written as

```
:- mode get_attr(?, -).
get_attr(X{A}, A) :-
    -?->
    true.
```

but in this case it must not be called with its second argument already instantiated.

6.5 Soft Cut

Sometimes it is useful to be able to remove a choice point which is not the last one and to keep the following ones, for example when defining an if-then-else construct which backtracks also into the condition. This functionality is usually called *soft cut* in the Prolog folklore.
When you define the operator op(1050, xfx, *−>) and import ***−>/2** from sepia_kernel, then the expression

A *−> B ; C

is evaluated as a soft cut: if A succeeds, B is executed and on backtracking subsequent solutions of A are returned, but C is never executed. If A fails, C is executed. It is similar to **−>/2**, with the exception that **−>/2** cuts both A and the disjunction if A succeeds, whereas ***−>/2** cuts only the disjunction.

Chapter 7

The Compiler

ECL^{i}PS^e has an efficient incremental compiler which compiles Prolog source into the instructions of an abstract machine and they are then executed by an emulator. The compiler is very fast, it compiles about 1000 lines/sec. on a Sun-4, and this makes the usual debugging cycle acceptably short. Unlike other Prolog systems, the ECL^{i}PS^e compiler generates code that can be used for debugging, so that no separate interpreter is necessary, and also the debugged code runs faster. The ECL^{i}PS^e compiler is interactive and incremental, which means that Prolog programs are compiled during a ECL^{i}PS^e session directly into the Prolog database. ECL^{i}PS^e has no means to compile Prolog programs off-line, store the abstract code into a file and load the file during the Prolog session, however compiling a file in ECL^{i}PS^e is as fast as loading the abstract code in other systems and so it makes it obsolete.

7.1 Program Source

When reading the input source, the compiler distinguishes *clauses* and *directives*. Directives are terms with main functor **:-/1** or **?-/1**. When the compiler encounters them, it executes immediately their first argument as a Prolog goal. If this goal succeeds, the compiler continues to the next input term without reporting the answer to the user. If the directive fails, an event is raised.

All other input terms are interpreted as clauses to be compiled. A sequence of consecutive clauses whose heads have the same functor is interpreted as one procedure, and so e.g. if the clauses of one procedure are mixed with directives or with clauses for another procedure, the compiler takes them as several different procedures. To allow the user to write non-consecutive procedures, the compiler raises an event whenever it encounters several procedures with the same name and arity in one file, or when a procedure defined in one file is being redefined in another file. Default action for the former is to emit a warning, for the latter the new procedure just replaces the old one. The library **scattered** redefines the former handler so that procedures which are scattered in one file are accepted as normal static procedures.

7.2 Procedure Types

Procedures can be **static** and **dynamic** and this feature can be queried with the `stability` flag of **get_flag/3**.

Static procedures are compiled as one unit, they are thus executed more efficiently, and they can be modified only by replacing them by another procedure. By contrast, dynamic procedures are compiled clause-wise, they are executed slightly less efficiently, but their source form can also be retrieved, and they can be modified by adding or removing single clauses or clause sequences.
By default all procedures are static, dynamic procedures must be declared by the **dynamic/1** declaration, except that undefined procedures for which **assert/1,2** is called are silently declared as dynamic by the event handler, and so no declaration is needed.
When compiling static procedures, the compiler remembers their position in the file, which can be then queried by **get_flag/3**. The library **scattered** actually uses this feature to retrieve predicates whose clauses are not consecutive.

7.3 Compiler Modes

The compiler has several modes of operation, each mode generating code with different properties. The operating mode is controlled by a set of global flags, which may be modified at any time, even during the compilation so that a part of the program is compiled in a different mode. These flags and the associated modes are listed below.

debug_compile When this flag is **on**, the compiler generates code which can be traced with the debugger. This code can sometimes be significantly less efficient than the untraceable one, and the generated code size is always significantly larger. To generate optimised code, this flag must be switched off. To achieve this, the predicate

```
:- nodbgcomp.
```

can also be called, and it also switches off the **variable_names** flag.

occur_check When this flag is **on**, the compiled code will perform the *occur check* if necessary. This means that every time a variable will be unified with a compound term that might already contain a reference to this variable, the compound term will be scanned for this occurrence and if it is found, the unification fails. In this way, the creation of infinite (cyclic) terms is impossible and thus the behaviour of the system is closer to the first order logic theory. Unifications with the occur check may sometimes be very slow, and most Prolog programs do not need it, because no cyclic terms are created. Note that this flag must be set both at compile time and at runtime in order to actually perform the checks.

dfid_compile When this flag is **on**, the compiler will generate code that keeps track of the number of ancestors of the current goal, which is used by the library **dfid** to execute the **bounded depth-first search**, either as iterative deepening or plain depth limiting.

float_precision This flag specifies if the compiler generates code for `single` or `double` precision floating point arithmetic. It is recommended not to mix code compiled in different modes, because single and double precision numbers do not unify and therefore may cause unexpected failures.

variable_names ECL^{i}PS^e can remember the source variable names of the input variables. When this flag is **on**, the compiled predicates will keep the names of the source variables and will display them whenever the variables are printed. In this case the usage of the global stack and code space is slightly higher (to store the name), and the efficiency of the code is marginally lower. Setting this flag to **check_singletons** has the same effect as **on**, but additionally, the compiler will issue warnings about variables which occur only once in a clause and whose names do not start with an underscore character.

all_dynamic When this flag is **on**, all procedures are compiled as dynamic ones (and there is no equivalent **static/1** declaration). It can be used to port programs from older interpreters which rely heavily on the fact that all predicates in these interpreters were dynamic. Another possible use is to switch it on at the beginning of a file that contains many dynamic predicates and switch it off at its end.

macro_expansion This is in fact a parser flag, is enables or disables the macro transformation (see Chapter 13) for the input source.

goal_expansion Specifies whether to apply goal-macros or not (see Chapter 13).

7.4 Compiler Input

The compiler normally reads a file up to its end. The file end can also be simulated with a clause

```
end_of_file.
```

The file is normally read consecutively, however the compiler uses the normal ECL^{i}PS^e I/O streams, and so if during the compilation the stream pointer is modified (e.g. by **seek/2** or **read/2**), the compiler continues at the specified place[1].
There are several built-in predicates which invoke the compiler:

compile(File) This is the standard compiler predicate. The contents of the file is compiled according to the current state of the global flags.

compile(File, Module) This predicate is used to compile the contents of a file into a specified module, without having to use the module declaration in the file itself.

compile_stream(Stream) This predicate compiles a given stream up to its end or to the `end_of_file` clause. It can be used when the input file is already open, e.g. when the beginning of the file does not contain compiler input, or when the input has to be processed in a non-consecutive way.

compile_term(Clauses) This predicate is used to compile a given term, usually a list of clauses and directives. Unlike **assert/1** it compiles a static procedure, and so it can be used to compile a procedure which is dynamically created and then used as a static one. For more information please refer to [11].

dump(File) This predicate stores the precompiled form of the given file into a file with the suffix **sd** (Sepia Dump). It can be used to speed up the compilation or to deliver program modules whose source should not be readable.

---

[1] An example of using this feature is the library **ifdef**.

ensure_loaded(File) This predicate compiles the specified file if it has not been compiled yet or if it has been modified since the last compilation.

make This predicate recompiles all files that have been modified since their last compilation.

lib(File) This predicate is used to ensure that a specified library file is loaded. If this library is not yet compiled, the system will look in all directories in the `library_path` flag for a file with this name. When the file is found, it is compiled and the system remembers it.

current_compiled_file(File, Time, Module) This predicate returns on backtracking all files that have been compiled in this session, together with the module from where the compilation was done and the modification time stamp of the file at the time it was compiled.

compiled_file(File, Line) This predicate allows to access the compiled file during the compilation. If it is called during the compilation, it returns the name of the file being compiled and the current line in it[2]. If some I/O operations are performed on the compiler stream, it influences the compiler, e.g. some procedures can be omitted and some compiled several times. An example of its use is the library **ifdef** which implements a C-like conditional compilation.

assert(Clause) This predicate compiles the given clause of a dynamic predicate.

7.5 Libraries

There is a number of libraries that support or complement the compiler:

check This library will check all code currently compiled inside the ECL^{i}PS^e session, and print warnings for missing predicates. We recommend its use for program development and for checking finished programs.

ifdef Conditional compilation similar to the C preprocessor.

ptags Creates *vi* tags of specified Prolog source files.

scattered Allows to compile procedures whose clauses are not consecutive.

xref Prints a list of predicates defined and not defined in given files.

7.6 Module Compilation

One source file can contain several modules and one module may spread over several files[3]. The module structure is controlled by the **module/1** or **module_interface/1** directive which tells the compiler that all subsequent input up to the end of file or another module directive will be part of the given module.

---

[2] This is in fact ambiguous; the system predicate **compiled_stream/1** which is exported from the module **sepia_kernel** is more precise.

[3] This style is not recommended.

When it encounters the **module_interface/1** directive, the compiler first erases previous contents of this module, if there was any, before starting to compile predicates into it. This means that if the contents of a module has to be generated incrementally, the module directive cannot be used because the previous contents of the module would be destroyed. In this case the predicate **compile(File, Module)** should be used.

7.7 Mode Declarations

Mode declarations are a way for the user to give some additional information to the compiler, thus enabling it to do a better job. The ECL^{i}PS^e compiler makes use of the mode information mainly to improve indexing and to reduce code size.

Mode declarations are optional. They specify the argument instantiation patterns that a predicate will be called with at runtime, for example:

```
:- mode p(+), q(-), r(++, ?).
```

The possible argument modes and their meaning are:

+ - The argument is instantiated, i.e. it is not a variable.

++ - The argument is ground.

− - The argument is not instantiated, it must be a free variable without any constraints, especially it must not occur in any other argument and it cannot be a suspending variable.

? - The mode is not known or it is neither of the above ones.

Note that, if the actual instantiation of a predicate call violates its mode declaration, the behaviour is undefined. Usually, an unexpected failure occurs in this case.

7.8 Inlining

To improve efficiency, calls to user-defined predicates can be preprocessed and transformed at compile time. The directive **inline/2**, e.g.

```
:- inline(mypred/1, mytranspred/2).
```

arranges for mytranspred/2 to be invoked at compile time for each call to the predicate mypred/1 before this call is being compiled.

The transformation predicate receives the original call to mypred/1 as its first argument, and is expected to return a replacement goal in its second argument. This replacement goal replaces the original call in the compiled code. Usually, the replacement goal would be sematically equivalent, but more efficient than the original goal. When the transformation predicate fails, the original goal is not replaced.

Typically, a predicate would be defined together with the corresponding inlining transformation predicate, e.g.

```
:- inline(double/2, trans_double/2).

double(X, Y) :-
```

```
        Y is 2*X.

trans_double(double(X, Y), Y=Result) :-
        not nonground(X),    % if X already known at compile time:
        Result is 2*X.       % do calculation at compile time!
```

All compiled calls to double/2 will now be preprocessed by being passed to trans_double/2. E.g. if we now compile the following predicate involving double/2

```
sample :-
        double(12,Y), ...,  double(Y,Z).
```

the first call to double will be replaced by `Y=24` while the second one will be unaffected. The code that the compiler sees and compiles is therefore

```
sample :-
        Y=24, ...,  double(Y,Z).
```

Note that meta-calls (e.g. via **call/1**) are never preprocessed, they always go directly to the definition of double/2.
Transformation can be disabled for debugging purposes by adding

```
:- pragma(noexpand).
```

to the compiled file, or by setting the gobal flag

```
:- set_flag(goal_expansion, off).
```

7.9 Compiler Pragmas

Compiler pragmas are compiler directives which instruct the compiler to emit a particular code type. Their syntax is similar to directives:

```
:- pragma(Option).
```

It is not possible to have several pragmas grouped together and separated by commas like goals, every pragma must be specified separately. *Option* can be one of the following:

- **debug** - generate code which can be inspected with the debugger. This overrides the global setting of the `debug_compile` flag.
- **nodebug** - generate optimized code with no debugger support. This overrides the global setting of the `debug_compile` flag.
- **silent_debug** - generate code which cannot be inspected by the debugger, but which allows to debug predicates called by it. This is similar to setting the `leash` flag of all subgoals in the following clauses to `notrace`. This option is useful e.g. for library predicates which call other Prolog predicates: the user wants to see in the debugger the call to the library predicate and to the invoked predicate, but no internal calls in the library predicates.
- **expand** - do in-line expansion of some subgoals, like =/2, **is/2** and others. This code can still be inspected with the debugger but the expanded subgoals look differently than in the normal debugged code, or their arguments cannot be seen. This pragma overrides the global setting of the `goal_expansion` flag.

- **noexpand** - inhibit the in-line goal expansion. This pragma overrides the global setting of the `goal_expansion` flag.
- **skip** - set the `skip` flag of all following predicates to `on`.
- **noskip** - set the `skip` flag of all following predicates to `off`.
- **system** - set the `type` flag of all following predicates to `built_in`. Moreover, all following predicates will have unspecified `source_file` and `source_line` flags.

By default, the compiler works as if the pragmas **debug**, **expand** and **noskip** were specified. The pragma is active from its specification in the file until the file end or until it is disabled by another pragma. Recursive compilations or calls to other compiling predicates are not affected by the pragma. Pragmas which have the same effect as global flags override the global flags if they specify more optimized code. For instance, the pragma **debug** has no effect if the global flag `debug_compile` is `off`, but the pragma **nodebug** overrides the global flag `debug_compile` being `on`.

The pragmas are useful mainly for libraries and other programs that should be always compiled in a particular mode independently of the global flags setting.

7.10 Writing Efficient Code

The ECL^{i}PS^e compiler tries its best, however there are some constructs which can be compiled more efficiently than others. On the other hand, many Prolog programmers overemphasise the importance of efficient code and write completely unreadable programs which can be only hardly maintained and which are only marginally faster than simple, straightforward and readable programs. The advice is therefore **Try the simple and straightforward solution first!** The second rule is to keep this original program even if you try to optimise it. You may find out that the optimisation was not worth the effort.

To achieve the maximum speed of your programs, you must produce the optimised code with the flag `debug_compile` being off, e.g. by calling **nodbgcomp/0** or **set_flag(debug_compile, off)**, or using the pragma **nodebug**. Setting the flag `variable_names` can also cause slight performance degradations and it is thus better to have it off, unless variable names have to be kept. Unlike in the previous releases, the flag `coroutine` has now no influence on the execution speed. Some programs spend a lot of time in the garbage collection, collecting the stacks and/or the dictionary. If the space is known to be deallocated anyway, e.g. on failure, the programs can be often speeded up considerably by switching the garbage collector off or by increasing the `gc_interval` flag. As the global stack expands automatically, this does not cause any stack overflow, but it may of course exhaust the machine memory.

When the program is running and its speed is still not satisfactory, use the profiling tools. The profiler can tell you which predicates are the most expensive ones, and the statistics tool tells you why. A program may spend its time in a predicate because the predicate itself is very time consuming, or because it was frequently executed. The statistics tool gives you this information. It can also tell whether the predicate was slow because it has created a choice point or because there was too much backtracking due to bad indexing.

One of the very important points is the selection of the clause that matches the current call. If there is only one clause that can potentially match, the compiler is expected to recognise this and generate code that will directly execute the right clause instead of trying several subsequent clauses until the matching one is found. Unlike most of the current Prolog compilers, the ECL^{i}PS^e compiler tries to base this selection (*indexing*) on the most suitable argument of the predicate[4]. It is therefore not necessary to reorder the predicate arguments so that the first one is the crucial argument for indexing. However, the decision is still based only on one argument. If it is necessary to look at two arguments in order to select the matching clause, e.g. in

```
p(a, a) :- a.
p(b, a) :- b.
p(a, b) :- c.
p(d, b) :- d.
p(b, c) :- e.
```

and if it is crucial that this procedure is executed as fast as possible, it is necessary to define an auxiliary procedure which can be indexed on the other argument:

```
p(X, a) :- pa(X).
p(X, b) :- pb(X).
p(b, c) :- e.

pa(a) :- a. pa(b) :- b.

pb(a) :- c. pb(d) :- d.
```

The compiler also tries to use for indexing all type-testing information that appears at the beginning of the clause body:

- Type testing predicates **free/1**, **var/1**, **meta/1**, **atom/1**, **integer/1**, **real/1**, **number/1**, **string/1**, **atomic/1**, **compound/1**, **nonvar/1** and **nonground/1**.
- Explicit unification and value testing **=/2**, **==/2**, **\==/2** and **\=/2**.
- Combinations of tests with **,/2**, **;/2**, **not/1**, **−>/2**.
- Arithmetic testing predicates **</2**, **=</2**, **>/2**, **>=/2** if one argument is an integer constant and the other one known to be of the integer type.
- A cut after the type tests.

If the compiler can decide about the clause selection at compile time, the type tests are never executed and thus they incur no overhead. When the clauses are not disjoint because of the type tests, either a cut after the test or more tests into the other clauses can be added. For example, the following procedure will be recognised as deterministic and all tests are optimised away:

```
% a procedure without cuts
p(X) :- var(X), ...
p(X) :- (atom(X); integer(X)), X \= [], ...
p(X) :- nonvar(X), X = [_|_], ...
p(X) :- nonvar(X), X = [], ...
```

---

[4] The standard approach is to index only on the first argument

Another example:

```
% A procedure with cuts
p(X{_}) ?- !, ...
p(X) :- var(X), !, ...
p(X) :- integer(X), ...
p(X) :- real(X), ...
p([H|T]) :- ...
p([]) :- ...
```

Integers less than or greater than a constant can also be recognised by the compiler:

```
p(X) :- integer(X), X < 5, ...
p(7) :- ...
p(9) :- ...
p(X) :- integer(X), X >= 15, ...
```

If the clause contains tests of several head arguments, only the first one is taken into account for indexing.
Here are some more hints for efficient coding with ECL^{i}PS^e:

- Arguments which are repeated in the clause head and in the first regular goal in the body do not require any data moving and thus they do not cost anything. For example,

  ```
  p(X, Y, Z, T, U) :- q(X, Y, Z, T, U).
  ```

  is as expensive as

  ```
  p :- q.
  ```

  On the other hand, switching arguments requires data moves and so

  ```
  p(A, B, C) :- q(B, C, A).
  ```

  is significantly more expensive.

- When accessing an argument of a structure whose functor is known, unification is better than **arg/3**. Note, however, that for better maintainability the library **structures** should be used to define the structures.

- Tests are generally rather slow unless they can be compiled away (see *indexing*).

- When processing all arguments of a structure, using **=../2** and list predicates is always faster, more readable and easier analyzable by automated tools than using **functor/3** and **arg/3** loops.

- Similarly, when adding one new element to a structure, using **=../2** and **append/3** is faster than functor/arg.

- Waking is less expensive than metacalling and more expensive than direct calling. Metacalls, although generally slow, are still a lot faster than in some other Prolog systems.

- Sorting using **sort/2** is very efficient and it does not use much space. Using **setof/3**, **findall/3** etc. is also efficient enough to be used every time a list of all solutions is needed.

- using **not not Goal** is optimised in the compiler to use only one choice point.

- **=/2**, when expanded by the compiler, is faster than **==/2** or **=:=/2**.

- **call_explicit/2** is optimised away by the compiler if both argument are known.

- Using several clauses is much more efficient than using a disjunction if the clause heads contain nonvariables which can be used for indexing. If no indexing can be made anyway, using a disjunction is slightly faster.

- Conditionals with $->$; are compiled more efficiently if the condition is a simple built-in test. However, using several clauses can be faster if the compiler optimises the test away.

7.11 Abstract Code Listing

The built-in predicate **als/1** lists the abstract code of the given predicate and it can thus be used by experts to check if the predicate was compiled as expected.

Chapter 8

Parallel Execution

ECL^{i}PS^e implements **Or-Parallelism**. This means that (on parallel hardware) the alternatives of non-deterministic choices can be explored in parallel rather than through sequential backtracking.
Note that this feature is currently not actively supported!

8.1 Using the Parallel System

A parallel ECL^{i}PS^e session consists of a number of processes that jointly execute your program in parallel. They are called **workers**. On a multi-processor machine, the number of workers should match the number of physical processors available on the machine. When there are more workers than processors, then several workers must share a processor which is slower than having just one worker per processor. When there are more processors than workers the power of the machine cannot be fully exploited since some processors may be left idle. Note that ECL^{i}PS^e allows you to add and remove workers during program execution.
A parallel session is started as follows:

```
% peclipse
ECRC Common Logic Programming System [sepia opium megalog parallel]
Version 3.5.0, Copyright ECRC GmbH, Wed Nov 31 10:13 1994
[eclipse 1]:
```

Parallel ECL^{i}PS^e takes the following additional command line options:

–w <number of workers> The initial number of workers. The default is 1. The space between w and the number is optional.

–wmi This option pops up an interactive worker manager window which allows you to dynamically control worker configuration during the session.

–wv Be verbose while starting up the workers.

–wx <worker executable> Use the specified sequential eclipse for the workers rather than the default one.

Apart from that, parallel ECL^{i}PS^e behaves much like the sequential version, in particular, all sequential command line options apply.

8.2 Parallel Programming Constructs

8.2.1 Parallel Annotation

The basic language construct for parallelising a program is the **parallel/1** annotation, for example

```
:- parallel colour/1.
colour(red).
colour(green).
colour(blue).
```

Without the annotation, the system would backtrack sequentially through the alternative solutions of colour(X), and X would successively take the values red, green and blue. When using the parallel annotation, all three solutions are (potentially) explored in parallel by different workers, so one worker continues with X=blue, another with X=red and a third one with X=green. Note that for a parallel predicate

- the order of the clauses is not relevant
- there is no programmer control over which worker takes which alternative[1].

Note that not only is colour/1 executed in parallel, but also the resulting alternative continuations, e.g. if a program contains the sequence

```
..., colour(X), work(X), ...
```

then the goals work(red), work(blue) and work(green) will also be executed in parallel, as a consequence of the nondeterminism in colour/1.
For many applications, it is enough to add parallel annotations to a small number of central points in a program in order to achieve parallel speedup.

8.2.2 Built-In

A parallel primitive that is useful to build upon is **fork/2**. It behaves as if defined by

```
:- parallel fork/2.
fork(N, N).
...
fork(N, 2).
fork(N, 1).
```

i.e. a call of `fork(100, X)` generates the numbers between 1 and 100 in parallel, where the limit does not have to be fixed at compile time.

8.2.3 Utility Libraries

The library `par_util` (see appendix A.9) contains some predicates that are frequently used and are built on top of the above mentioned primitives. **par_member/2**, **par_delete/3**,

---

[1] this is controlled by an automatic scheduler.

par_between/3, **par_maplist/3** etc. are parallel versions of the corresponding sequential predicates. It also contains **&/2** which implements a restricted form of AND-parallelism. The finite domain solver library library(fd) provides **par_indomain/1**, a parallel version of **indomain/1**. The finite set solver library(conjunto) provides **par_refine/1**, a parallel version of **refine/1**. The library `elipsys` provides compatibility with the ElipSys system and uses the `par_util` library.

8.3 Controlling and Analysing the Execution

8.3.1 Which worker executes this code?

Although the parallelism is controlled automatically, during program development it is useful to find out how the system actually behaves. In the following example we use **get_flag/2** to find out which worker finds which solution:

```
[eclipse 1]: fork(10,X), get_flag(worker,W).

X = 6
W = 4     More? (;)

X = 7
W = 2     More? (;)
```

The solution X=6 has been found on worker 4 and X=7 on worker 2. In a parallel session, the number identifying the worker is greater or equal to 1, in a sequential session it is 0.

8.3.2 Measuring Runtimes

Measuring runtimes of parallel executions is especially tricky, since the processes involved have their own local timers, e.g. for measuring cputime. The simplest way is to measure true elapsed time instead of cputimes, and use an otherwise unloaded machine. The primitive that should be used to read the clock is

```
statistics(session_time, T)
```

where `T` is the number of seconds elapsed since the start of the parallel session.

8.3.3 Amount of Parallelism

On hardware that provides a high-precision low-overhead timer, the predicate **par_statistics/0** from the `par_util` library can be used. It prints a compact summary information about where the different workers spent their time, together with some other data about the parallel execution. For example:

```
[eclipse 7]: par_statistics_reset, queens(10), par_statistics.
 Wrkr Jobs Prun Published Copy      Copied  Idling Working Copying Scheduling
   ID    #    # cpts alts    #       bytes      ms      ms      ms       ms

    1   24    0   34   34   11       30208      50    7591      10      157
    2   24    0   16   16   15       29088     147    7638       8       18
 3   35    0   38   38   18        39656     134    7604       38      35
 4   30    0   25   25   13        36668     192    7519       34      24
```

8.3.4 Adding and Removing Workers

Workers in a parallel ECL^{i}PS^e session can be in one of two states: active (awake) or asleep. As one would expect, only active workers take part in any computation. A newly created worker's default state is active. New workers can be added and the number of active workers can be altered using the worker manager interface. These actions are performed asynchronously, thus the configuration can be altered even during parallel execution: a newly added worker will join the computation and when a worker is sent to sleep, it will stop working at an appropriate point of the execution. Note that the worker manager interface be started either using the `-wmi` command-line option or via the **wm_set/2** builtin.

Worker management is also possible under program control. Use the builtins **wm_get/2** to inquire about, and **wm_set/2** to affect the worker configuration. For example, to enquire about the number of workers currently active:

```
[eclipse 1]: wm_get(workers(Host), Awake+Asleep).

Host = "turing"
Awake = 2
Asleep = 1
yes.
```

This means that the there are a total of 3 workers on the machine "turing", out of which 2 are active. In the above example, if one wanted to have only one worker active:

```
[eclipse 2]: wm_set(workers(turing),1), wm_get(workers(turing),Status).

Status = 1 + 2
yes.
```

8.4 Parallelism and Side Effects

In the current version, all side effect builtins like assert, record, setval and the I/O predicates work on resources that are shared between the workers and are accessed in a mutually exclusive way. For example, when two workers write onto a stream at the same time, the access will be sequentialised such that one of them writes first, and then the other. The order is unspecified. It is however, expected that the internal database builtins (such as assert and retract) will not be fully supported in the next major release (which will allow the system to execute on distributed memory platforms).

The current version also provides an explicit mutual exclusion primitive **mutex/2**. It can be used to make a sequence of several goals atomic, ie. to make sure that the execution of a piece of code is not interleaved with the execution of similarly protected code on other workers. For example, the following code will make sure that every list is printed in one chunk, even when several workers execute different instances of atomic_write_list/1 in parallel:

```
:- mutex_init(my_lock).
```

```
atomic_write_list(List) :-
    mutex(my_lock, write_list(List)).
write_list([]) :- nl.
write_list([X|Xs]) :- writeln(X), write_list(Xs).
```

8.5 Parallel Cuts

The semantics of cut follows the philosophy that the order of clauses in a parallel predicate is not relevant. E.g. a predicate like

```
:- parallel p/0.
p :- writeln(a).
p :- !, writeln(b).
p :- !, writeln(c).
```

may print 'a' and 'b', 'a' and 'c', only 'b' or only 'c'. It depends on which cut is executed first, and whether it is executed before or after 'a' has been printed.

8.6 Restrictions

Some features of sequential ECL^{i}PS^e are not fully supported or make no sense in the parallel version. These include:

- The Debugger: The prolog debugger cannot be used to trace parallel sessions i.e. programs in which more than one worker is active.

- File queries: When a compiled file contains queries (i.e. lines of the form `:- <goal>.` or `?- <goal>.`, then these goals will not be executed in parallel. To start a parallel computation, either start it from the toplevel, or use the -e command line option.

- Dynamic predicates can currently not be declared parallel.

- Dynamic loading: This feature is not currently available in the parallel version, but should be available in subsequent releases.

- Unix process related primitives: Currently most primitives such as socket, accept, listen, exec, wait which depend on private data structures created by the operating system are not fully supported. For example, it is currently up to the user to guarantee that some operations on a socket (such as **accept/3**) are only performed on the worker on which the socket in question was created. In future releases most of these restrictions will be removed, however, it is likely that some complex builtins (e.g. **select/3**) will not be completely supported.

- Timing statistics: The values returned by the **statistics(times,_)** builtin do not make sense in a parallel session, since they only refer to timers local to the worker on which the call is executed. As noted earlier, elapsed time for a parallel session should be measured using the `statistics(session_time,Time)` builtin. It is envisaged that the the **wm_get/2** builtin will be expanded in the future in order to allow the user to query total cpu usage of all active workers.

8.7 Troubleshooting

8.7.1 No Space

When the system complains about lack of swap space, then it is likely that there is no space in your /tmp filesystem. In this case set the environment variable ECLIPSETMP to a directory that has enough space to hold the temporary files that the system uses at runtime. It is recommended to use a different directory for different hosts, e.g.

```
setenv ECLIPSETMP ~/tmp/`hostname`
```

8.7.2 Process structure

A parallel ECL^{i}PS^e session consists of

- The worker manager process `peclipse`. This is usually the parent process of the whole parallel session. Its process id is used to identify a parallel session. All temporary files belonging to a parallel session contain this process id in their name.
- A number of worker processes `eclipse.exec`. They are normal sequential ECL^{i}PS^e processes that do the actual work in parallel.
- A **name server** process `nsrv`. The name server is a support process and is not really part of a parallel ECL^{i}PS^e session. A name server process is launched when a parallel session is started on a machine for the first time. It stays in existence even after the session is finished and is reused by concurrent or future parallel sessions on the same machine. The name server puts some data files into the $ECLIPSETMP (or /tmp) directory. Their names start with `nsrv`.

8.7.3 Crash recovery

After a machine crash or after an abnormal termination of a parallel session, it may be necessary to kill some processes and to remove files in the temporary directory (if ECLIPSETMP is not set, this defaults to /tmp):

1. When you still have a worker manager window, try exiting using the **EXIT** button. If that does not help:
2. Kill the peclipse process and then any remaining workers (eclipse.exec). This will most likely require a hard kill (-9).
3. Remove temporary files $ECLIPSETMP/session_id.*.map where session_id is the process number of the peclipse process.

If it is not possible to restart a parallel session after this cleanup, then the name server may be corrupted as well. In this case:

1. Kill the nsrv process. Use kill -9 only if the process does not go away after a soft kill.
2. Remove $ECLIPSETMP/nsrv* if these files still exist.

Chapter 9

Module System

The ECL^{i}PS^e module system has been designed to meet the following criteria:

1. *Be a structuring tool* allowing to develop and maintain large applications.
2. *Avoid name clashes* by having a separate name space for each module.
3. *Support privacy* by allowing restrictions on the access to certain information of a module.
4. *Be incremental* by giving as much freedom as possible in the order in which the predicate are defined and re-defined and the visibility declared and changed and by allowing the changes to be done dynamically.
5. *Be transparent to non-modular applications.* A Prolog program written in a non-modular system should run without changes when put into a unique module.

These requirements make it clear that the main function of the module system must be to regulate the access to certain (or all) Prolog items. For instance, requirement 2 states that the module system must be able to recognise that there are two items with the same name, but used independently, and provide the means to restrict their accessibility so that no conflict arises. The module system is therefore concerned with the *access* to certain items. The main task of the ECL^{i}PS^e module system is to control the access in such a way that the above requirements can be fulfilled.

9.1 Terminology

Modular Item: a modular item is a Prolog item that is affected by the module system. In ECL^{i}PS^e the different *types* of modular items are e.g. predicate, operator, record, array and global variable.

Visibility: the visibility of a modular item specifies under what conditions the item is visible from a given module. The visibility of a modular item is defined by Prolog predicates called *visibility declarations.*

Module Interface: the module interface is the set of declarations and definitions that defines the connection of a module to the others. Parts of the interface can be dynamically modified.

Module Body: the module body is the set of modular items that are defined in a module.

Definition Module: the definition module of a modular item is the module where this item is defined. The definition module is a static notion.

Caller Module: the caller module of a goal is the definition module of the predicate that calls that goal. The caller module is a dynamic notion.

9.2 Basic Properties

The module system of ECL^{i}PS^e has the following properties:

- The module system is *flat*, no module is part of another module and module names must be unique.

- The modular items are predicates, operators, macros, records, arrays, global variables, syntax classes, syntax option flags and metaterm attributes. In particular, the syntax recognised by the parser is completely module-dependent and can be independently changed in different modules. Only predicates can be exported and imported.

- The module system is *procedure based*, that is the visibility of the procedure is module dependent, but the functors and the syntax are global. However it is possible to have local properties for functors (i.e. operators, records, arrays and global variables may be defined locally).

- The module is a completely dynamic object, its body part and the visibility of its predicates can be changed during the session or execution of a goal. The module interface is syntactically separated from the module body and it can be modified only by recompiling the whole module.

- In any particular module, at most one among all items with the same name and type is visible.

  However items with the same name but different types are accessed independently. This property is not a necessary conclusion of our requirements, it is more a question of philosophy: we do believe that e.g. the procedure `p/0` and the term recorded under the key `p/0` do not necessarily have to have the same visibility.

- The module system affects the runtime system and the compiler, but has no effect on the underlying abstract machine. This means that the pure Prolog execution does not suffer from an overhead when modules are extensively used.

- ECL^{i}PS^e module system can be described as a set of access rules that defines what modular item is visible from what module (according to the current module interfaces) when a goal is executed.

  Those rules consist in finding a *definition module* for a modular item that occurs in a goal called from a given *caller module*.

  The access rules of ECL^{i}PS^e are based on the caller module.

9.3 Modules and the Top Level Loop

Everything that is typed to the top level prompt happens in a module called the top level module. This module name is displayed in the prompt and the default is `eclipse`. Note that this module is in no way special. It is just the default for typing in queries, and it is initially empty. When a predicate is compiled from the toplevel, then it is defined in the top level module, e.g.:

```
[eclipse 1]: [user].
 p(hello).
 user       compiled traceable 40 bytes in 0.00 seconds

yes.
[eclipse 2]: get_flag(p/1, definition_module, M).

M = eclipse
yes.
```

The call to **get_flag/3** tells us that p/1 is defined in module `eclipse`. The top level module can be changed by using **module/1**:

```
[eclipse 3]: module(mymodule).
warning: creating a new module in module(mymodule)
[mymodule 4]: p(X).
calling an undefined procedure p(X) in module mymodule
```

Since `mymodule` did not exist previously, it is automatically created. The predicate p/1 that was defined in module `eclipse` is not visible in `mymodule`, the attempt to call it results in an error message.

Similarly, when a file is compiled from the toplevel, and the file does not contain any module directives, then its content is compiled into the top level module.

9.4 Modules and Source Files

When a source file contains no module directives, it is compiled into the module from which the compilation was invoked. This makes it possible to write small programs without caring about modules. However, serious applications should be structured into modules.

A proper module definition consists of

module interface starting with a **module_interface/1** directive

module body starting with a **begin_module/1** directive

Both parts of the module can be in a single file where the interface part is followed by the body part. Alternatively, the interface can be in one file and the body can be spread over one or more additional files, each beginning with a **begin_module/1** directive. In this case, the interface file has to be compiled first.

The directive **module_interface/1** will first erase the module if it already exists, i.e. remove all items and interfaces contained in the module and then create a new empty module. This is necessary to maintain the integrity of the data and perform a complete recompilation of the

module. The module interface contains the declarations and definitions of all modular items which this module shares with other modules that use it. As long as no further module directive occurs, all clauses and queries of the file have this module as definition module. The effect of **module_interface/1** ends at the next **module_interface/1** or **begin_module/1** directive or a the end of the file. If the file contains queries to compile another file containing module directives, this is done and then the module of the current file is resumed.
module/1 in a file is an obsolete shorthand for a module with an empty interface (and it may not be available in future releases).

9.4.1 The Module Interface

The module interface is a set of goals and procedure definitions which are made available to other modules. Moreover, the module interface may contain the definition of the syntax which is needed to read its body part. The notion of the module interface serves several purposes:

- It specifies which items are to be shared with other modules.
- It makes possible to share modular items that cannot be directly exported and imported.
- It specifies the part of the module, which has to be compiled and executed even if the module is only processed by source-level tools. For instance, if the module body contains terms with an associated macro, the corresponding macro transformation predicate has to be compiled even if the module body is only being read in.

If the module is used by another module via the predicate **use_module/1**, all queries that appear in the module interface part will be executed in the other module, except for **export/1** and **global/1** predicates. Therefore, declarations of local operators, macros, records etc. will be executed in the caller module and these items will become available there. The macro transformation predicates must be exported to be visible in the using module. The **export/1** queries in the module interface are conceptually replaced by **import_from/2** and thus the exported predicates are imported in the caller module. Predicate definitions which appear in the module interface will not be repeated when the module is used; if they have to be accessible in another module, they should be exported instead.
Here is an sample module that can serve as a guideline of which declarations to place where:

```
:- module_interface(mod).

% syntax directives for this and for importing modules
:- op(300, xfx, >>>).
:- set_chtab(0'$, lower_case).
:- define_macro((|)/2, trans_bar/2, []).

% libraries to use here and in the importing modules
:- use_module(library(cio)).

% predicates to import here and in importing modules
:- import current_predicate_body/2 from sepia_kernel

% predicates defined globally by this module
```

```
:- global g/1.

% predicates exported from this module
:- export p/1, t/1, e/1.
:- export trans_bar/2.     % needed for the macro above

% type declarations for exported tools and externals
:- tool(t/1).
:- external(e/1).

% definition of macro transformation predicates
trans_bar(no_macro_expansion('|'(X,Y)), (X;Y)).

:- begin_module(mod).       % the module body

% syntax directives for this module only
:- op(300, xfx, <<<).

% predicates to import only here
:- import setof_body/4 from sepia_kernel.

% special type predicate definitions
:- tool(t/1, t_body/2).
:- external(e/1, my_c_function).

% normal predicate definitions
g(hello).
p(world).
```

When calling **use_module(mod)**, all queries in the interface will be executed and the macro transformation correctly defined. The module interface may also contains goals not directly related to the definition module, like e.g. the **import from** in the above example.

The module body can also contain declaration of local modular items, however they remain local in the definition module. The definition of the module body starts with the **begin_module/1** directive and it can be used only if the module already exists, either created by **create_module/1**, **compile/2** or by defining the module interface. There may be several **begin_module/1** directives for the same module. When they are compiled, their contents is *added* to the module body.

9.5 Creating and Erasing Modules at Runtime

A module can also be created explicitly by a running program with **create_module/1** and erased with **erase_module/1**. The latter should be used with care, erasing a module while a predicate defined in that module is being executed can provoke unpredictable results.

9.6 Visibility of Predicates

In ECL^{i}PS^e, a predicate can have different levels of visibility. In a given module, a predicate has one and only one of the following visibilities:

local: a procedure declared as local is only visible from its definition module.

export: a procedure declared as exported is visible from its definition module and from all modules that import it.

import: a procedure declared as imported is visible from the module it is imported to. An imported procedure is always equal to the exported procedure it imports (if any).

global: a procedure declared as global is visible from all modules where no local, exported or imported procedure with the same name is visible. There can be only one global procedure of a given name at any time. Global procedures should be used with care, the normal module interface mechanism is to be preferred.

A procedure whose visibility is not declared is local by default. Therefore, if a predicate shall be visible from outside of its definition module, it must be explicitly declared exported.
Moreover, a procedure may be visible but not defined since it can be declared before it is compiled or asserted.

9.6.1 Access rule for predicates as goal name

Considering the goal *Pred(A1, ... AN)* invoked from module *Caller_module*:
If there is a local, exported, imported or global predicate named *Pred/N* defined in *Caller_module*, this predicate is the visible one. Else if there is a global predicate declared in any module, this one is visible, else there is no visible predicate *Pred/N*.
It is also possible to call a non-visible predicate, provided it is global or exported from its definition module. This is done using the **:/2** primitive, e.g.

```
def_module:p(X,Y)
```

9.6.2 Access rules for predicates as arguments of built-ins

Not all built-ins access their predicate arguments with the same access rule. Some built-ins access the predicate *visible from* the caller module, others access the predicate *defined in* the caller module. E.g. **spy/1** can be used to set spy points on imported predicates or predicates defined as global in another module than the caller module, whereas **abolish/1** can only be used to abolish predicates defined in the caller module or import links (only the link, not the corresponding exported predicate) declared in the caller module. Most of the time, the rule to apply can be found intuitively.
Considering the goal *Pred(A1, ..., Pred_arg, ..., AN)* called from module *Caller_module* where *Pred_arg* specifies a predicate (e.g. like in **spy(p/1)** or in **assert(p(a))**). The visibility rule applied for *Pred_arg* depends on the predicate *Pred/N*:

- if *Pred/N* is one of the following, the accessed predicate is the predicate specified by *Pred_arg* and **defined** (as local, exported or global) in *Caller_module*.

- built-ins that declare and change the visibilities: **abolish/1**, **export/1**, **global/1**, **import_from/2**, **local/1**;
- built-ins that compile or define predicates (those predicates create local procedures in the caller module): **b_external/2**, **compile/1,2**, **./2**, **compile_stream/1**, **compile_term/1**, **external/2**, **lib/1,2**, **mode/1**, **tool/2**;
- built-ins that compile, access and modify dynamic procedures: **assert/1**, **asserta/1**, **clause/1,2**, **dynamic/1**, **listing/0,1**, **retract/1**, **retract_all/1**.

  Those predicates will raise errors when used on imported predicates (since imported predicates are always defined as exported in an other module). The import link must be cut explicitly with **abolish/1**.

- if *Pred/N* is one of the following, the accessed predicate is the predicate specified by *Pred_arg* **visible** from *Caller_module*.
  - built-ins that access predicates flags: **get_flag/3**, **set_flag/3**, **is_built_in/1**, **is_dynamic/1**, **is_predicate/1**, **spy/1**, **traceable/1**, **skipped/1**, **untraceable/1**, **unskipped/1**;
  - built-ins that declare procedure types: **b_external/1**, **external/1**, **tool/1**;
  - built-ins that call their goals arguments: **->/2**, **;/2**, **,/2**, **</2**, **<=/2**, **=/2**, **>/2**, **>=/2**, **=:=/2**, **bagof/3**, **block/3**, **call/1**, **coverof/3**, **fail_if/1**, **not/1**, **findall/3**, **forall/2**, **is/2**, **once/1**, **phrase/2,3**, **setof/3**;
  - other built-ins that access visible predicates: **pred/1**, **current_built_in/1**, **current_predicate/1**, **set_error_handler/2**, **set_interrupt_handler/2**.

9.6.3 Defining and modifying the visibility

There are 5 visibility declaration predicates:
local PredList declares the predicates in *PredList* (maybe not yet defined) as local in the caller module.
export PredList declares the predicates in *PredList* (maybe not yet defined) as exported in the caller module.
global PredList declares the predicates in *PredList* (maybe not yet defined) as global in the caller module.
import PredList from Module declares the predicates in *PredList* to be imported predicates in the caller module. Each of them are linked to their corresponding exported predicates (maybe not yet) defined in *Module*.
abolish PredList removes the declaration and the definition of the predicates in *Predlist* declared or defined in the caller module. As the visibility declaration predicates act on the caller module only, abolishing an imported predicate does not affect the exported predicate itself but only the import link.
The predicate visibility may be changed at any time.
With respect to requirement 2, some visibility changes are restricted: import links (created with **import_from/2**) must be cut down explicitly (with *abolish/1*) before defining a new visibility. Vice-versa a local, exported or global declaration or definition must be abolished before an import link is created with **import_from/2**.
Warnings are raised when redundant declarations occur (e.g. declaring twice the same predicate as local).

9.6.4 Tools

There are predicates in a modular Prolog system that need to work in the space of other modules rather than in the module where they are defined. The most common case is when a predicate is a meta-predicate, i.e. when a predicate has a goal as argument. Other cases are predicates that have other module-dependent arguments (e.g. a record key) or I/O predicates that need to be executed in a certain module context in order to obey the syntax of this module. We call these predicates **tool** predicates.

Consider the case of a predicate *pred/1* which has a goal argument: If the argument goal is *called* from that predicate, it will be executed in the module space of the definition module of the predicate *pred/1* and not in the one of the caller module.

```
pred(Goal) :-
        ...
        call(Goal), % Goal is called from the definition module of pred/1
        ... .
```

When the goal argument of a goal must be used in the module space of the *caller* of that predicate, we need an additional module argument.

```
pred(Goal, CallerModule) :-
        ...
        call(Goal)@CallerModule,   % Goal is called from CallerModule
        ... .
```

To prevent the user having to supply the caller module argument to such predicates (which is likely to cause problem when the predicate that must supply the module argument is recompiled in another module) and to fulfill requirement 3 concerning the privacy (refer to section 9.9), the concept of tool interface has been developed.

The *tool interface* is a predicate (defined with **tool/2**) that is connected to a *tool body* (whose arity is one more than the arity of its tool interface). Its functionality is to automatically supply the caller module of the interface to the last argument of the body procedure and to call that body procedure. Let us assume we have compiled the following tool in the module `eclipse`

```
:- global current_def/1.
:- tool(current_def/1, current_def/2).

current_def(Pred, Module) :-
        % get the predicates visible from Module
        current_predicate(Pred)@Module,
        % get the flag of Pred visible from Module
        get_flag(Pred, definition_module, Module)@Module.
```

Using the debugger we can easily see how the tool interface supplies the caller module to its tool body.

```
[eclipse 2]: trace(current_def(X)).
Start debugging - creep mode
  (1) 0  CALL   current_def(X) (dbg)?- creep             % type c
  (2) 1  CALL   current_def(X, eclipse) (dbg)?- no debug  % type n
```

```
X = current_def / 2     More? (;)

X = current_def / 1     More? (;)

no (more) solution.
```

And from a new empty module we have:

```
[eclipse 3]: module(new_module).
[new_module]: assert(p), trace(current_def(X)).
Start debugging - creep mode
  (1) 0  CALL   current_def(X) (dbg)?- creep
  (2) 1  CALL   current_def(X, new_module) (dbg)?- no debug

X = p / 0     More? (;)

no (more) solution.
```

When a call to a tool interface is compiled, an additional module argument is supplied by the compiler. Therefore, the compiler must be informed that a predicate is a tool interface before any call is compiled to it. An error is raised when this rule is not respected.

However, it is sometimes not convenient to have the tool definition before compiling any call to it (e.g. when a tool is actually defined in a library that is not yet compiled). This can be solved by using the declaration predicate **tool/1**. This predicate only informs the system that the predicate specified in its argument is (or will be) a tool interface.

Note that when changing the visibility of a predicate, tools may become visible in modules that already compiled a call to that predicate but not as a tool call. This is for example the case when abolishing a local (non tool) predicate making therefore a global tool visible or when exporting a tool *after* other modules have imported the predicate. Such visibility changes will raise errors ("inconsistent procedure redefinition").

9.6.5 System Tools

Many of the system built-in predicates are in fact tools, e.g. **read/1**, **write/1**, **record/2**, **compile/1**, etc. All predicates which handle modular items must be tools so that they know from which module they have been called. In case that the built-in predicate has to be executed in a different module (this is very often the case when writing user tool predicates), the **@/2** construct must be used. It simulates a call of the tool predicate from within a different module context:

```
current_predicate(P) @ SomeModule
```

9.7 Libraries

A library is usually implemented in a module so that its implementation details are hidden to the outside. Its access is done through its module interface.

The library is made available using **use_module(library(...))** or **lib/1** which will search for the file (specified as argument) in the library path (see **get_flag/2**), compile it and make its interface available in the caller module.
Name clashes are reported when two predicates of the same name exist in two different imported libraries. This event is a warning; the predicate defined in the last imported library is imported. It is possible to prevent the name clash by explicitly importing the predicate that causes the name clash with **import_from/2** and then import the library, or to cancel the link using **abolish/1**.
A useful predicate when writing a library that must redefine an existing predicate using its previous definition is **:/2**. This predicate calls the goal in its first argument like with **call/1** except that the predicate called (it must be global or exported) is the one **defined** in the module specified in the second argument. For example, to redefine the built-in **at/2** switching the 2 arguments we can do:

```
at(Pointer, Stream) :-
     sepia_kernel:at(Stream, Pointer).
```

Note that the caller module of the goal argument will be the same as the caller module of **:/2**. The purpose of **:/2**. is to specify another predicate than the visible one whereas the purpose of **@/2** is to call the goal in its first argument with the second argument as caller module.

9.8 Other Modular Items

Operators, records, macros, arrays and global variables have only two visibility levels: global (visible from all modules that do not define a local one) or local (only visible from its definition module).
Operators can be defined globally or locally using **global/1** or **local/1** respectively, e.g.

```
:- local op(500, xfy, before).
:- global op(500, xfy, before).
:- op(500, xfy, before).          % defaults to local
```

A local operator only affects reading and writing in the module where it is defined. A local operator always hides a global one. Moreover, a local operator of precedence 0 can be used to hide a global operator definition of the same associativity class. It can be removed (and the global one made visible again) with **abolish_op/2**.
Records are global by default. A local record is declared with

```
:- local record(key1).
```

All predicates that access records work on the visible record. A local record can be abolished (and therefore a global be made visible if there is one) with **abolish_record/1**. Please note that **erase_all/1** is different to **abolish_record/1** since it only removes all the values recorded under a key but does not remove the local declaration of that record (i.e. an existing global record is not made visible).
Non-logical variables and arrays are global or local depending on whether they are defined using **local/1** like

```
:- local variable(name).
:- local array(arr(3,4)).
```

or **global/1** like

```
:- global variable(name).
:- global array(arr(3,4)).
```

All built-ins that access arrays or global variables access the visible one. Any array can be removed using **erase_array/1**.

Macros are local by default, they can be made global if the flag `global` is specified in the flag list.

Character classes and syntax options are only local, they cannot be made global. If two or modules have to share them, they have to be defined in a module interface.

Metaterm attributes are accessed by a name which is usually the name of a module. Attribute names are global, they cannot be made local.

9.9 Privacy

In an incremental Prolog system like ECL^{i}PS^e the privacy requirement can sometimes be obstructive with regard to program development. The internals of modules which are developed incrementally should be accessible from outside, e.g. for debugging purposes. Therefore, the module system of ECL^{i}PS^e provides therefore a way of switching on the privacy of a module, called *locking* a module. Once an application or a part of it is stable, the respective modules will be locked and gain complete privacy.

Once a module has been locked (with **lock/1,2**), it is not possible to access the information contained in it from the outside of the locked module. Its contents is hidden to the outside. However, it can be normally used through its interface. Only modular items which are global or exported can be accessed.

Moreover, if a tool (no matter whether it belongs to a locked module or not) is not called through its interface, it will not be possible to execute any predicate in the module space of the module passed as argument if that module is locked (an error will be raised in this case). This is necessary to prevent calling a tool body with a locked module *M* as argument from another module than the module *M* itself, otherwise privacy would not be respected. However, when the tool is called through its interface, the system certifies the "authenticity" of the module argument and the tool body will be able to use the module argument safely.

Note also that predicates like **listing/0** work only on the predicates defined in the caller module (not on the visible ones). It means that the tool body of **listing/0** must be called to list procedure defined in other modules. If that module is locked, the access will be refused. Therefore, the only way to list procedures defined in a locked module is to call the listing predicate (through its interface procedure) *from* that module.

ECL^{i}PS^e provides two locking mechanisms: one which is not reversible (**lock/1**), the other one which allows the module to be unlocked providing a given key (**lock/2** and **unlock/2**).

9.10 Dynamic Procedures

Most built-ins that access dynamic predicates work on the predicates *defined in* the caller module. Therefore, if the source module of a dynamic procedure is locked, but the procedure has to be modified from outside, some interface must be provided by the locked module. For instance :

```
:- module_interface(mod).
:- global assert_in_mod/1.
:- begin_module(mod).
assert_in_mod(X) :-
        assert(X). % mod is the caller module
```

Then, calling `assert_in_mod(P)` from any module will assert `P` in `mod`.

9.11 Event handlers

The caller module of event handlers is the module from where the handler has been set (using **set_error_handler/2** or **set_interrupt_handler/2**). This allows to set event handlers that are local to a module and still be able to call it no matter from what module the event occurred. The caller module of the culprit of an error can however be obtained in the third argument of the error handler (if it is provided).

9.12 Debugger

If it is possible to trace the execution of a predicate in a locked module with the debugger, privacy might not be respected. Therefore, the predicates in a locked module must be set to skipped if they have to be hidden (they are not skipped by default). It is not allowed to reset the skipped flag of procedures defined in a locked module.
Note also that the debugger provides an option (`m`) to output the definition and caller module of a goal in the trace line.

Chapter 10

Arithmetic

10.1 Built-Ins to Evaluate Arithmetic Expressions

Unlike other languages, Prolog usually interprets an arithmetic expression like **3 + 4** as a compound term with functor + and two arguments. Therefore a query like **3 + 4 = 7** fails because a compound term does not unify with a number. The evaluation of an arithmetic expression has to be explicitly requested by using one of the built-ins described below.
The basic predicate for evaluating an arithmetic expression is **is/2**. Apart from that only the 6 arithmetic comparison predicates evaluate arithmetic expressions automatically.

Result is Expression **Expression** is a valid arithmetic expression and **Result** is an uninstantiated variable or a number. The system evaluates **Expression** which yields a numeric result. This result is then unified with **Result**. An error occurs if **Expression** is not a valid arithmetic expression or if the evaluated value and **Result** are of different types.

Expr1 < Expr2 succeeds if (after evaluation and type coercion) Expr1 is less than Expr2.

Expr1 >= Expr2 succeeds if (after evaluation and type coercion) Expr1 is greater or equal to Expr2.

Expr1 > Expr2 succeeds if (after evaluation and type coercion) Expr1 is greater than Expr2.

Expr1 =< Expr2 succeeds if (after evaluation and type coercion) Expr1 is less or equal to Expr2.

Expr1 =:= Expr2 succeeds if (after evaluation and type coercion) Expr1 is equal to Expr2.

succeeds if (after evaluation and type coercion) Expr1 is not equal to Expr2.

10.2 Numeric Types and Type Conversions

ECL^{i}PS^e distinguishes three types of numbers: **integers**, **rationals** and **floats**.

10.2.1 Integers

The magnitude of integers is only limited by your available memory. However, integers that fit into the word size of your computer are represented more efficiently (this distinction is invisible to the user). Integers are written in decimal notation or in base notation, e.g.:

Expr1 =\= Expr2 `0   3   -5   1024   16'f3ae   0'a   15511210043330985984000000`

10.2.2 Rationals

Rational numbers implement the corresponding mathematical domain, i.e. ratios of two integers (numerator and denominator). ECL^{i}PS^e represents rationals in a canonical form where the greatest common divisor of numerator and denominator is 1 and the denominator is positive. Rational constants are written as numerator and denominator separated by an underscore[1], e.g.

`1_3   -30517578125_32768   0_1`

Rational arithmetic is arbitrarily precise. When the global flag `prefer_rationals` is set, the system uses rational arithmetic wherever possible. In particular, dividing two integers then yields a precise rational rather than a float result.

10.2.3 Floating Point Numbers

Floating point numbers conceptually correspond to the mathematical domain of real numbers, but are not precisely represented. Floats are written with decimal point and/or an exponent, e.g.

`0.0   3.141592653589793   6.02e23   -35e-12   -1.0Inf`

ECL^{i}PS^e can handle both single and double precision floats. Which precision is used depends on the setting of the global flag `float_precision`[2]. The default is double precision.

10.2.4 Type Conversions

Note that numbers of different types never unify, e.g. 3, 3_1 and 3.0 are all different. Use the arithmetic comparison predicates when you want to compare numeric values. When numbers of different types occur as arguments of an arithmetic operation or comparison, the types are first made equal by converting to the more general of the two types, i.e. the rightmost one in the sequence

integer → rational → single float → double float

The operation or comparison is then carried out with this type and the result is of this type as well, unless otherwise specified. Beware of the potential loss of precision in the rational → float conversion! Note that the system never does automatic conversions in the opposite direction. Such conversion must be programmed explicitly using the **fix**, **rational** and **float** functions.

10.3 Arithmetic Functions

10.3.1 Predefined Arithmetic Functions

The following predefined arithmetic functions are available. E, E1 and E2 stand for arbitrary arithmetic expressions.

---

[1] When the library `rationals` is loaded, rationals are printed and accepted in the more familiar form of e.g. 1/3. For compatibility reasons, this syntax is not the default one.

[2] Since single and double floats do not unify, the `float_precision` flag should not be switched during execution to avoid confusing behaviour.

Function	Description	Argument Types	Result Type
+ E	unary plus	number	number
− E	unary minus	number	number
abs(E)	absolute value	number	number
sgn(E)	sign value	number	integer
floor(E)	round down to integral value	number	number
round(E)	round to nearest integral value	number	number
E1 + E2	addition	number × number	number
E1 − E2	subtraction	number × number	number
E1 * E2	multiplication	number × number	number
E1 / E2	division	number × number	see below
E1 // E2	integer division	integer × integer	integer
E1 mod E2	modulus operation	integer × integer	integer
E1 ^ E2	power operation	number × number	number
min(E1,E2)	minimum of 2 values	number × number	number
max(E1,E2)	maximum of 2 values	number × number	number
\ E	bitwise complement	integer	integer
E1 /\ E2	bitwise conjunction	integer × integer	integer
E1 \/ E2	bitwise disjunction	integer × integer	integer
xor(E1,E2)	bitwise exclusive disjunction	integer × integer	integer
E1 >> E2	shift E1 right by E2 bits	integer × integer	integer
E1 << E2	shift E1 left by E2 bits	integer × integer	integer
sin(E)	trigonometric function	number	float
cos(E)	trigonometric function	number	float
tan(E)	trigonometric function	number	float
asin(E)	trigonometric function	number	float
acos(E)	trigonometric function	number	float
atan(E)	trigonometric function	number	float
exp(E)	exponential function e^x	number	float
ln(E)	natural logarithm	number	float
sqrt(E)	square root	number	float
pi	the constant pi = 3.1415926...	—	float
e	the constant e = 2.7182818...	—	float
fix(E)	convert to integer (truncate)	number	integer
float(E)	convert to float	number	float
rational(E)	convert to rational	number	rational
numerator(E)	extract numerator of a rational	integer or rational	integer
denominator(E)	extract denominator of a rational	integer or rational	integer
sum(L)	sum up list elements	list	number
eval(E)	evaluate runtime expression	term	number

Argument types other than specified yield a type error. As an argument type, *number* stands for *integer, rational or float* with the type conversions as specified above. As a result type, *number* stands for *the more general of the argument types*. *float* stands for *single or double precision float*, depending on the value of the `float_precision` flag. The division operator / yields either a rational or a float result, depending on the value of the global flag `prefer_rationals`. The same is true for the result of ^ if an integer is raised to a negative integral power.

The relation between integer division // and modulus operation **mod** is as follows:

```
X =:= (X mod Y) + (X // Y) * Y
```

10.3.2 Evaluation Mechanism

An arithmetic expression is a Prolog term that is made up of variables, numbers, atoms and compound terms, e.g.

```
3 * 1.5 + Y / sqrt(pi)
```

Compound terms are evaluated by first evaluating their arguments and then calling the corresponding evaluation predicate. The evaluation predicate associated with a compound term **func(a_1,..,a_n)** is the predicate **func/(n+1)**. It receives a_1,..,a_n as its first n arguments and returns a numeric result as its last argument. This result is then used in the arithmetic computation. For instance, the expression above would be evaluated by the goal sequence

```
*(3,1.5,T1), sqrt(3.14159,T2), /(Y,T2,T3), +(T1,T3,T4)
```

where T*i* are auxiliary variables created by the system to hold intermediate results.
Although this evaluation mechanism is usually transparent to the user, it becomes visible when errors occur, when subgoals are delayed, or when inline-expanded code is traced.

10.3.3 User Defined Arithmetic Functions

This evaluation mechanism outlined above is not restricted to the predefined arithmetic functions shown in the table. In fact it works for all atoms and compound terms. It is therefore possible to define a new arithmetic operation by just defining an evaluating predicate:

```
[eclipse 1]: [user].
 :- op(200, yf, !).             % let's have some syntaxtic sugar
 !(N, F) :- fac(N, 1, F).
 fac(0, F0, F) :- !, F=F0.
 fac(N, F0, F) :- N1 is N-1, F1 is F0*N, fac(N1, F1, F).
 user       compiled traceable 504 bytes in 0.00 seconds

yes.
[eclipse 2]: X is 23!.       % calls !/2

X = 25852016738884976640000
yes.
```

Note that this mechanism is not only useful for user-defined predicates, but can also be used to call ECL^{i}PS^e built-ins inside arithmetic expressions, eg.

```
T is cputime - T0.
L is string_length("abcde") - 1.
```

which call **cputime/1** and **string_length/2** respectively. Any predicate that returns a number as its last argument can be used in a similar manner.

However there is a difference compared to the evaluation of the predefined arithmetic functions (as listed in the table above): The arguments of the user-defined arithmetic expression are *not evaluated* but passed unchanged to the evaluating predicate. E.g. the expression `twice(3+4)` is transformed into the goal `twice(3+4, Result)` rather than `twice(7, Result)`. This makes sense because otherwise it would not be possible to pass any compound term to the predicate. If evaluation is wanted, the user-defined predicate can explicitly call **is/2** or use eval/1.

10.3.4 Runtime Expressions

In order to enable efficient compilation of arithmetic expressions, ECL^{i}PS^e requires that variables in compiled arithmetic expressions must be bound to numbers at runtime, not symbolic expressions. E.g. in the following code **p/1** will only work when called with a numerical argument, else it will raise error 24:

```
p(Number) :- Res is 1 + Number, ...
```

To make it work even when the argument gets bound to a symbolic expression at runtime, use eval/1 as in the following example:

```
p(Expr) :- Res is 1 + eval(Expr), ...
```

If the expression is the only argument of is/2, the eval/1 may be omitted.

10.4 Low Level Arithmetic Builtins

The low level builtins (like **+/3**, **sin/2** etc.) which are used to evaluate the predefined arithmetic functions can also be called directly, but this is not recommended for portability reasons. Moreover, there is no need to use them directly since the ECL^{i}PS^e compiler will transform all arithmetic expressions into calls to the corresponding low level builtins [3].

10.5 The Multi-Directional Predicates plus/3 and times/3

A drawback of arithmetic using **is/2** is that the right hand side must be fully instantiated at evaluation time. Often it is desirable to have predicates that define true logic relationships between their arguments like "Z is the sum of X and Y". For integer addition and multiplication this is provided as:

plus(X, Y, Z) True if the sum of X and Y is Z. At most one of X, Y, Z can be a variable.

times(X, Y, Z) True if the product of X and Y is Z. At most one of X, Y, Z can be a variable.

They work only with integer arguments but any single argument can be a variable which is then instantiated so that the relation holds. If more than one argument is uninstantiated, an instantiation fault is produced.
Note that if one of the first two arguments is a variable, a solution doesn't necessarily exist. For example, the following goal has no integer solution :

---

[3] Note that this optimisation is only done in `:- nodbgcomp` mode

```
[eclipse 1]: times(2, X, 3).

no (more) solution.
```

Since any one of the arguments of these two predicates can be a variable, it does not make much sense to use them in arithmetic expressions where always the first arguments are taken as input and the last one as output. Their most convenient use is in the coroutining mode, where they delay until their arguments are sufficiently instantiated. This is described in section 10.6.

10.6 Arithmetic and Coroutining

The behaviour of the arithmetic predicates is slightly different when the system operates in coroutining mode. This applies to all the predicates that evaluate their arguments (ie. **is/2** and the arithmetic comparisons) as well as to **plus/3**, **times/3** and the low level evaluation predicates like **+/3** etc. Every condition which yields an instantiation fault in non-coroutining mode now causes the predicates to delay until their arguments are sufficiently instantiated. For example:

```
X is 1 + 3 * Y.
```

where Y is an uninstantiated variable will generate an instantiation fault in non-coroutining mode. In coroutining mode the goal will delay and will be woken as soon as Y is bound. Refer to page 156 on how to switch on the coroutining mode.

Note that the predicates that delay are the low level evaluation predicates. In the above example the free variable Y causes the multiplication to delay and the unavailable result of the multiplication causes the addition to delay:

```
[eclipse 1]: X is 1 + 3 * Y.

X = X
Y = Y

Delayed goals:
        *(3, Y, _d142)
        +(1, _d142, X)
yes.
```

The comparison predicates behave in a similar way:

```
[eclipse 1]: 3 > Y + 1.

Y = Y

Delayed goals:
       +(Y, 1, _d138)
       3 > _d138
yes.
```

Chapter 11

Arrays and Global Variables

11.1 Introduction

This chapter describes the features provided by ECL^{i}PS^e for the declaration and use of arrays and non-logical variables. These provide a mechanism to maintain information across backtracking, in a more procedural programming manner.
Arrays and non-logical variables are handled by a single set of builtins, where a non-logical variable is just considered as an array with no dimensions. Builtins that accept array specifications in the form *Name/Arity* also accept *Name/0* or just *Name* to denote a non-logical variable.

11.2 Non-logical Variables

Non-logical variables in ECL^{i}PS^e are a means of storing a copy of a Prolog term under a name (an atom). The atom is the *name* and the associated term is the *value* of the non-logical variable. This term may be of any form, whether an integer or a huge compound structure. Note that the associated term is being copied and so if it is not ground, the retrieved term is not strictly identical to the stored one but is a *variant* of it[1]. There are two fundamental operations that can be performed on a non-logical variable: setting the variable (giving it a value), and referencing the variable (finding the value currently associated with it).
The value of a non-logical variable is set using the **setval/2** predicate. This has the format

setval(Name, Value)

For instance, the goal

```
setval(firm, 3)
```

gives the non-logical variable *firm* the value 3. The value of a non-logical variable is retrieved using the **getval/2** predicate. The goal

```
getval(firm, X)
```

will unify *X* to the value of the non-logical variable *firm*, which has been previously set by **setval/2**. If no value has been previously set, the call raises an exception. If the value of a non-logical variable is an integer, the predicates **incval/1** and **decval/1** may be used to increment

---

[1] Though this feature could be used to make a copy of a term with new variables, it is cleaner and more efficient to use **copy_term/2** for that purpose

and decrement the value of the variable, respectively. The predicates **incval/1** and **decval/1** may be used e.g. in a failure-driven loop to provide an incremental count across failures as in the example:

```
count_solutions(Goal, _) :-
        setval(count, 0),
        call(Goal),
        incval(count),
        fail.
count_solutions(_, N) :-
        getval(count, N).
```

However, code like this should be used carefully. Apart from being a non-logical feature, it also causes the code to be not reentrant. I.e. if count_solutions/2 would be called recursively from inside `Goal`, this would smash the counter and yield incorrect results[2].

The visibility of a non-logical variable can be controlled using **global/1** or **local/1** declarations. E.g. in the above example one could use one of

```
:- local variable(count).
:- global variable(count).
```

If local, the variable is only accessible from within the module where it was declared.

11.3 Non-logical Arrays

Non-logical arrays are a generalisation of the non-logical variable, capable of storing multiple values. Arrays have to be declared in advance. They have a fixed number of dimensions and a fixed size in each dimension. Arrays in ECL^{i}PS^e are managed solely by special predicates. In these predicates, arrays are represented by compound terms, e.g. **matrix(5, 8)** where **matrix** is the name of the array, the arity of 2 specifies the number of dimensions, and the integers 5 and 8 specify the size in each dimension. The number of elements this array can hold is thus 5*8 = 40. The elements of this array can be addressed from **matrix(0,0)** up to **matrix(4,7)**.

An array must be explicitly created using a **global/1** or **local/1** declaration, e.g.

```
:- local array(matrix(5, 8)).
:- global array(matrix(5, 8)).
```

If local, the array is only accessible from within the module where it was declared. The declaration will create a two-dimensional, 5-by-8 array with 40 elements matrix(0,0) to matrix(4, 7).

Arrays can be erased using the predicate **erase_array/1**, e.g.

```
erase_array(matrix/2).
```

The value of an element of the array is set using the **setval/2** predicate. The first argument of **setval/2** specifies the element which is to be set, the second specifies the value to assign to it. The goal

```
setval(matrix(3, 2), plato)
```

---

[2] A similar problem can occur when the counter is used by an interrupt handler

sets the value of element (3, 2) of array `matrix` to the atom `plato`. Similarly, values of array elements are retrieved by use of the **getval/2** predicate. The first argument of **getval/2** specifies the element to be referenced, the second is unified with the value of that element. Thus if the value of matrix(3, 2) had been set as above, the goal

```
getval(matrix(3, 2), Val)
```

would unify `Val` with the atom `plato`. Similarly to non-logical variables, the value of integer array elements can be updated using **incval/1** and **decval/1**.

It is possible to declare arrays whose elements are constrained to belong to certain types. This allows ECL^{i}PS^e to increase time and space efficiency of array element manipulation. Such an array is created for instance by the predicate

```
:- local array(primes(100),integer).
```

The second argument specifies the type of the elements of the array. It takes as value an atom from the list `real` (for real numbers), `integer` (for integers), `byte` (an integer modulo 256), or `prolog` (any Prolog term - the resulting array is the same as if no type was specified). When a typed array is created, the value of each element is initialised to zero in the case of `byte`, `integer` and `real`, and to an uninstantiated variable in the case of `prolog`. Whenever a typed array element is set, type checking is carried out.

As an example of the use of a typed array, consider the following goal, which creates a 3-by-3 matrix describing a 90 degree rotation about the x-axis of a Cartesian coordinate system.

```
:- local array(rotate(3, 3), integer).
:- setval(rotate(0, 0), 1),
   setval(rotate(1, 2), -1),
   setval(rotate(2, 1), 1).
```

(The other elements of the above array are automatically initialised to zero).

The predicate **current_array/2** is provided to find the size, type and visibility of defined arrays. of the array and its type to be found:

current_array(Array, Props)

where *Array* is the array specification as in the declaration (but it may be uninstantiated or partially instantiated), and *Props* is a list indicating the array's type and visibility. Non-logical variables are also returned, with *Array* being an atom and their type is `prolog`.

```
[eclipse 1]: local(array(pair(2))),
        setval(count, 3),
        global(array(count(3,4,5), integer)).

yes.
[eclipse 2]: current_array(Array, Props).

Array = pair(2)
Props = [prolog, local]     More? (;)

Array = count
```

```
Props = [prolog, global]     More? (;)

Array = count(3, 4, 5)
Props = [integer, global]     More? (;)

no (more) solution.
[eclipse 3]: current_array(count(X,Y,Z), _).

X = 3
Y = 4
Z = 5
yes.
```

11.4 Global References

Terms stored in non-logical variables and arrays are copies of the **setval/2** arguments, and the terms obtained by **getval/2** are thus not identical to the original terms, in particular their variables are different. Sometimes it is more convenient or even necessary to be able to access the original term with its variables, i.e. to have *global variables* in the meaning of conventional programming languages. A typical example is the use of graphical interface: if we want to modify the value of a Prolog variable through a graphical user interface, this mechanism has to be used because the user interface has no means to access Prolog terms directly. Another use is global state that a set of predicates wants to share without having to pass an argument pair through all the predicate invocations.

ECL^{i}PS^e offers the possibility to store references to general terms and to access them even inside predicates that have no common variables with the predicate that has stored them. They are stored in so-called **references**. For example,

```
:- local reference(p).
```

creates a named reference **p** which can be used to store references to terms. This reference is accessed and modified in the same way as non-logical variables, with **setval/2** and **getval/2**, but the following points are different for references:

- the accessed term is identical to the stored term (with its current substitutions):

  ```
  [eclipse 1]: local reference(a), variable(b).

  yes.
  [eclipse 2]: Term = p(X), setval(a, Term), getval(a, Y), Y == Term.
  X = X
  Y = p(X)
  Term = p(X)
  yes.
  [eclipse 3]: Term = p(X), setval(b, Term), getval(b, Y), Y == Term.

  no (more) solution.
  ```

- the modifications are backtrackable, when the execution fails over the **setval/2** call, the previous value of the global variable is restored

  ```
  [eclipse 4]: setval(a, 1), (setval(a, 2), getval(a, X); getval(a, Y)).
  X = 2
  Y = Y     More? (;)

  X = X
  Y = 1
  ```

- there are no arrays of references, but the same effect can be achieved by storing a structure in a reference and using the structure's arguments. The arguments can then be accessed and modified using **arg/3** and **setarg/3** respectively.

.

There is only a limited number of references available and their use should be considered very carefully. Their misuse can lead to very bad programs which are difficult to understand and difficult to optimize.

Chapter 12

Input and Output

12.1 Streams in ECL^{i}PS^e

To provide input to and output from a Prolog program the ECL^{i}PS^e system can communicate with files in the host machine environment. (The user's terminal is regarded as a file for this purpose.) This is done by opening communication channels, known as *streams*, to the files. The streams may be opened for input only (*read mode*), output only (*write mode*), or for both input and output (*update mode*). Each stream is associated with a file or a virtual file (a pipe or a terminal). The number of files that can be open at one time depends on the operating system limitations, but the number of ECL^{i}PS^e streams is not limited. When the system is entered, a small number of standard streams are available, the exact number depending on the operating system. Until instructed otherwise by the user, the system uses these streams for all input and output.

A *stream* defines a logical I/O channel which is used by built-in predicates to perform input and output on. A stream is identified by its name, which is an atom. Each *logical* stream is assigned to a *physical* stream. The *physical* streams, denoted by small integers, are directly connected to files, pipes etc. (the physical stream, however, has no relation to the UNIX file descriptor). Most of the built-in predicates that handle streams explicitly have the stream argument at the first position, e.g. *write(Stream, Term)*.

NOTE: Although physical streams can be directly accessed and used, the ECL^{i}PS^e programs should not contain explicit references to physical streams, because the correspondence of logical and physical streams may not be maintained in various releases. In future ECL^{i}PS^e versions it might even not be possible to access physical streams directly.

It is possible to find the physical stream to which a logical stream is assigned. This is done by a call of the predicate **get_stream/2**:

```
get_stream(Logical_Stream, Stream)
```

This call will return in *Stream* the physical stream to which the `Logical_Stream` is assigned, but it will also succeed if *Stream* is another logical stream associated to the same physical stream as *Logical_Stream*.

New physical streams are created by predicates **open/3**, **pipe/2** (see also section 12.3). These predicates return the newly created stream(s) in their argument(s). If this argument is a free variable, the number of the new physical stream is returned, for instance

```
[eclipse 1]: open(new_file, write, Stream).
```

```
Stream = 6
yes.
```

If the stream argument is a stream name, new logical stream with that name is created if necessary, and the new physical stream is associated to it:

```
[eclipse 1]: open(new_file, write, new_stream).

yes.
```

It is also possible to assign a logical stream to an existing physical or logical stream using the predicate **set_stream/2**:

```
set_stream(New_Stream, Existing_Stream)
```

Here the logical stream *New_Stream* has been assigned to the physical stream that is currently associated with *Existing_Stream*. A logical stream is always connected only to a physical stream, it is not possible to associate a logical stream to another logical stream so that changing one would change the other as well:

```
[eclipse 1]: set_stream(a, 0), set_stream(b, a), set_stream(a, 1).

yes.
[eclipse 2]: get_stream(a, A), get_stream(b, B).

A = 1
B = 0
yes.
```

The predicate

```
close(Stream)
```

is used to close a stream. If the specified stream is a logical one, it is closed and if its associated physical stream is still open, it is closed as well. When a physical stream is closed and there are some logical streams associated to it, they are not closed, but they still refer to the closed physical stream and this physical stream cannot be used for another channel until all logical streams associated to it are redirected to other physical streams or closed.
The predicate

```
current_stream(?Stream)
```

can be used to backtrack over all the currently opened streams. A stream's properties can be accessed using

```
get_stream_info(+Stream, +Property, -Value)
```

e.g. its mode, line number, file name etc.

12.2 System Streams

Apart from stream names defined by the user, there are a number of predefined system stream names. At the beginning of a session they are assigned to the process's standard I/O streams, but they can be changed by the user. The system streams are:

input Used by the input predicates that do not have the stream as an explicit argument, e.g. **read/1**.

output Used by the output predicates that do not have the stream as an explicit argument, e.g. **write/1**.

error Output for error messages and warnings and all messages about exceptional states.

null A dummy stream, output to it is discarded, on input it always gives end of file. It can be used to ignore part of a program's output without having to remove the output predicates. Note that this stream is similar to the UNIX file '/dev/null' but works much faster. This stream cannot be redirected to any other physical stream.

In development environments, there are a few more system streams:

toplevel_input The input for the top-level Prolog loop, it reads the user query and the semicolon or newline typed to indicate whether other solutions are required.

toplevel_output The messages from the top-level loop, e.g. the yes/no answer, delayed goals, messages about loaded libraries.

answer_output Outputs the top-level loop answer bindings for the user query.

debug_input Input to the debugger.

debug_output Output from the debugger

Each of the physical *input* streams has its own prompt which is printed on the specified output stream whenever new input is required. The goal

```
set_prompt(InputStream, Prompt, OutputStream)
```

sets the prompt for *InputStream* to be *Prompt* printed on the output stream *OutputStream*. The predicate **get_prompt/3** can be used to query the prompt of an input stream.
Apart from these system streams there are other logical stream identifiers provided by the system:

stdin The UNIX standard input stream when the ECL^{i}PS^e session is started.

stdout The UNIX standard output stream when the ECL^{i}PS^e session is started.

stderr The UNIX standard error output stream when the ECL^{i}PS^e session is started.

user This identifier is provided for compatibility with other Prolog systems and it is identical with **stdin** and **stdout** depending on the context where it is used. When it is not possible for the system to decide whether the input or output stream was meant by **user**, an exception is raised. This stream cannot be redirected to another physical stream.

The streams **stdin**, **stdout** and **stderr** are used to initialize all other system streams and also to reset them when the current system stream is closed. To allow greater flexibility of the system, these streams can be modified by the user.

When an attempt is made to close a system stream, the exception 196 is raised. The default handler for this exception resets all system streams connected with this physical streams to their defaults and then the physical stream is closed.

When `set_stream(Logical_Stream, Stream)` is used to redirect a system stream, *Stream* must have a mode compatible with that of `Logical_Stream`. That is, if *Logical_Stream* specifies a stream with read mode, *Stream* must have either read or update mode; if *Logical_Stream* has write mode, *Stream* must have either write, append or update mode; and if *Logical_Stream* has update mode, *Stream* must have update mode.

12.3 Opening New Streams

A stream is opened for input or output by means of the **open/3** or **open/4** predicate. The goals

```
open(SourceSink, Mode, Stream)
open(SourceSink, Mode, Stream, Options)
```

open a communication channel with *SourceSink*.

If *SourceSink* is an atom or a string, a file is being opened and *SourceSink* takes the form of a file name in the host machine environment. ECL^{i}PS^e uses an operating system independent path name syntax, where the components are separated by forward slashes. The following forms are possible:

- abolute path name, e.g. /usr/peter/prolog/file.pl
- relative to the current directory, e.g. prolog/file.pl
- relative to the own home directory, e.g. ~/prolog/file.pl
- start with an environment variable, e.g. $HOME/prolog/file.pl
- relative to a user's home directory, e.g. ~peter/prolog/file.pl (UNIX only)
- specifying a drive name, e.g. //C/prolog/file.pl (Windows only)

Note that path names usually have to be quoted (in single or double quotes) because they contain non-alphanumeric characters.

If *SourceSink* is of the form `string(InitString)` a pseudo-file in memory is opened, see section 12.5.1.

If *SourceSink* is of the form `queue(InitString)` a pseudo-pipe in memory is opened, see section 12.5.2.

Mode must be one of the atoms `read`, `write`, `append` or `update`, which means that the stream is to be opened for input, output, output at the end of the existing stream, or both input and output, respectively. Opening a file in `write` mode will create it if it does not exist, and erase the previous contents if it does exist. Opening a file in `append` mode will keep the current contents of the file and start writing at its end.

Stream is a logical stream identifier or an uninstantiated variable. If it is uninstantiated, the system will create an identifier. This stream identifier may then be used in predicates which have a named stream as one of their arguments. For example

```
open('foo', update, stream), write(stream, subject)
```

will write the atom *subject* to the file 'foo'. A stream *Stream* opened by the **open/3** predicate may be subsequently closed by the call

```
close(Stream)
```

The predicate **pipe/2** is used like

```
pipe(In, Out)
```

and opens a pipe, i.e. two streams, *In* for reading and *Out* for writing, which are connected together using the *pipe(2)* system call. This mechanism is normally used to communicate with other processes which were forked by the main process.

Sockets streams are opened with the primitives **socket/3** and **accept/3**, more details are in chapter 21.

On most devices, output is buffered, i.e. any output does not appear immediately on the file, pipe or socket, but goes into a buffer first. To make sure the data is actually written to the device, the stream usually has to be flushed using **flush/1**. If this is forgotten, the receiving end of a pipe or socket may hang in a blocking read operation.

12.4 Communication with Streams

The contents of a stream may be interpreted in one of the three basic ways. The first one is to consider it as a sequence of characters, so that the basic unit to be read or written is a character. The second one interprets the stream as a sequence of tokens, thus providing an interface to the Prolog lexical analyzer and the third one is to consider a stream as a sequence of Prolog terms.

12.4.1 Character I/O

The **get/1, 2** and **put/1, 2** predicates corresponds to the first way of looking at streams. The call

```
get(Char)
```

takes the next character from the current input stream and matches it as a single character with Char. Note that a character in ECL^{i}PS^e is represented as an integer corresponding to the ASCII code of the character. If the end of file has been reached then an exception is raised. The call

```
put(Char)
```

puts the char Char on to the current output stream. The predicates

```
get(Stream, Char)
```

and

```
put(Stream, Char)
```

work similarly on the specified stream.

The input and output is normally buffered by ECL^{i}PS^e. To make I/O in *raw mode*, without buffering, the predicates **tyi/1, 2** and **tyo/1, 2** are provided.

12.4.2 Token I/O

The predicate

```
read_token(Token, Class)
```

represents the second way of interpreting stream contents. It reads the next token from the current input stream, unifies it with *Token*, and its token class is unified with *Class*. A token is either a sequence of characters with the same or compatible character class, e.g. ab_1A, then it is a Prolog constant or variable, or a single character, e.g. ')'. The token class represents the type of the token and its special meaning, e.g. `fullstop`, `comma`, `open_par`, etc.

```
read_token(Stream, Token, Class)
```

reads a token from the specified stream. A further, very flexible possibility to read a sequence of characters is provided by the built-ins

```
read_string(Stream, Delimiters, Length, String)
read_string(Delimiters, Length, String)
```

Here, the input is read up to a specified delimiter or up to a specified length, and returned as an ECL^{i}PS^e string.

12.4.3 Term I/O

The **read/1, 2** and **write/1, 2** predicates correspond to the third way of looking at streams. The goal

```
read(Term)
```

reads the next term from the current input stream and unifies it with *Term*. The input term must be followed by a full stop, that is, a '.' character followed by a layout character (tab, space or newline) or by the end of file. If end of file has been reached then an exception is raised, the default handler causes the atom *end_of_file* to be returned. A term may be read from a stream other than the current input stream by the call

```
read(Stream, Term)
```

which reads the term from the named stream. The goal

```
write(Term)
```

writes *Term* to the current output stream. This is done by taking the current operator declarations into account. Output produced by the **write/1, 2** predicate is not (necessarily) in a form suitable for subsequent input to a Prolog program using the **read/1** predicate, for this purpose **writeq/1, 2** is to be used. The goal

```
write(Stream, Term)
```

writes *Term* to the named output stream. The predicate

```
display(Term)
```

outputs the *Term* on the current output stream in the functor syntax, ignoring possible operator declarations. The predicate

```
readvar(Stream, Term, VarList)
```

can be used to read a term from the specified stream and obtain the list of variable names contained in the *Term*. *VarList* is a list of pairs `[VarName|Var]` where *VarName* is the atom corresponding to the variable name and *Var* is the corresponding variable.
When the flag `variable_names` is switched off, the output predicates are not able to write free variables in their source form, i.e. with the correct variable names. Then the variables are output in the form

```
_aN
```

where `a` is a character which depends on the memory area where the variable is located or on its properties: `l` for a local variable, `g` for a global variable or a metaterm. `N` is a number.
It is possible to pass any input stream to the ECL^{i}PS^e compiler using the predicate

```
compile_stream(Stream)
```

and it is of course possible to mix the compilation with other input predicates. If, for example, the file **a.pl** contains the following data

```
p(1).
p(2).
end_of_file.
p(3).
```

it is possible to execute

```
[eclipse 1]: open('a.pl', read, a).

yes.
[eclipse 2]: read(a, X).

X = p(1)
yes.
[eclipse 3]: compile_stream(a).
a.pl    compiled 40 bytes in 0.00 seconds

yes.
[eclipse 4]: read(a, X).

X = p(3)
yes.
[eclipse 5]: p(X).

X = 2
yes.
```

To specify a position in the file to write to or read from, the predicate **seek/2** is provided. The call

```
seek(Stream, Pointer)
```

moves the current position in the file (the 'file pointer') to the offset *Pointer* (a number specifying the length in bytes) from the start of the file. If *Pointer* is the atom *end_of_file* the current position is moved to the end of the file. Hence a file could be open in `append` mode using

```
open(File, update, Stream), seek(Stream, end_of_file)
```

The current position in a file may be found by the predicate **at/2**. The call

```
at(Stream, Pointer)
```

unifies *Pointer* with the current position in the file. The predicate

```
at_eof(Stream)
```

succeeds if the current position in the given stream is at the file end.

12.5 In-memory Streams

There are two kinds of in-memory streams, string streams and queues. String streams behave much like files, they can be read, written, positioned etc, but they are implemented as buffer in memory. Queues are intended mainly for message-passing-style communication between ECL^{i}PS^{e}and a host language, and they are also implemented as memory buffers.

12.5.1 String Streams

In ECL^{i}PS^e it is possible to associate a stream with a Prolog string in its memory, and this string is then used in the same way as a file for the input and output operations. A string stream is opened like a file by the **open/3** predicate call

```
open(string(InitString), Mode, Stream)
```

where *InitString* can be a ECL^{i}PS^e string or a variable and represents the initial contents of the string stream. If a variable is supplied for *InitString*, the initial value of the string stream is the empty string and the variable is bound to this value:

```
[eclipse 1]: open(string(S), update, s).
S = ""
yes.
```

Once a string stream is opened, all predicates using streams can take it as argument and perform I/O on it. In particular the predicates **seek/2** and **at/2** can be used with them.
While writing into a stream changes the stream contents destructively, the initial string that has been opened will never be affected. The new stream contents can be retrieved either by reading from the string stream, or as a whole by using **get_stream_info/3**:

```
[eclipse 1]: S = "abcdef", open(string(S), write, s), write(s, ---).

S = "abcdef"
yes.
[eclipse 2]: get_stream_info(s, name, S).
```

```
S = "---def"
yes.
[eclipse 3]: seek(s, 1), write(s, .), get_stream_info(s, name, S).

S = "-.-def"
yes.
[eclipse 4]: seek(s, end_of_file), write(s, ine),
             get_stream_info(s, name, S).

S = "-.-define"
yes.
```

12.5.2 Queue streams

A queue stream is opened by the **open/3** predicate

```
open(queue(InitString), Mode, Stream)
```

The initial queue contents is *InitString*. It can be seen as a string which gets extended at its end on writing and consumed at its beginning on reading.

```
[eclipse 11]: open(queue(""), update, q), write(q, hello), write(q, " wo").
yes.
[eclipse 12]: read_string(q, " ", _, X).
S = "hello"
yes.
[eclipse 13]: write(q, "rld"), read(q, X).
S = world
yes.
[eclipse 14]: at_eof(q).
yes.
```

It is not allowed to seek on a queue. Therefore, once something is read from a queue, it is no longer accessible. A queue is considered to be at its end-of-file position when it is currently empty, however this is no longer the case when the queue is written again.
A useful feature of queues is that they can raise a synchronous event when data arrives on the empty queue. To create such an event-raising queue, this has to be specified as an option when opening the queue with **open/4**. In the example we have chosen the same name for the stream and for the event, which is not necessary but convenient when the same handler is going to be used for different queues:

```
[eclipse 1]: [user].
 handle_queue_event(Q) :-
        read_string(Q, "", _, Data),
        printf("Queue %s received data: %s\n", [Q,Data]).
yes.
[eclipse 2]: set_event_handler(eventq, handle_queue_event/1).
yes.
```

```
[eclipse 3]: open(queue(""), update, eventq, [event(eventq)]).
yes.
[eclipse 4]: write(eventq, hello).
Queue eventq received data: hello
yes.
```

12.6 Modifying the Output

There are several possible ways to modify the standard way of outputting terms in ECL^{i}PS^e.

12.6.1 The printf/2, 3 Predicate

This predicate subsumes all other output predicates, and it also offers formatted printing. Its syntax is similar to the C printf(3) function, but it also has further Prolog-specific options. For example, the sequence

```
write('The result is '),
writeq(T),
write(', which is '),
write(P),
write('% better than '),
write(R),
nl,
flush(output).
```

can be written with

```
printf("The result is %q, which is %d%% better than %w\n%b", [T, P, R]).
```

In the **printf/2,3** predicate, several options for printing Prolog terms can be specified by using the following option characters in the **%w** format string:

O ignore operator declarations

D disregard depth limit for nested terms

. print lists as ./2 structures

Q print quotes around functors when needed

v print variables as unique numbers, e.g. _g123

V print variables as names and numbers, e.g. X_g123

P use portray/1,2 if defined

U use portray/1,2 even for variables

m print metaterm attributes using user-defined handlers

M print metaterm attributes in a standard form

G print term as a goal, i.e. apply goal write macros

T don't apply write macro transformations

A depth limit can be sepcified for the printed term by giving an integer immediately after the % in the format string.
Using those options, the other I/O predicates can be defined in terms of **printf/2** as follows:

```
write(X)           :- printf("%w",     [X]).
writeq(X)          :- printf("%QDTMvw", [X]).
print(X)           :- printf("%Pw",    [X]).
display(X)         :- printf("%0.w",   [X]).
write_canonical(X) :- printf("%0.QDTMvw", [X]).
```

12.6.2 The output_mode flag

The flag **output_mode**, set by **set_flag/2**, is a string of control characters (as recognised by the **%w** format of **printf/3**). It is used to specify the format in which the system prints terms

- in the debugger (the **o** command allows some modifications of the **output_mode** flag from within the debugger)
- when printing the answer bindings in the top-level loop

For example, calling **set_flag(output_mode, "O.P")** will cause the debugger to write the traced goals without operators, with dot notation for lists and using the user-defined predicate **portray/1, 2**. The default value is **"QPm"**.

12.6.3 The syntax_option flag

When

```
:- set_flag(syntax_option, '$VAR').
```

is set, terms of the form '$VAR'(N) are printed in a special way by all the predicates that obey operator declarations (i.e. write, writeq, print and partly printf). '$VAR'(0) is printed as A, '$VAR'(25) as Z, '$VAR'(26) as A1 and so on. When the argument is an atom or a string, just this argument is printed. This option is also used for Quintus compatibility mode.

12.6.4 The print/1, 2 Predicate

When **print/1, 2** is used to print a term, the user-definable predicate **portray/1, 2** is called on all its nonvariable subterms and if it succeeds, it is assumed that it has printed the term, otherwise the term is printed in the standard way. The **portray/1, 2** predicate is also invoked by **printf/2, 3** when the option **P** is used. Note that **portray/1, 2** is also invoked to print metaterms, but it is not invoked for variable subterms, unless the option **U** in **printf/2, 3** or in the **output_mode** flag is used.

Chapter 13

ECL^{i}PS^e Macros

13.1 Introduction

ECL^{i}PS^e provides a very general mechanism to perform macro expansion of Prolog terms. Macro expansion can be performed in two situations:

read macros they are applied just after a Prolog term has been read by the ECL^{i}PS^e parser, i.e. during compilation or in a read predicate

write macros they are applied just before a Prolog term is printed by one of the output predicates

Macros are attached to classes of terms specified by their functors or by their type. Macros obey the module system's visibility rules. They may be either locally (default) or globally visible. The macro expansion is performed by a user-defined Prolog predicate.

13.2 Using the macros

The following built-ins control macro expansion:

define_macro(+TermClass, +TransPred, +Options) define a macro for the given *TermClass*. The transformation will be performed by the predicate *TransPred*.

erase_macro(+TermClass) erase a currently defined macro for *TermClass*. This can only be done in the module where the definition was made.

current_macro(?TermClass, ?TransPred, ?Options, ?Module) retrieve information about currently defined visible macros.

Macros are selectively applied only to terms of the specified class. *TermClass* can take two forms:

Name/Arity transform all terms with the specified functor

type(Type) transform all terms of the specified type, where Type is one of `compound`, `string`, `integer`, `rational`, `real`, `atom`, `goal`[1].

---

[1] type(goal) stands for suspensions.

The +TransPred argument specifies the predicate that will perform the transformation. It has to be of arity 2 or 3 and should have the form:

```
trans_function(OldTerm, NewTerm [, Module]) :- ... .
```

At transformation time, the system will call *TransPred* in the module where **define_macro/3** was invoked. The term to transform is passed as the first argument, the second is a free variable which the transformation predicate should bind to the transformed term, and the optional third argument is the module where the term is read or written.
Options is a list which may be empty (in this case the macro defaults to a local read term macro) or contain specifications from the following categories:

- visibility

  local: The transformation is only visible in this module (default).

  global: The transformation is globally visible.

- mode

  read: This is a read macro and shall be applied after reading a term (default).

  write: This is a write macro and shall be applied before printing a term.

- type

  term: Transform all terms (default).

  clause: Transform only if the term is a program clause, i.e. inside **compile/1**, **assert/1** etc. Write macros are applied using the 'C' option in the **printf/2** predicate.

  goal: Transform only if the term is a subgoal in the body of a program clause. Write macros are applied using the 'G' option in the **printf/2** predicate.

- additional specification

  protect_arg: Disable transformation of subterms (optional).

  top_only: Consider only the whole term, not subterms (optional).

Here is an example of a conditional read macro:

```
[eclipse 1]: [user].
 trans_a(a(X,Y), b(Y)) :-    % transform a/2 into b/1,
        number(X),           % but only under these
        X > 0.               % conditions

:- define_macro(a/2, trans_a/2, []).
  user       compiled traceable 204 bytes in 0.00 seconds

yes.
[eclipse 2]: read(X).
        a(1, hello).

X = b(hello)                 % transformed
```

```
yes.
[eclipse 3]: read(X).
        a(-1, bye).

X = a(-1, bye)              % not transformed
yes.
```

If the transformation function fails, the term is not transformed. Thus, **a(1, zzz)** is transformed into **b(zzz)** but **a(-1, zzz)** is not transformed. The arguments are transformed bottom-up. It is possible to protect the subterms of a transformed term by specifying the flag `protect_arg`. A term can be protected against transformation by quoting it with the "protecting functor" (by default it is **no_macro_expansion/1**):

```
[eclipse 4]: read(X).
        a(1, no_macro_expansion(a(1, zzz))).
X = b(a(1, zzz)).
```

Note that the protecting functor is itself defined as a macro:

```
trprotect(no_macro_expansion(X), X).
:- define_macro(no_macro_expansion/1, trprotect/2, [global, protect_arg]).
```

A macro is by default only visible in the module where it has been defined. When it is defined inside a module interface, then it is copied to all other modules that contain a **use_module/1** for this module. The transformation function should be exported in this case and be defined in the module interface as well.

A macro can also be made visible in all modules by specifying the `global` option in the option list. As usual, local definitions hide global ones. The global flag **macro_expansion** can be used to disable macro expansion globally, e.g. for debugging purposes. Use `set_flag(macro_expansion, off)` to do so.

The next example shows the use of a type macro. Suppose we want to represent integers as s/1 terms:

```
[eclipse 1]: [user].
 tr_int(0, 0).
 tr_int(N, s(S)) :- N > 0, N1 is N-1, tr_int(N1, S).
 :- define_macro(type(integer), tr_int/2, []).

yes.
[eclipse 2]: read(X).
        3.

X = s(s(s(0)))
yes.
```

When we want to convert the s/1 terms back to normal integers so that they are printed in the familiar form, we can use a write macro. Note that we first erase the read macro for integers, otherwise we would get unexpected effects since all integers occurring in the definition of tr_s/2 would turn into s/1 structures:

```
[eclipse 3]: erase_macro(type(integer)).

yes.
[eclipse 4]: [user].
 tr_s(0, 0).
 tr_s(s(S), N) :- tr_s(S, N1), N is N1+1.
 :- define_macro(s/1, tr_s/2, [write]).

yes.
[eclipse 2]: write(s(s(s(0)))).
3
yes.
```

13.3 Definite Clause Grammars — DCGs

Grammar rules are described in many standard Prolog texts ([2]). In ECL^{i}PS^e they are provided by a predefined global[2] macro for `-->/2`. When the parser reads a clause whose main functor is `-->/2`, it transforms it according to the standard rules. The syntax for DCGs is as follows :

```
grammar_rule --> grammar_head, ['-->'], grammar_body.

grammar_head --> non_terminal.
grammar_head --> non_terminal, [','], terminal.

grammar_body --> grammar_body, [','], grammar_body.
grammar_body --> grammar_body, [';'], grammar_body.
grammar_body --> grammar_body_item.

grammar_body_item --> ['!'].
grammar_body_item --> ['{'], Prolog_goals, ['}'].
grammar_body_item --> non_terminal.
grammar_body_item --> terminal.
```

The non-terminals are any valid prolog term (other than a variable, a number, or a string), the terminals are prolog terms between square brackets. Every term is transformed, unless it is enclosed in curly brackets. The or (`;/2` or `|/2`), the cut (`!/0`), the condition (`->/1`) do not need to be enclosed in curly brackets.

The grammar can be accessed with the built-ins **phrase/2** and `phrase/3`. The first argument of **phrase/2** is the name of the grammar to be used, the second argument one is a list containing the input to be parsed. If the parsing is successful the built-in will succeed. For instance with the grammar

```
a --> [] | [z], a.
```

`phrase(a, X)` will give on backtracking : `X = [z] ; X = [z, z] ; X = [z, z, z] ; ....`

---

[2] So that the user can redefine it with a local one.

13.3.1 Simple DCG example

The following example illustrates a simple grammar declared using the DCGs.

```
sentence --> imperative, noun_phrase(Number), verb_phrase(Number).

imperative, [you] --> [].
imperative --> [].

noun_phrase(Number) --> determiner, noun(Number).
noun_phrase(Number) --> pronom(Number).

verb_phrase(Number) --> verb(Number).
verb_phrase(Number) --> verb(Number), noun_phrase(_).

determiner --> [the].

noun(singular) --> [man].
noun(singular) --> [apple].
noun(plural) --> [men].
noun(plural) --> [apples].

verb(singular) --> [eats].
verb(singular) --> [sings].
verb(plural) --> [eat].
verb(plural) --> [sing].

pronom(plural) --> [you].
```

The above grammar may be successfully parsed using **phrase/2**. If the predicate succeeds then the input has been parsed successfully.

```
[eclipse 1]: phrase(sentence, [the,man,eats,the,apple]).

yes.
[eclipse 2]: phrase(sentence, [the,men,eat]).

yes.
[eclipse 3]: phrase(sentence, [the,men,eats]).

no.
[eclipse 4]: phrase(sentence, [eat,the,apples]).

yes.
[eclipse 5]: phrase(sentence, [you,eat,the,man]).

yes.
```

The predicate **phrase/3** may be used to return the point at which parsing of a grammar fails — if the returned list is empty then the input has been successfully parsed.

```
[eclipse 1]: phrase(sentence, [the,man,eats,something,nasty],X).

X = [something, nasty]     More? (;)

no (more) solution.
[eclipse 2]: phrase(sentence, [eat,the,apples],X).

X = [the, apples]     More? (;)

X = []     More? (;)

no (more) solution.
[eclipse 3]: phrase(sentence, [hello,there],X).

no (more) solution.
```

13.3.2 Mapping to Prolog Clauses

Grammar rule are translated to Prolog clauses by adding two arguments which represent the input before and after the nonterminal which is represented by the rule. The effect of the transformation can be observed, e.g. by switching on the `all_dynamic` flag so that the compiled clauses can be listed:

```
[eclipse 1]: set_flag(all_dynamic, on), [user].
 p(X) --> q(X).
 p(X) --> [a].
user       compiled traceable 296 bytes in 0.25 seconds

yes.
[eclipse 2]: listing.
p(_g212, _g214, _g216) :-
        q(_g212, _g214, _g216).
p(_g212, _g214, _g216) :-
        _g214 = [a|_g216].

yes.
```

13.3.3 Parsing other Data Structures

DCGs are in principle not limited to the parsing of lists. The predicate **'C'/3** is responsible for reading resp. generating the input tokens. The default definition is

```
'C'([Token|Rest], Token, Rest).
```

The first argument represents the parsing input before consuming Token and Rest is the input after consuming Token. By redefining 'C'/3, it is possible to apply a DCG to other input sources than a list, e.g. to parse directly from an I/O stream:

```
'C'(Stream-Pos0, Token, Stream-Pos1) :-
```

```
        seek(Stream, Pos0),
        read_string(Stream, " ", _, TokenString),
        atom_string(Token, TokenString),
        at(Stream, Pos1).

sentence --> noun, [is], adjective.
noun --> [prolog] ; [lisp].
adjective --> [boring] ; [great].
```

This can then be applied to a string as follows:

```
[eclipse 1]: String = "prolog is great", open(String, string, S),
             phrase(sentence, S-0, S-End).
...
End = 15
yes.
```

Unlike the default definition, this definition of 'C'/3 is not bi-directional. Consequently, the grammar rules using it can only be used for parsing, not for generating sentences.

Note that every grammar rule uses the definition of 'C'/3 which is visible in the module where the grammar rule itself is defined.

Chapter 14

Events and Interrupts

The normal execution of a Prolog program may be interrupted by Events and Interrupts:

Interrupts

Interrupts usually originate from the operating system, e.g. on a Unix host, signals are mapped to ECL^{i}PS^e interrupts.

- they occur asynchronously
- the handler is executed asynchronously in a separate ECL^{i}PS^e engine. This means that
- the handler cannot interact with interrupted execution, except via global variables, files and the like.
- failure of the handler is ignored.
- interrupt handlers are **not available** in embedded ECL^{i}PS^e systems
- the development system catches and handles many operating system signals as interrupts, user abort by typing ^C, data arriving at sockets, memory protection faults, etc.

Events

- they may occur asynchronously (posted by the environment) or synchronously (raised by the program itself).
- they are handled synchronously by a handler goal that is inserted into the resolvent.
- the handler can interact with interrupted execution via global references.
- the handler can cause the interrupted execution to fail or to abort.
- the handler can cause waking of delayed goals.

Errors

Errors are a special case of events. They are raised by built-in predicates (e.g. when the arguments are of the wrong type) and usually pass the culprit goal to the error handler.

14.1 Events

14.1.1 Event Identifiers

Events are identified by names or by small numbers. User defined events always have names, while the ECL^{i}PS^e system uses events with a numerical identifier to raise errors (The error numbers are listed in appendix D).

14.1.2 Handling Events

When an event occurs, a call to the appropriate handler is inserted into the resolvent (the sequence of executing goals). The handler will be executed as soon as possible, which means at the next synchronous point in execution, which is usually just before the next regular predicate is invoked. Note that there are a few built-in predicates that can run for a long time and will not allow handlers to be executed until they return (e.g. read/1, sort/4).
A handler is defined using a call like this

```
my_handler(Event) :-
    <code to deal with Event>

:- set_event_handler(hello, my_handler/1).
```

The handler's first argument is the event identifier, in this case the atom 'hello'.
Note that to ensure the handling of all events, an event handler should not directly fail or raise an exception. This is because the system will backtrack if the handler fails or raise an exception, and any other raised events that has not yet been handled will not be handled, and thus the system will seem to 'forget' about such events. The event handler itself should also be run at the highest priority (1), and if failure is desired, this can be done indirectly through triggering a suspended goal which runs at a lower priority.

14.1.3 Raising Events

Events are normally posted to the ECL^{i}PS^e engine from its software environment, e.g. from a C program using

```
ec_post_event(ec_atom(ec_did("hello",0)));
```

This works both when the foreign code is called from ECL^{i}PS^e or when ECL^{i}PS^e has been called from the foreign code.
It is also possible to post an event from within an interrupt handler by setting the interrupt handler to **event/1**. This is the recommended mechanism to translate an asynchronous interrupt into a synchronous event. E.g.

```
:- set_interrupt_handler(alrm, event/1).
:- set_event_handler(alrm, handle_alarm/1).
```

An event can also be raised by the running program itself, using **event/1**:

```
    ..., event(hello), ...
```

However, this is mainly useful for test purposes, since it is almost the same as calling the handler directly.

14.1.4 Timed Events (after events)

ECL^{i}PS^e provides support for setting up an event which is then triggered after a specified amount of elasped time. Previous to version 4.2, the user can program this functionality using the (now obsolete) low level OS dependent `set_timer/2` primitives. From version 4.2, a higher level interface is provided, allowing for multiple independent timed events to be set up in a standardised way. These events are known as after events, as they are set up so that the event occurs *after* a certain amount of elasped time. They are setup by two predicates:

event_after(+Name, +Time) This sets up an event Name so that the event is raised once after Time seconds of elasped time from when the predicate is executed. Name is an atom and Time is a positive number.

event_after_every(+Name, +Time) This sets up an event Name so that the event is raised *every* Time seconds has elasped from when the predicate is executed.
Once an after event has been set up, it is pending until it is raised. In the case of `event_after_every/2`, the event will always be pending because it is rasied repeatedly. A pending event can be cancelled so that it will not be raised:

cancel_after_event(+Name) This cancels the pending after event Name. If Name is not a pending after event, the predicate fails.

current_after_event(+Name) This tests if Name is a pending after event. The predicate suceeds if it is, fails if it is not.

More details on after events

More precisely, Time is actually the minimum of elasped time before the event is raised. Factors constraining the actual time of raising of the event include the granularity of the system clock, and also that ECL^{i}PS^e must be in a state where it can synchronously process the event – it needs to be where it can make a procedure call.
The event is raised and executed at priority 1, so that it cannot be interrupted by execution of woken goals in the middle of handling the event. However, any other events that are raised during the execution of the event handler will interrupt the execution of the original event handler. It is thus advisable to keep the event handling code as short as possible – if more complex actions needs to be performed, it should be done via the event handler triggering a suspended goal, which will execute at a lower priority than 1.
The after event make use of the `vtalrm` signal where this signal exists, or the `alrm` signal if it doesn't, so elasped time is normally measured in elasped user cpu time, or in real time in the case of `alrm`. Currently, `alrm` is used only on the Windows platform. The user should not make use of these signals for their own purpose if they plan on using the after event mechanism.
The after event mechanism allows multiple events to make use of the timing mechanism independently of each other. However, the same event can be setup multiple times with multiple calls to `event_after/2` and `event_after_every/2`. The `cancel_after_event/1` will cancel all instances of an event.
Using the suspension and event handling mechanisms, the user can cause a goal to be added to the resolvent which would then be executed after a defined elasped time. The goal will be

suspended and attached to a symbolic trigger, which is triggered by the event handler. The goal behaves 'logically', in that if the execution backtracks pass the point in which the suspended goal is created, the goal will disappear from the resolvent as expected and thus not be executed. The event will still be raised, but there will not be a suspended goal to wake up.

The following is an example for waking a goal with a timed event. Once `monitor(X)` is called, the current value of X will be printed every second:

```
:- set_event_handler(monvar, trigger/1).

monitor(Var) :-
     suspend(m(Var), 3, trigger(monvar)),
     event_after_every(monvar, 1).

:- demon m/1.
m(Var) :- writeln(Var).
```

Note the need to declare `m/1` as a demon: otherwise, once `m/1` is woken up once, it will disappear from the resolvent and the next `monvar` event will not have a suspended `m/1` to wake up. Note also that it is necessary to connect the event machanism to the waking mechanism by setting the event handler to **trigger/1**.

14.2 Errors

Errors handling is one particular use of events. The main property of error events is that they have a culprit goal, ie. the goal that detected or caused the error. The error handler obtains that goal as an argument.

The errors that the system raises have numerical identifiers, as documented in appendix D. Whenever an error occurs, the ECL^{i}PS^e system identifies the type of error, and calls the appropriate handler. For each type of error, it is possible for the user to define a separate handler. This definition will replace the default error handling routine for that particular error - all other errors will still be handled by their respective handlers. It is of course possible to associate the same user defined error handler to more than one error type.

When a goal is called and produces an error, execution of the goal is aborted and the appropriate error handler is invoked. This invocation of the error handler is seen as *replacing* the invocation of the erroneous goal:

- If the error handler fails it has the same effect as if the erroneous goal failed.
- If the error handler succeeds, possibly binding some variables, the execution continues at the point behind the call of the erroneous goal.
- If the handler calls **exit_block/1**, it has the same effect as if this was done by the erroneous goal itself.

For errors that are classified as warnings the second point is somewhat different: If the handler succeeds, the goal that raised the warning is allowed to continue execution.

Apart from binding variables in the erroneous goal, error handlers can also leave backtrack points. However, if the error was raised by an external or a builtin that is implemented as an external, these choicepoints are discarded[1].

---

[1] This is necessary because the compiler recognises simple predicates as deterministic at compile time and so

14.2.1 Error Handlers

The predicate **set_error_handler/2** is used to assign a procedure as an error handler. The call

```
set_error_handler(N, PredSpec)
```

sets the error handler for error type *N* to the procedure specified by *PredSpec*, which must be of the form **Name/Arity**.
The corresponding predicate **get_error_handler/3** may be used to identify the current handler for a particular error. The call

```
get_error_handler(N, PredSpec, HomeModule)
```

will, provided *N* is a valid error identifier, unify *PredSpec* with the specification of the current handler for error *N* in the form Name/Arity, and *HomeModule* will be unified with the module where the error handler has been defined. Note that this error handler might not be visible from every module and therefore may not be callable.
To re-install the system's error handler in case the user error handler is no longer needed, **reset_error_handler/1** should be used. **reset_error_handlers/0** resets all error handlers to their default values.
To enable the user to conveniently write predicates with error checking the built-ins

```
error(N, Goal)
error(N, Goal, Module)
```

are provided to raise the corresponding error number *N* with the culprit *Goal*. Inside tool procedures it is usually necessary to use **error/3** in order to pass the caller module to the error handler. Typical error checking code looks like this

```
increment(X, X1) :-
        integer(X) ->
            X1 is X + 1
        ;
            error(5, increment(X, X1)).
```

The predicate **current_error/1** can be used to yield all valid errors, a valid error is that one to which an error message and an error handler are associated. The predicate **error_id/2** gives the corresponding error message to the specified error number. To ease the search for the appropriate error number, the library **util** contains the predicate

```
list_error(Text, N, Message)
```

which returns on backtracking all the errors whose error message contains the string *Text*.
The ability to define any Prolog predicate as the error handler permits a great deal of flexibility in error handling. However, this flexibility should be used with caution. The action of an error handler could have side effects altering the correctness of a program; indeed it could be responsible for further errors being introduced. One particular area of danger is in the use of input and output streams by error handlers. For example: a particular error handler may interact with the user at the terminal, to explain the nature of the error and ask for directions

---

if a simple predicate would cause the invocation of a non-deterministic error handler, the generated code may no longer be correct.

regarding what action should be taken. Care should be taken in such a case to ensure that the error handler does not affect the input to the program. If it does, since program execution continues normally after exit of the error handler, any input consumed by the error handler is lost.

14.2.2 Arguments of Error Handlers

An error handler has 3 optional arguments. The first argument is the number that identifies the error, the second argument is the culprit (a structure corresponding to the call which caused the error). For instance, if, say, a type error occurs upon calling the second goal of the procedure `p(2, Z)`:

```
p(X, Y) :- a(X), b(X, Y), c(Y).
```

the structure given to the error handler is `b(2, Y)`. Note that the handler could bind Y which would have the same effect as if `b/2` had done the binding.
The third argument is only defined for a subset of the existing errors. If the error occurred inside a tool body, it holds the caller module, otherwise it is a free variable. Note that some events are not errors but are used for different purposes. In thoses cases the second and third argument are sometimes used differently. See Appendix D for details.
The error handler is free to ignore some of these arguments, i.e. it can have any arity from 0 to 3. The first argument is provided for the case that the same procedure serves as the handler for several error types - then it can distinguish which is the actual error type. An error handler is just an ordinary Prolog procedure and thus within it a call may be made to any other procedure, or any built in predicate; this in particular means that a call to **exit_block/1** may be made (see the section on the **block/3** predicate). This will work 'through' the call to the error handler, and so an exit may be made from within the handler out of the current block (i.e. back to the corresponding call of the **block/3** predicate). Specifying the predicates **true/0** or **fail/0** as error handlers will make the erroneous predicate succeed (without binding any further variables) or fail respectively.

14.2.3 User Defined Errors

The following example illustrates the use of a user-defined error. We declare a handler for the event 'Invalid command' and raise the new error in the application code.

```
% Command error handler - output invalid command, sound bell and abort
command_error_handler(_, Command) :-
        printf("\007\nInvalid command: %w\n", [Command]),
        abort.

% Activate the handler
:- set_event_handler('Invalid command', command_error_handler/2).

% top command processing loop
go :-
        writeln("Enter command."),
        read(Command),
        ( valid_command(Command)->
                process_command(Command),
                go
            ;
                error('Invalid command',Command)  % Call the error handler
            ).

    % Some valid commands
    valid_command(start).
    valid_command(stop).
```

14.3 Interrupts

Operating systems such as Unix provide a notion of asynchronous interrupts or signals. In a standalone ECL^{i}PS^e system, the signals can be handled by defining interrupt handlers for them. In fact, a set of default handlers is already predefined in this case.
In an embedded ECL^{i}PS^e, signals are usually handled by the host application. It is recommended to use the event mechanism (the ec_post_event() library function) when signals are meant to be handled by ECL^{i}PS^e code.

14.3.1 Interrupt Identifiers

Interrupts are identified either by their signal number (Unix) or by a name which is derived from the name the signal has in the operating system. Most built-ins understand both identifiers. It is usually more portable to use the symbolic name. The built-in **current_interrupt/2** is provided to check and/or generate the valid interrupt numbers and their mnemonic names.

14.3.2 Asynchronous handling

When an interrupt happens the ECL^{i}PS^e system calls an interrupt handling routine in a manner very similar to the case of event handling. The only argument to the handler is the interrupt number. Just as event handlers may be user defined, so it is possible to define interrupt handlers. The goal

```
set_interrupt_handler(N, PredSpec)
```

assigns the procedure specified by *PredSpec* as the interrupt handler for the interrupt identified by *N* (a number or a name). Some interrupts can not be caught by the user (e.g. the *kill* signal), trying to establish a handler for them yields an error message.
To test interrupt handlers, the built-in **kill/2** may be used to send a signal to the own process. The predicate **get_interrupt_handler/3** may be used to find the current interrupt handler for an interrupt N, in the same manner as **get_error_handler**:

```
get_interrupt_handler(N, PredSpec, HomeModule)
```

The predicates **reset_interrupt_handler/1** and **reset_interrupt_handlers/0** are used to reset a particular interrupt handler or all interrupt handlers to their default values.
An interrupt handler has one optional argument, which is the interrupt number. There is no argument corresponding to the error culprit, since the interrupt has no relation to the currently executed predicate. A handler may be defined which takes no argument (such as when the

handler is defined for only one interrupt type). If the handler has one argument, the identifier of the interrupt is passed to the handler when it is called.

When an interrupt occurs, the system halts what it is currently doing and calls the interrupt handler. Just as in the case with error handling, the interrupt handler can be any Prolog procedure. However, unlike the situation in the case of error handling, when the handler exits, be it with success or failure, the execution is resumed at the point where it was interrupted, the interrupt handling is in this case completely independent[2]. This "resume and forget" policy means that to the Prolog program, an interrupt is "invisible" — providing the handler has no side effects, the program continues as if the interrupt had never happened. As a consequence it is not significant whether the handler succeeds or fails. However, again just as in the case of error handlers, a call to the predicate **exit_block/1** may be made in order to escape from within the handler to the corresponding call of **block/3**. Obviously, in this case the interrupted execution can no longer be resumed.

There are a few special settings for interrupt handlers:

default/0

performs the standard UNIX handling of the specified interrupt (signal). Setting this handler is equivalent to calling *signal(N, SIG_DFL)* on the C level. Thus e.g. specifying

```
?- set_interrupt_handler(int, default/0)
```

will exit the ECL^{i}PS^e system when ∧C is pressed.

true/0

This is equivalent to calling *signal(N, SIG_IGN)* on the C level, ie. the interrupt is ignored.

event/1

The signal is handled by posting a (synchronous) event. The event name is the symbolic name of the interrupt.

Apart from these special cases, all other arguments will result in the specified predicate to be called when the appropriate interrupt occurs.

14.3.3 Example

Here is an example for the use of an asynchronous timer signal and a synchronous event handler for implementing a time-out predicate. Mapping the interrupt to an event is necessary in order to cleanly abort the running excecution at a well-defined point in execution.

```
timeout(Goal, Seconds, TimeOutGoal) :-
        block(
            timeout_once(Goal, Seconds),
            timeout,
            call(TimeOutGoal)
        ).

timeout_once(Goal, Seconds) :-
```

---

[2] Note that since the interrupt handler has only one optional argument which is a number, it cannot bind any variables in the current resolvent.

```
        set_timer(real,Seconds),
        call(Goal),
        !,
        set_timer(real,0).
timeout_once(_, _) :-
        set_timer(real,0),
        fail.

timeout_handler :-
        set_timer(real,0),
        exit_block(timeout).

:- set_interrupt_handler(alrm, event/1).
:- set_event_handler(alrm, timeout_handler/0).
```

Chapter 15

Debugging

NOTE: The ECL^{i}PS^e debugger has been completely reimplemented for release 4.1. The debugger is now more modular, easier to extend and can be equipped with different user interfaces (command line, graphical, programmable). It is also more reliable and more scalable with respect to large applications.
Some features of the old debugger are no longer supported, they fall into two categories:

- Features that have been dropped for good in order to reduce time and space overheads and make the system more scalable. These include the possibility of inspecting exited subtrees, the availability of instantiations at fail- and leave-ports, and the tracing of cut and unify ports.
- Features that are desirable and may be reintroduced in some form in forthcoming releases. These include some support for retrying goals and the tracing of external predicates.

15.1 The Box Model

The ECL^{i}PS^e debugger is based on a port model which is an extension of the classical Box Model commonly used in Prolog debugging.
A procedure invocation (or goal) is represented by a box with entry and exit ports. Each time a procedure is invoked, a box is created and given a unique invocation number. The invocations of subgoals of this procedure are seen as boxes inside this procedure box.
Tracing the flow of the execution consists in tracing the crossing of the execution flow through any of the port of the box.
The five basic ports of the box model of ECL^{i}PS^e are the CALL, EXIT, REDO, FAIL and NEXT ports, the suspension facilities are traced through the DELAY and RESUME ports, and the exceptional exit is indicated by LEAVE.

CALL: When a procedure is invoked, the flow of the execution enters the procedure box by its CALL port and enters the first clause box which could (since not all clauses are tried, some of them being sure to fail, i.e. indexing is shown) unify with the goal. It may happen that a procedure is called with arguments that make it sure to fail (because of indexing). In such cases, the flow does not enter any clause box.

For each CALL port a new procedure box is created and is given:

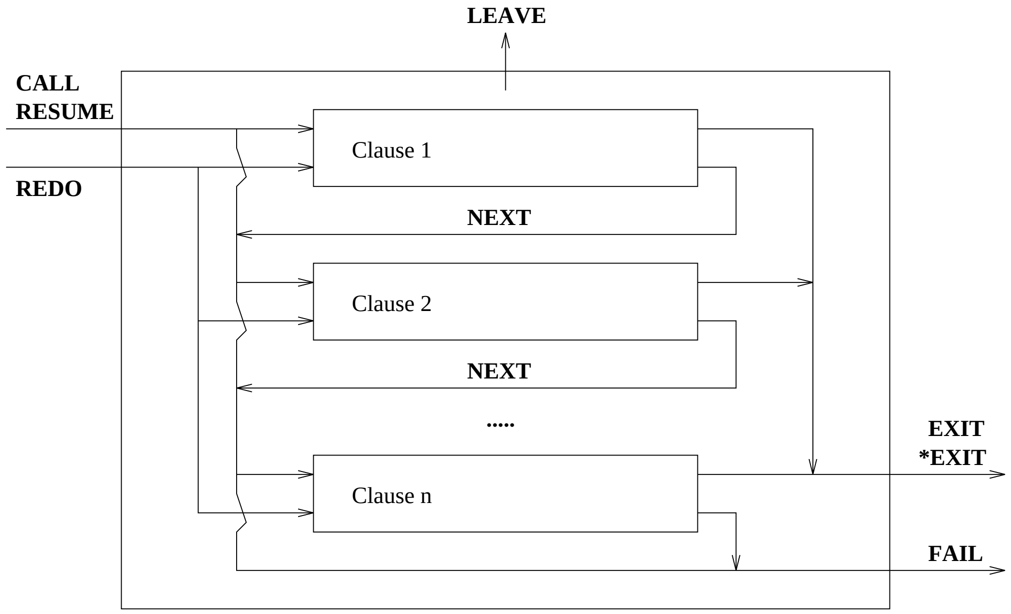


Figure 15.1: The box model

- an *invocation number* that is one higher than that given for the most recent CALL port. This allows to uniquely identify a procedure invocation and all its corresponding ports.
- a *level* that is one higher than that of its parent goal.

The displayed variable instantiations are the ones at call time, i.e. before the head unification of any clause.

EXIT: When a clause of a predicate succeeds (i.e. unification succeeded and all procedures called by the clause succeeded), the flow gets out of the box by the EXIT port of both boxes (only the EXIT port of the *procedure box* is traced).

When a procedure exits non-deterministically (and there are still other clauses to try on that procedure or one of its children goals has alternatives which could be resatisfied), the EXIT port is traced with an asterisk (*EXIT). When the last possibly matching clause of a procedure is exited, the exit is traced without asterisk. This means that this procedure box will never be retried as there is no other untried alternative.

The instantiations shown in the EXIT port are the ones at exit time, they result from the (successful) execution of the procedure.

FAIL: When a clause of a procedure fails (because head unification failed or because a sub-goal failed), the flow of the execution exits the clause box and leaves the procedure box via the FAIL port. Note that the debugger cannot display any argument information at FAIL ports (an ellipsis ... is displayed instead for each argument).

NEXT: If a clause fails and there is another possibly matching clause to try, then that one is tried for unification. The flow of the execution from the failure of one clause to the head unification of a following clause is traced as a NEXT port. The displayed variable instantiations are the same as those of the corresponding CALL or REDO port.

ELSE: This is similar to the NEXT port, but indicates that the next branch of a **disjunction (;/2)** it tried after the previous branch failed. The predicate that gets displayed with the port is the predicate which contains the disjunction (the immediate ancestor).

REDO: When a procedure box is exited trough an *EXIT port, the box can be retried later to get a new solution. This will happen when a later goal fails. The backtracking will cause failing of all procedures that do not have any alternative, then the execution flow will enter a procedure box that an contains alternative through a REDO port.

Two situations may occur: either the last tried clause has called a procedure that has left a choice point (it has exited through an *EXIT port). In that case the nested procedure box is re-entered though another REDO-port.

Otherwise, if the last clause tried does not contain any nondeterministically exited boxes, but there are other untried clauses in the procedure box, the next possibly matching clause will be tried.

The last REDO port in such a sequence is the one which contains the actual alternative that is tried. The variable instantiations for all REDO ports in such a sequence are the ones corresponding to the call time of the last one.

LEAVE: This port allows to trace the execution of a the **block/3** and **exit_block/1** predicates within the box model. The predicate **block/3** is traced as a normal procedure. If the goal in its first argument fails, **block/3** fails, if it exits, **block/3** exits. If the predicate **exit_block/1** is called (and exited since it never fails), all the goals inside the matching block are left through a special port called LEAVE, so that each entry port matches with an exit port. The recover procedure (in the third argument of **block/3**) is then called and traced normally and **block/3** will exit or fail (or even leave) depending on the recover procedure.

As with the FAIL port, no argument value are displayed in the LEAVE port.

DELAY: The displayed goal becomes suspended. This is a singleton port, it does not enter or leave a box. However, a new *invocation number* is assigned to the delayed goal, and this number will be used in the matching RESUME port. The DELAY port is caused by one of the built-in predicates **suspend/3**, **suspend/4**, **make_suspension/3** or a delay clause. The port is displayed just after the delayed goal has been created.

RESUME: When a waking condition causes the resuming of a delayed goal, the procedure box is entered through its RESUME port. The box then behaves as if it had been entered through its CALL port. The invocation number is the same as in its previous DELAY port. which makes it easy to identify corresponding delay and resume events. However the depth level of the RESUME corresponds to the waking situation. It is traced like a subgoal of the goal which has caused the waking.

In the rest of this chapter the user interface to the debugger is described, including the commands available in the debugger itself as well as built-in predicates which influence it. Some of the debugger commands are explained using an excerpt of a debugger session. In these examples, the user input is always underlined (it is in fact not always output as typed) to distinguish it from the computer output. For the description of the windowing interface to the debugger in the KEGI environment see [5].

15.2 Format of the Tracing Messages

All trace messages are output to the **debug_output** stream, (see section 12.2), the debugger command input is taken from the stream **debug_input**. These streams are by default connected to the user's terminal.
The format of one trace line is as follows:

```
S+(4) 2 *EXIT module:foo(one, X, two)   %>
12 3  4 5 6     7       8               9
```

1. The first character shows some properties of the displayed procedure. It may be one of
   - C - an external procedure, not implemented in Prolog
   - S - a *skipped* procedure, i.e. a procedure whose subgoals are not traced
2. A '+' displayed here shows that the procedure has a spy point set.
3. The number between parentheses shows the box invocation number of this procedure call. Since each box has a unique invocation number, it can be used to identify ports that belong to the same box. It also shows how many procedure redos have been made since the beginning of the query. Only boxes that can be traced obtain an invocation number, for instance subgoals of a procedure which is compiled in debug mode or has its skip-flag set are not numbered.

   When a delayed goal is resumed, it keeps the invocation number it was assigned when it delayed. This makes it easy to follow all ports of a specified call even in data-driven computation.
4. The second number shows the level or depth of the goal, i.e. the number of its ancestor boxes. When a subgoal is called, the level increases and after exit it decreases again. The initial level is 1.

   Since a resumed goal is considered to be a descendant of the procedure that woke it, the level of a resumed goal may be different from the level the goal had when it delayed.
5. An asterisk before an EXIT means that this procedure is nondeterministic and that it might be resatisfied.
6. The next word is the name of the port. It might be missing if the displayed goal is not the current position in the execution (e.g. when examining ancestors or delayed goals).

   CALL a procedure is called for the first time concerning a particular invocation,

   DELAY a procedure delays,

   EXIT a procedure succeeds,

   FAIL a procedure fails, there is no (other) solution,

   LEAVE a procedure is left before having failed or exited because of a call to **exit_block/1**,

NEXT the next possibly matching clause of a procedure is tried because unification failed or a sub-goal failed,

ELSE the next branch of a disjunction is tried because some goal in the previous branch failed.

REDO a procedure that already gave a solution is called again for an alternative,

RESUME a procedure is woken (the flow enters the procedure box as for a call) because of a unification of a suspending variable,

7. The optional module name followed by a colon. Printing of the module can be enabled and disabled by the debugger command **m**. If it is enabled, the module from where the procedure is called is displayed. By default the module printing is disabled.

8. The goal is printed according to the current instantiations of its variables. Arguments of the form `...` represent subterm that are not printed due to the depth limit in effect. The depth limit can be changed using the < command.

   The goal is printed with the current **output_mode** settings. which can be changed using the **o** command.

9. The prompt of the debugger, which means that it is waiting for a command from the user.

15.3 Debugging-related Predicate Properties

Predicates have a number of properties which can be listed using the **pred/1** built-in. The following predicate flags and properties affect the way the predicate is traced by the debugger:

debugged
Indicates whether the predicate has been compiled in debug-compile mode. If **on**, calls the the predicate's subgoal will be traced. The value of this property can only be changed by re-compiling the predicate in a different mode.

leash
If **notrace**, no port of the predicate will be shown in the trace (but the invocations will be counted nevertheless). If **stop**, the ports of this predicate will be shown and the debugger will stop and await new commands. (The **print** setting is currently not supported). The value of this property can be changed with **traceable/1**, **untraceable/1** or **set_flag/3**.

spy
If **on**, the predicate has a spy-point and the debugger will stop at its ports when in leap mode. The value of this property can be changed with **spy/1**, **nospy/1** or **set_flag/3**.

skipped
If **on**, the predicate's subgoal will not be traced even if it has been compiled in debug-compile mode. The value of this property can be changed with **skippped/1**, **unskippped/1** or **set_flag/3**.

start_tracing
If **on**, a call to the predicate will activate the debugger if it is not already running. Only the execution within this predicate's box will be traced. This is useful for debugging part of a big program without having to change the source code. The effect is similar to wrapping all call of the predicate into **trace/1**.

15.4 Starting the Debugger

Several methods can be used to switch the debugger on. If the interactive top-level is used, the commands **trace/0** and **debug/0** are used to switch the debugger on for the following queries typed from the top-level. **trace/0** will switch the debugger to *creep* mode whereas **debug/0** will switch it in it leap mode. When the debugger is in it creep mode, it will prompt for a command at the crossing of the first port of a leashed procedure. When the debugger is in *leap* mode, it will prompt for a command at the first port of a leashed procedure that has a spy point. The debugger is switched off either from the toplevel with the commands **nodebug/0** or **notrace/0**, or by typing **n** or **N** to the debugger prompt.

A spy point can be set on a procedure using **spy/1** (which will also switch the debugger to *leap*) and removed with **nospy/1**. They both accept a *SpecList* as argument. Note that **set_flag/3** can be used to set and reset spy points without switching the debugger on and without printing messages.

debugging/0 can be used to list the spied predicates and the current debugger mode.

```
[eclipse 1]: spy writeln/1.
spypoint added to writeln / 1.

yes.
Debugger switched on - leap mode
[eclipse 2]: debugging.
Debug mode is leap
writeln / 1 is being spied

yes.
[eclipse 3]: true, writeln(hello), true.
B+(2) 0  CALL   writeln(hello) %> l leap
hello
B+(2) 0  EXIT   writeln(hello) %> c creep
B (3) 0  CALL   true %> l leap

yes.
[eclipse 4]: trace.
Debugger switched to creep mode

yes.
[eclipse 5]: true, writeln(hello), true.
B (1) 0  CALL   true %> c creep
B (1) 0  EXIT   true %> c creep
B+(2) 0  CALL   writeln(hello) %> l leap
hello
B+(2) 0  EXIT   writeln(hello) %> l leap

yes.
```

15.5 Debugging Parts of Programs

15.5.1 Mixing debuggable and non-debuggable code

The debugger can trace only procedures which have been compiled in debug mode. The compiler debug mode is by default switched on and it can be changed globally by setting the flag *debug_compile* with the **set_flag/2** predicate or using **dbgcomp/0** or **nodbgcomp/0**. The global compiler debug mode can be overruled on a file-by-file basis using one of the compiler pragmas

```
:- pragma(nodebug).
:- pragma(debug).
```

Once a program (or a part of it) has been debugged, it can be compiled in *nodbgcomp* mode so that all optimisations are done by the compiler. The advantages of non-debugged procedures are

- They run slightly faster than the debugged procedures when the debugger is switched off. When the debugger is switched on, the non-debugged procedures run considerably faster than the debugged ones and so the user can selectively influence the speed of the code which is being traced as well as its space consumption.
- Their code is shorter than that of the debugged procedures.

Although only procedures compiled in the *dbgcomp* mode can be traced, it is possible to mix the execution of procedures in both modes. Then, calls of *nodbgcomp* procedures from *dbgcomp* ones are traced, however further execution within *nodbgcomp* procedures, i.e. the execution of their subgoals, no matter in which mode, is not traced. In particular, when a *nodbgcomp* procedure calls a *dbgcomp* one, the latter is normally not traced. There are two important exceptions from this rule:

- When a debuggable procedure has delayed and its DELAY port has been traced, then its RESUME port is also traced, even when it is woken inside non-debuggable code.
- When non-debuggable code *meta-calls* a debuggable procedure (i.e. via **call/1**), then this procedure can be traced. This is a useful feature for the implementation of meta- predicates like **setof/3**, because it allows to hide the details of the setof-implementation, while allowing to trace the argument goal.

Setting a procedure to skipped (with **set_flag/3** or **skipped/1**) is another way to speed up the execution of procedures that do not need to be debugged. The debugger will ignore everything that is called inside the skipped procedure like for a procedure compiled in *nodbgcomp* mode. However, the debugger will keep track of the execution of a procedure skipped with the command **s** of the debugger so that it will be possible to 'creep' in it on later backtracking or switch the debugger to *creep* mode while the skip is running (e.g. by interrupting a looping predicate with ^C and switching to *creep* mode).

The two predicates **trace/1** and **debug/1** can be used to switch on the debugger in the middle of a program. They execute their argument in *creep* or *leap* mode respectively. This is particularly useful when debugging large programs that take too much time (or need a lot of memory) to run completely with the debugger.

```
[eclipse 1]: debugging.
Debugger is switched off

yes.
[eclipse 2]: big_goal1, trace(buggy_goal), big_goal2.
Start debugging - creep mode
  (1) 0  CALL   buggy_goal %> c creep
  (1) 0  EXIT   buggy_goal %> c creep
Stop debugging.

yes.
```

It is also possible to enable the debugger in the middle of execution without changing the code. To do so, use **set_flag/3** to set the **start_tracing** flag of the predicate of interest. Tracing will then start (in leap mode) at every call of this predicate[1]. To see the starting predicate itself, set a spy point in addition to the **start_tracing** flag:

```
[eclipse 1]: debugging.
Debugger is switched off

yes.
[eclipse 2]: set_flag(buggy_goal/0, start_tracing, on),
             set_flag(buggy_goal/0, spy, on).

yes.
[eclipse 3]: big_goal1, buggy_goal, big_goal2.
 +(0) 0 CALL  buggy_goal   %> creep
 +(0) 0 EXIT  buggy_goal   %> creep

yes.
```

15.6 Using the Debugger via the Command Line Interface

This section describe the commands available at the debugger prompt in the debugger's command line interface (for the graphical user interface, please refer to the online documentation). Commands are entered by typing the corresponding key (without newline), the case of the letters is significant. The action of some of them is immediate, others require additional parameters to be typed afterwards. Since the ECL^{i}PS^e debugger has the possibility to display not only the goal that is currently being executed (the *current* goal or procedure), but also its predecessors, some of the commands may work on the *displayed* procedure whatever it is, and others on the *current* one.

15.6.1 Counters and Command Arguments

Some debugger commands accept a counter (a small integer number) before the command letter (e.g. **c** creep). The number is just prefixed to the command and terminated by the command letter itself. If a counter is given for a command that doesn't accept a counter, it is ignored.

---

[1] provided the call has been compiled in debug_compile mode, or the call is a meta-call

When a counter is used and is valid for the command, the command is repeated, decrementing the counter until zero. When repeating the command, the command and the remaining counter value is printed after the debugger prompt instead of waiting for user input.
Some commands prompt for a parameter, e.g. the **j** *(jump)* command asks for the number of the level to which to jump. Usually the parameter has a sensible default value (which is printed in square backets). If just a newline is typed, then the default value is taken. If a valid parameter value is typed, followed by newline, this value is taken. If an illegal letter is typed, the command is aborted.

15.6.2 Commands to Continue Execution

All commands in this section continue program execution. They difference between them is the condition under which execution will stop the next time. When execution stops again, the next trace line is printed and a new command is accepted.

*n*c **creep**
This command allows exhaustive tracing: the execution stops at the next port of any leashed procedure. No further parameters are required, a counter *n* will repeat the command *n* times. It always applies on the current procedure, even when the displayed procedure is not the current one (e.g. during term inspection). An alias for the **c** command is to just type newline (Return-key).

*n*s **skip**
If given at an entry port of a box (CALL, RESUME, REDO), this command skips the execution until an exit port of this box (EXIT, FAIL, LEAVE). If given in an exit port it works like *creep.* (Note that sometimes the **i** command is more appropriate, since it skips to the next port of the current box, no matter which). A counter, if specified, repeats this command.

*n*l **leap**
Continues to the next spy point (any port of a procedure which has its spy flag set). A counter, if specified, repeats this command.

i *par* **invocation skip**
Continue to the next port of the box with the invocation number specified. The default invocation number is the one of the current box. Common uses for this command are to skip from CALL to NEXT, from NEXT to NEXT/EXIT/FAIL, from *EXIT to REDO, or from DELAY to RESUME.

j *par* **jump to level**
Continue to the next port with the specified nesting level (which can be higher or lower than the current one). The default is the parent's level, i.e. to continue until the current box is exited, ignoring all the remaining subgoals of the current clause. This is particularly useful when a **c** (*creep*) has been typed where a **s** (*skip*) was wanted.

n **nodebug**
This command switches tracing off for the remainder of the execution. However, the next top-level query will be traced again. Use **N** to switch traceing off permanently.

v **var/term modification skip**

This command sets up a monitor on the currently displayed term, which will cause a MODIFY-port to be raised on each modification to any variable in the term. These ports will all have a unique invocation number which is assigned and printed at the time the command is issued. This number can then be used with the **i** command to skip to where the modifications happen.

```
[eclipse 4]: [X, Y] :: 1..9, X #>= Y, Y#>1.
  (1) 1 CALL  [X, Y] :: 1..9  %> var/term spy? [y]
Var/term spy set up with invocation number (2)  %> jump to invoc: [1]? 2
  (2) 3 MODIFY  [X{[1..9]}, Y{[2..9]}] :: 1..9  %> jump to invoc: [2]?
  (2) 4 MODIFY  [X{[2..9]}, Y{[2..9]}] :: 1..9  %> jump to invoc: [2]?
```

Note that these monitors can also be set up from within the program code using one of the built-ins **spy_var/1** or **spy_term/2**.

z *par* **zap**

This command allows to skip over, or to a specified port. When this command is executed, the debugger prompts for a port name (e.g. **fail** or a negated port name (e.g. ∼**exit**). Execution then continues until the specified port appears or, in the negated case, until another than the specified port appears. The default is the negation of the current port, which is useful when exiting from a deep recursion (a long sequence of EXIT or FAIL ports).

15.6.3 Commands to Modify Execution

f *par* **fail**

Force a failure of the procedure with the specified invocation number. The default is to force failure of the current procedure.

a **abort**

Abort the execution of the current query and return to the top-level. The command prompts for confirmation.

15.6.4 Display Commands

This group of commands cause some useful information to be displayed.

d *par* **delayed goals**

Display the currently delayed goals. The optional argument allows to restrict the diplay to goal of a certain priority only. The goals are displayed in a format similar to the trace lines, except that there is no depth level and no port name. Instead, the goal priority is displayed in angular brackets:

```
[eclipse 5]: [X, Y] :: 1..9, X #>= Y, Y #>= X.
  (1) 1 CALL  [X, Y] :: 1..9  %> creep
  (1) 1 EXIT  [X{[1..9]}, Y{[1..9]}] :: 1..9  %> creep
  (2) 1 CALL  X{[1..9]} - Y{[1..9]}#>=0  %> creep
  (3) 2 DELAY  X{[1..9]} - Y{[1..9]}#>=0  %> creep
  (2) 1 EXIT  X{[1..9]} - Y{[1..9]}#>=0  %> creep
```

```
    (4) 1 CALL  Y{[1..9]} - X{[1..9]}#>=0   %> creep
    (5) 2 DELAY  Y{[1..9]} - X{[1..9]}#>=0   %> delayed goals
                                                  with prio: [all]?
  ------- delayed goals -------
    (3) <2>  X{[1..9]} - Y{[1..9]}#>=0
    (5) <2>  Y{[1..9]} - X{[1..9]}#>=0
  ------------ end ------------
    (5) 2 DELAY  Y{[1..9]} - X{[1..9]}#>=0   %>
```

u *par* **scheduled goals**
Similar to the **d** command, but displays only those delayed goals that are already scheduled for execution. The optional argument allows to restrict the diplay to goal of a certain priority only. Example:

```
  [eclipse 13]: [X,Y,Z]::1..9, X#>Z, Y#>Z, Z#>1.
    (1) 1 CALL  [X, Y, Z] :: 1..9   %> creep
    (1) 1 EXIT  [X{[1..9]}, Y{[1..9]}, Z{[1..9]}] :: 1..9   %> creep
    (2) 1 CALL  X{[1..9]} - Z{[1..9]}+-1#>=0   %> creep
    (3) 2 DELAY  X{[2..9]} - Z{[1..8]}#>=1   %> creep
    (2) 1 EXIT  X{[2..9]} - Z{[1..8]}+-1#>=0   %> creep
    (4) 1 CALL  Y{[1..9]} - Z{[1..8]}+-1#>=0   %> creep
    (5) 2 DELAY  Y{[2..9]} - Z{[1..8]}#>=1   %> creep
    (4) 1 EXIT  Y{[2..9]} - Z{[1..8]}+-1#>=0   %> creep
    (6) 1 CALL  0 + Z{[1..8]}+-2#>=0   %> creep
    (3) 2 RESUME  X{[2..9]} - Z{[2..8]}#>=1   %> scheduled goals
                                                  with prio: [all]?
  ------ scheduled goals ------
    (5) <2>  Y{[2..9]} - Z{[2..8]}#>=1
  ------------ end ------------
    (3) 2 RESUME  X{[2..9]} - Z{[2..8]}#>=1   %>
```

G **all ancestors**
Prints all the current goal's ancestors from the oldest to the newest. The display format is similar to trace lines, except that `....` is displayed in the port field.

. **print definition**
If given at a trace line, the command displays the source code of the current predicate. If the predicate is not written in Prolog, or has not been compiled from a file, or the source file is inaccessible, no information can be displayed.

h **help**
Print a summary of the debugger commands.

? **help**
Identical to the **h** command.

15.6.5 Navigating among Goals

While the debugger waits for commands, program execution is always stopped at some port of some predicate invocation box, or goal. Apart from this current goal, two types of other goals are also active. These are the ancestors of the current goal (the enclosing, not yet exited boxes in the box model) and the delayed goals. The debugger allows to navigate among these goals and inspect them.

g ancestor
Move to and display the ancestor goal (or parent) of the displayed goal. Repeated application of this command allows to go up the call stack.

x *par* examine goal
Move to and display the goal with the specified invocation number. This must be one of the active goals, i.e. either an ancestor of the current goal or one of the currently delayed goals. The default is to return to the current goal, i.e. to the goal at whose port the execution is currently stopped.

15.6.6 Inspecting Goals and Data

This family of commands allow the subterms in the goal displayed at the port to be inspected[2]. The ability to inspect subterms is designed to help overcome two problems when examining a large goal with the normal display of the goal at a debug port:

1. Some of the subterms may be omitted from the printed goal because of the print-depth;
2. If the user is interested in particular subterms, it may be difficult to precisely locate them from the surrounding arguments, even if it is printed.

With inspect subterm commands, the user is able to issue commands to navigate through the subterms of the current goal and examine them. A *current subterm* of the goal is maintained, and this is printed after each inspect subterm command, instead of the entire goal. Initially, the current subterm is set to the goal, but this can then be moved to the subterms of the goal with navigation commands.
Once inspect subterm is initiated by an inspect subterm command, the debugger enters into the inspect subterm mode. This is indicated in the trace line by 'INSPECT' instead of the name of the port, and in addition, the goal is not shown on the trace line:

```
INSPECT  (length/2)   %>
```

Instead of showing the goal, a summary of the current subterm – generally its functor and arity if the subterm is a structure – is shown in brackets.

# *par* move down to *par*th argument

The most basic command of inspect subterm is to move the current subterm to an argument of the existing current subterm. This is done by typing a number followed by carriage return, or by typing `#`, which causes the debugger to prompt for a number. In both cases, the number specifies the argument number to move down to. In the following example, the `#` style of the command is used to move to the first argument, and the number style of the command to move to the third argument:

---

[2] In ECL^{i}PS^e 4.0, this was implemented as a submode (invoked with two key strokes - Hi). It is now fully integrated into the debugger

```
  (1) 1 CALL  foo(a, g(b, [1, 2]), X)   %> inspect arg #: 1<NL>
a
        INSPECT  (atom)   %>

  (1) 1 CALL  foo(a, g(b, [1, 2]), X)   %>  3<NL>
X
        INSPECT  (var)   %>
```

The new current subterm is printed, followed by the INSPECT trace line. Notice that the summary shows the type of the current subterm, instead of Name/Arity, since in both cases the subterms are not structures.

If the current subterm itself is a compound term, then it is possible to recursively navigate into the subterm:

```
  (1) 1 CALL  foo(a, g(b, [1, 2]), X)   %> 2<NL>
g(b, [1, 2])
        INSPECT  (g/2)   %> 2<NL>
[1, 2]
        INSPECT  (list  1-head 2-tail)   %> 2<NL>
[2]
        INSPECT  (list  1-head 2-tail)   %>
```

Notice that lists are treated as a structure with arity 2, although the functor (`./2`) is not printed.

In addition to compound terms, it is also possible to navigate into the attributes of attributed variables:

```
[eclipse 21]: suspend(foo(X), 3, X->inst), foo(X).<NL>
  (1) 1 DELAY  foo(X)   %> <NL>
creep
  (2) 1 CALL  foo(X)   %> 1<NL>
X
        INSPECT  (attributes  1-suspend 2-fd )   %>1<NL>
suspend(['SUSP-1-susp'|_218] - _218, [], [])
        INSPECT  (struct suspend/3)   %>
```

The variable X is an attributed variable in this case, and when it is the current subterm, this is indicated in the trace line. The debugger also shows the user the currently available attributes, and the user can then select one to navigate into (`fd` is available in this case because the finite domain library was loaded earlier in the session. Otherwise, it would not be available as a choice here).

Note that the `suspend/3` summary contains a `struct` before it. This is because the `suspend/3` is a predefined structure with field names (see section 6.1). It is possible to view the field names of such structures using the `.` command in inspect mode.

If the number specified is larger than the number of the arguments of the current subterm, then an error is reported and no movement is made:

```
foo(a, g(b, [1, 2]), 3)
        INSPECT  (foo/3)   %> 4<NL>

Out of range.....

foo(a, g(b, [1, 2]), 3)
        INSPECT  (foo/3)   %>
```

*n*uparrow key Move current subterm up by N levels

*n*A Move current subterm up by N levels

In addition to moving the current subterm down, it can also be moved up from its current position. This is done by typing the uparrow key. This key is mapped to `A` by the debugger, so one can also type `A`. Typing `A` may be necessary for some configurations (combination of keyboards and operating systems) because the uparrow key is not correctly mapped to `A`.

An optional argument can preceded the uparrow keystroke, which indicates the number of levels to move up. The default is 1:

```
  (1) 1 CALL  foo(a, g(b, [1, 2]), 3)   %> 2<NL>
g(b, [1, 2])
        INSPECT  (g/2)   %> 1<NL>
b
        INSPECT  (atom)   %> up subterm
g(b, [1, 2])
        INSPECT  (g/2)   %> 1up subterm
foo(a, g(b, [1, 2]), 3)
        INSPECT  (foo/3)   %>
```

The debugger prints `up subterm` when the uparrow key is typed. The current subterm moves back up the structure to its parent for each level it moves up, and the above move can be done directly by specifying 2 as the levels to move up:

```
b
        INSPECT  (atom)   %> 2up subterm
foo(a, g(b, [1, 2]), 3)
        INSPECT  (foo/3)   %>
```

If the number of levels specified is more than the number of levels that can be traversed up, the current subterm stops at the toplevel:

```
  (1) 1 CALL  foo(a, g(b, [1, 2]), 3)   %> 2<NL>
g(b, [1, 2])
        INSPECT  (g/2)   %> 2<NL>
[1, 2]
        INSPECT  (list  1-head 2-tail)   %> 5up subterm
```

```
foo(a, g(b, [1, 2]), 3)
        INSPECT  (foo/3)   %>
```

0 Move current subterm to toplevel

It is possible to quickly move back to the top of a goal that is being inspected by specifying 0 (zero) as the command:

```
  (1) 1 CALL  foo(a, g(b, [1, 2]), 3)   %> 2<NL>
g(b, [1, 2])
        INSPECT  (g/2)   %> 2<NL>
[1, 2]
        INSPECT  (list  1-head 2-tail)   %> 2<NL>
[2]
        INSPECT  (list  1-head 2-tail)   %> 2<NL>
[]
        INSPECT  (atom)   %> 0<NL>
foo(a, g(b, [1, 2]), 3)
        INSPECT  (foo/3)   %>
```

Moving to the top can also be done by the `#` command, and not giving any argument (or `0`) when prompted for the argument.

*n*leftarrow key Move current subterm left by N positions

*n*D Move current subterm left by N positions

The leftarrow key (or the equivalent `D`) moves the current subterm to a sibling subterm (i.e. fellow argument of the parent structure) that is to the left of it. Consider the structure `foo(a, g(b, [1, 2]), 3)`, then for the second argument, `g(b, [1, 2])`, `a` is its (only) left sibling, and `3` its (only) right sibling. For the third argument, `3`, both `a` (distance of 2) and `g(b, [1, 2])` (distance of 1) are its left siblings. The optional numeric argument for the command specifies the distance to the left that the current subterm should be moved. It defaults to 1.

```
foo(a, g(b, [1, 2]), 3)
        INSPECT  (foo/3)   %> 3<NL>
3
        INSPECT  (integer)   %> 2left subterm
a
        INSPECT  (atom)   %>
```

If the leftward movement specified would move the argument position before the first argument of the parent term, then the movement will stop at the first argument:

```
foo(a, g(b, [1, 2]), 3)
        INSPECT  (foo/3)   %> 3<NL>
```

```
    3
            INSPECT  (integer)   %> 5left subterm
    a
            INSPECT  (atom)   %>
```

In the above example, the current subterm was at the third argument, thus trying to move left by 5 argument positions is not possible, and the current subterm stopped at leftmost position – the first argument.

*n*rightarrow key Move current subterm right by N positions

*n*C Move current subterm right by N positions

The rightarrow key (or the equivalent `C`) moves the current subterm to a sibling subterm (i.e. fellow argument of the parent structure) that is to the right of it. Consider the structure `foo(a, g(b, [1, 2]), 3)`, then for the first argument, `a`, `g(b, [1, 2])` is a right sibling with distance of 1, and `3` is a right sibling with distance of 2. The optional numeric argument for the command specifies the distance to the left that the current subterm should be moved. It defaults to 1.

```
    foo(a, g(b, [1, 2]), 3)
            INSPECT  (integer)   %> 2left subterm
    a
            INSPECT  (atom)   %>
```

If the rightward movement specified would move the argument position beyond the last argument of the parent term, then the movement will stop at the last argument:

```
    foo(a, g(b, [1, 2]), 3)
            INSPECT  (foo/3)   %> 3<NL>
    3
            INSPECT  (integer)   %> right subterm
    3
            INSPECT  (integer)   %>
```

In the above example, the current subterm was at the third (and last) argument, thus trying to move to the right (by the default 1 position in this case) is not possible, and the current subterm remains at the third argument.

*n*downarrow key Move current subterm down by N levels

*n*B Move current subterm down by N levels

The down-arrow key moves the current subterm down from its current position. This command is only valid if the current subterm is a compound term and so has subterms itself. A structure has in general more than one argument, so there is a choice of which argument position to move down to. This argument is not directly specified by the user as part of the command, but is implicitly specified: the argument position selected is the argument position of the current subterm within its parent:

```
    foo(a, g(b, [1, 2]), 3)
            INSPECT  (foo/3)   %> 2<NL>
    g(b, [1, 2])
            INSPECT  (list  1-head 2-tail)   %> 3down subterm 2 for 3 levels
    []
            INSPECT  (atom)   %>
```

In the above example, the user moves down into the second argument, and then use the down-arrow key to move down into the second argument for 2 levels – the numeric argument typed before the arrow key specified the number of levels that the current subterm was moved down by. The command moves into the second argument because it was at the second argument position when the command was issue.

However, there is not always an argument position for the current sub-term. For example, when the current sub-term is at the toplevel of the goal or if it is at an attribute. In these cases, the default for the argument position to move down into is the first argument:

```
            INSPECT  (atom)   %> 0<NL>
    foo(a, g(b, [1, 2]), 3)
            INSPECT  (foo/3)   %> down subterm 1 for 1 levels
    a
            INSPECT  (atom)   %>
```

In the above example, the down-arrow key is typed at the top-level, and thus the argument position chosen for moving down is first argument, with the default numeric argument for the

If the argument position to move into is beyond the range of the current subterm's number of arguments, then no move is performed:

```
      (1) 1 CALL  foo(a, b, c(d, e))   %> 3<NL>
    c(d, e)
            INSPECT  (c/2)   %> Out of range after traversing down arg...
    c(d, e)
            INSPECT  (c/2)   %>
```

In this case, the down-arrow key was typed in the second trace line, which had the current subterm at the third argument of its parent term, and thus the command tries to move the new current subterm to the third argument of the current sub-term, but the structure does not have a third argument and so no move was made. In the case of moving down multiple levels, then the movement will stop as soon as the argument position to move down to goes out of range.

Moving down is particularly useful for traversing lists. As discussed, lists are really structures with arity two, so the #N command would not move to the N^{th} element of the list. With the down-arrow command , it is possible to move into the N^{th} position in one command:

```
[eclipse 30]: foo([1,2,3,4,5,6,7,8,9]).
  (1) 1 CALL  foo([1, 2, 3, ...])   %> 1<NL>
[1, 2, 3, 4, ...]
        INSPECT  (list  1-head 2-tail)   %> 2<NL>
[2, 3, 4, 5, ...]
        INSPECT  (list  1-head 2-tail)   %> 6down subterm 2 for 6 levels
[8, 9]
        INSPECT  (list  1-head 2-tail)   %>
```

In order to move down a list, we repeatedly move into the tail of the list – the second argument position. In order to do this with the down-arrow command, we need to be at the second argument position first, and this is done in the second trace line. Once this is done, then it is possible to move arbitrarily far down the list in one go, as is shown in the example.

. **Print structure definition**

In ECL^{i}PS^e, it is possible to define field names for structures (see section 6.1). If the inspector encounters such structures, then the user can get the debugger to print out the field names. Note that this functionality only applies within the inspect subterm mode, as the debugger command '`.`' normally prints the source for the predicate. The fact that a structure has defined field names are indicated by a "struct" in the summary:

```
:- local struct(capital(city,country)).

.....

  (1) 1 CALL  f(capital(london, C))   %> 1<NL>
capital(london, C)
        INSPECT  (struct capital/2)   %> structure definition:
1=city 2=country
   %>
```

In this example, a structure definition was made for `captial/2`. When this structure is the current subterm in the inspect mode, the `struct` in the summary for the structure indicates that it has a structure definition. For such structures, the field names are printed by the structure definition command.

If the command is issued for a term that does not have a structure definition, an error would be reported:

```
        INSPECT  (f/1)   %> structure definition:
No struct definition for term f/1@eclipse.
   %>
```

p **Show subterm path**

As the user navigates into a term, then at each level, a particular argument position (or attribute, in the case of attributed variables) is selected at each level. The user can view the position the current subterm is at by the `p` command. For example,

```
  (1) 1 CALL  foo(a, g(b, [1, 2]), 3)   %> 2<NL>
g(b, [1, 2])
        INSPECT  (g/2)   %> 2<NL>
[1, 2]
        INSPECT  (list  1-head 2-tail)   %> 1<NL>
1
        INSPECT  (integer)   %> p
Subterm path:  2, 2, 1
   %>
```

The subterm path shows the argument positions taken at each level of the toplevel term to reach the current subterm, starting from the top.

Extra information (in addition to the numeric argument position) will be printed if the subterm at a particular level is either a structure with field names or an attributed variable. For example:

```
:- local struct(capital(city,country)).

.....

[eclipse 8]: suspend(capital(london, C), 3, C -> inst), f(capital(london, C)).

....

  (2) 1 CALL  f(capital(london, C))   %> 1<NL>
capital(london, C)
        INSPECT  (struct capital/2)   %> 2<NL>
C
        INSPECT  (attributes  1-suspend )   %> 1<NL>
suspend(['SUSP-1-susp'|_244] - _244, [], [])
        INSPECT  (struct suspend/3)   %> 1<NL>
['SUSP-1-susp'|_244] - _244
        INSPECT  (-/2)   %>
Subterm path:  1, country of capital (2), attr: suspend, inst of suspend (1)
   %>
```

In this example, except for the toplevel argument, all the other positions are either have field names or are attributes. This is reflected in the path, for example, `country of capital (2)` shows that the field name for the selected argument position (2, shown in brackets) is `country`, and the structure name is `capital`. For the 'position' of the selected attribute (`suspend`) of the attributed variable `C`, the path position is shown as `attr: suspend`.

Interaction between inspect subterm and output modes

The debugger commands that affect the print formats in the debugger also affects the printed current subterm. Thus, both the print depth and output mode of the printed subterm can be changed.

The changing of the output modes can have a significant impact on the inspect mode. This is because for terms which are transformed by write macros before they are printed (see chapter 13), different terms can be printed depending on the settings of the output modes. In particular, output transformation is used to hide many of the implementation related extra fields and even term names of many ECL^{i}PS^e data structures (such as those used in the finite domain library). For the purposes of inspect subterms, the term that is inspected is always the printed form of the term, and thus changing the output mode can change the term that is being inspected.

Consider the example of looking at the attribute of a finite domain variable:

```
A{[4..10000000]}
        INSPECT  (attributes  1-suspend 2-fd )   %> 2<NL>
[4..10000000]
        INSPECT  (list  1-head 2-tail)   %> 1<NL>
4..10000000
        INSPECT  (../2)   %> 2up subterm
A{[4..10000000]}
        INSPECT  (attributes  1-suspend 2-fd )   %> <o>
current output mode is "QPm", toggle char: T
new output mode is "TQPm".
A{[4..10000000]}
        INSPECT  (attributes  1-suspend 2-fd )   %> 2<NL>
fd(dom([4..10000000], 9999997), [], [], [])
        INSPECT  (struct fd/4)   %> 1<NL>
dom([4..10000000], 9999997)
        INSPECT  (dom/2)   %>
```

After selecting the output mode `T`, which turns off any output macros, the internal form of the attribute is shown. This allows previously hidden fields of the attribute to be examined by the subterm navigation. Note that if the current subterm is inside a structure which will be changed by a changed output mode (such as inside the fd attribute), and the output mode is changed, then until the current subterm is moved out of the structure, the existing subterm path is still applicable.

Also, after a change in output modes, the current subterm will still be examining the structure that it obtained from the parent subterm. Consider the finite domain variable example again:

```
4..10000000
        INSPECT  (../2)   %> up subterm
[4..10000000]       ***** printed structure 1
        INSPECT  (list  1-head 2-tail)   %> <o>
current output mode is "QPm", toggle char: T
```

```
new output mode is "TQPm".
[4..10000000]
        INSPECT  (list  1-head 2-tail)   %> up subterm
A{[4..10000000]}
        INSPECT  (attributes  1-suspend 2-fd )   %> 2
fd(dom([4..10000000], 9999997), [], [], [])
        INSPECT  (struct fd/4)   %> <o>
current output mode is "QPmT", toggle char: T
new output mode is "QPm".
fd(4..10000000, [], [], [])    ***** printed structure 2
        INSPECT  (struct fd/4)   %>
```

Printed structures 1 and 2 in the above example are at the same position (toplevel of the finite domain structure), and printed with the same output mode (`QPm`), but are different because the structure obtained from the parent subterm is different – in printed structure 2, the output mode was not changed until after the `fd/4` structure was the current subterm.

15.6.7 Changing the Settings

The following commands allow to change the parameters which influence the way the tracing information is displayed or processed.

< *par* **set print depth**
Allows to modify the *print_depth*, i.e. the depth up to which nested argument terms are printed. Everything nested deeper than the specified depth is abbreviated as .... Note that the debugger has a private *print_depth* setting with default 5, which is different from the global setting obtained from **get_flag/2**.

> *par* **set indentation step width**
Allows to specify the number of spaces used to indent trace lines according to their depth level. The default is 0.

m **module**
Toggles the module printing in the trace line. If enabled, the module from where the procedure is called is printed in the trace line:

```
(1) 1 CALL  true   %> show module
(1) 1 CALL  eclipse : true   %>
```

o **output mode**
This command allows to modify the options used when printing trace lines. It first prints the current `output_mode` string, as obtained by **get_flag/2**, then it prompts for a sequence of characters. If it contains valid output mode flags, the value of these flags is then inverted. Typing an invalid character will display a list describing the different options. Note that this command affects the global setting of `output_mode`.

```
(1) 1 CALL  X is length([1, 2, ...])   %> current output mode
                                          is "QPm", toggle char: V
```

```
new output mode is "VQPm".
  (1) 1 CALL  X_72 is length([1, 2, ...])   %> current output mode
                                            is "QVPm", toggle char: O
new output mode is "OQVPm".
  (1) 1 CALL  is(X_72, length([1, 2, ...]))   %> current output mode
                                            is "OQVPm", toggle char: .
new output mode is ".OQVPm".
  (1) 1 CALL  is(X_72, length(.(1, .(2, .(...)))))   %>
```

+ **set a spy point**

Set a spy point on the displayed procedure, the same as using the **spy/1** predicate. It is possible to set a spy point on any existing procedure, even on a built-in on external one. If the procedure already has a spy point, an error message is printed and any counter is ignored.

Note that the debugger does not check for spy points that occur inside skipped procedures or during the execution of any other skip command than the *leap* command **l**.

− **remove a spy point**

Similarly to the previous command, this one removes a spy point from a procedure, if it has one.

15.6.8 Environment Commands

b **break**

This command is identical to the **break/0** call. A new top-level loop is started with the debugger switched off. The state of the database and the global settings is the same as in the previous top-level loop. After exiting the break level with ^D, or `end_of_file` the execution returns to the debugger and the last trace line is redisplayed.

N **nodebug permanently**

This command switches tracing off for the remainder of the execution as well as for subsequent top-level queries. It affects the global flag **debugging**, settin it to *nodebug*.

15.7 Extending the Debugger

15.7.1 User-defined Ports

The primitive **trace_port(?Invoc, +Port, +Term)** allows the programmer to create user-defined debugger ports, thus enhancing the debugger's basic box model with application-specific checkpoints. Calls to **trace_port/3** can be inserted in the code, and when the debugger is on, they will cause execution to stop and enter the debugger, displaying a trace line with the user-defined port and data.

The following example defines two special ports that indicate that values are selected for a boolean variable:

```
:- pragma(nodebug).
bool(B) :-
    trace_port(Invoc, 'TRY_ZERO', B=0),
        (
            B=0
        ;
            trace_port(Invoc, 'TRY_ONE', B=1),
            B=1
        ).
    :- untraceable bool/1.
```

In the execution, this will look as follows:

```
[eclipse 9]: bool(B).
(3) 3 TRY_ZERO  B = 0   %> creep

B = 0     More? (;)
(3) 3 TRY_ONE  B = 1   %> creep

B = 1
yes.
```

Note that both ports share the same invocation number, so the **i** command can be used to skip from one to the other.

15.8 Switching To Creep Mode With CTRL-C

When the debugger is on and a program is running, typing CTRL-C prompts for input of an option. The d-option switches the debugger to *creep* mode and continues executing the interrupted program. The debugger will then stop at the next port of the running program.

```
[eclipse 1]: debug.
Debugger switched on - leap mode
[eclipse 2]: repeat,fail.
^C

interruption: type a, b, c, d, e, or h for help : ? d
  (1) 1 *EXIT  repeat   %>
```

Chapter 16

Attributed Variables

16.1 Introduction

Attributed variable is a special ECL^{i}PS^{e}data type which represents a variable with some attached attributes. [1]

It is a powerful means to implement various extensions of the plain Prolog language. In particular, it allows the system's behaviour on unification to be customised. In most situations, an attributed variable behaves like a normal variable. E.g. it can be unified with other terms and **var/1** succeeds on it. The differences compared to a plain variable are:

- an attributed variable has a number of associated *attributes*
- the attributes are included in the module system
- when an attributed variable occurs in the unification and in some built-in predicates, each attribute is processed by a user-defined *handler*

16.2 Declaration

An attributed variable can have any number of attributes. The attributes are accessed by their name. Before an attribute can be created and used, it must be declared with the predicate **meta_attribute/2**. The declaration has the format

meta_attribute(Name, HandlerList)

Name is an atom denoting the attribute name and usually it is the name of the module where this attribute is being created and used. **HandlerList** is a (possibly empty) list of handler specifications for this attribute (see Section 16.7).

---

[1] We use both *metaterm* and *attributed variable* to denote the same data structure. The latter is more appropriate, but for the reasons of backward compatibility we sometimes use the former. The name "metaterm" originates from its application in meta-programming: for an object-level program, a metaterm looks like a variable; but the meta-program accesses the metaterm's attribute and can store meta-level information in it.

16.3 Syntax

The most general attributed variable syntax is

$Var\{Name_1 : Attr_1, Name_2 : Attr_2, \ldots, Name_n : Attr_n\}$

where the syntax of *Var* is like that of a variable, $Name_i$ are attribute names and $Attr_i$ are the values of the corresponding attributes. The expression **Var{Attr}** is a shorthand for **Var{Module:Attr}** where **Module** is the current module name. The former is called *unqualified* and the latter *qualified* attribute specification. As the attribute name is usually identical with the source module name, all occurrences of an attributed variable in the source module may use the unqualified specification.

If there are several occurrences of the same attributed variable in a single term, only one occurrence is written with the attribute, the others just refer to the name, e.g.

```
p(X, X{attr})
```

or

```
p(X{attr}, X)
```

both describe the same term, which has two occurrences of a single attributed variable with attribute `attr`. The following is a syntax error (even when the attributes are identical):

```
p(X{attr}, X{attr})
```

16.4 Creating Attributed Variables

A new attribute can be added to a variable using the tool predicate

add_attribute(Var, Attr).

An attribute whose name is not the current module name can be added using **add_attribute/3** which is its tool body predicate (exported in `sepia_kernel`). If **Var** is a free variable, it will be bound to a new attributed variable whose attribute corresponding to the current module is **Attr** and all its other attributes are free variables. If **Var** is already an attributed variable and its attribute is uninstantiated, it will be bound to **Attr**, otherwise the effect of this predicate will be the same as unifying **Var** with another attributed variable whose attribute corresponding to the current module is **Attr**.

16.5 Decomposing Attributed Variables

The attributes of an attributed variable can be accessed using one-way unification in a matching clause, e.g.

```
get_attribute(X{Name:Attribute}, A) :-
    -?->
    A = Attribute.
```

This clause succeeds only when the first argument is an attributed variable, and it binds `X` to the whole attributed variable and `A` to the attribute with name `Name`. Note that a normal (unification) clause can **not** be used to decompose an attributed variable (it would create a new attributed variable and unify this with the caller argument, but the unification is handled by an attributed variable handler, see Section 16.7).

16.6 Attribute Modification

Often an extension needs to modify the data stored in the attribute to reflect changes in the computation. The usual Prolog way to do this is by reserving one argument in the attribute structure for this next value. before accessing the most recent attribute value this chain of values has to be dereferenced until a value is found whose link is still free. A perfect compiler should be able to detect that the older attribute values are no longer accessed and it would compile these modifications using destructive assignment. Current compilers are unfortunately not able to perform this optimization (some systems can reduce these chains during garbage collection, but until this occurs, the list has to be dereferenced for each access and update). To avoid performance loss for both attribute updating and access, ECL^{i}PS^{e}provides a predicate for explicit attribute update **setarg/3** (which can be imported from the module **sepia_kernel**). **setarg(I, Term, NewArg)** will update the **I**'th argument of **Term** to be **NewArg**. Its previous value will be restored on backtracking.

Libraries which define user-programmable extensions like e.g. **fd.pl** usually define predicates that modify the attribute or a part of it, so that an explicit use of the **setarg/3** predicate is not necessary.

16.7 Attributed Variable Handlers

An attributed variable is a variable with some additional information which is ignored by ordinary *object level* system predicates. *Meta level* operations on attributed variables are handled by extensions which know the contents of their attributes and can specify the outcome of each operation. This mechanism is implemented using *attributed variable handlers*, which are user-defined predicates invoked whenever an attributed variable occurs in one of the predefined operations. The handlers are specified in the attribute declaration **meta_attribute(Name, HandlerList)**, the second argument is a list of handlers in the form

```
[unify:UnifyHandler, test_unify:TUHandler, ...]
```

Handlers for operations which are not specified or those that are **true/0** are ignored and never invoked. If **Name** is an existing extension, the specified handlers replace the current ones.

Whenever one of the specified operations detects an attributed variable, it will invoke all handlers that were declared for it and each of them receives either the whole attributed variable or its particular attribute as argument. The system does not check if the attribute that corresponds to a given handler is instantiated or not; this means that the handler must check itself if the attributed variable contains any attribute information or not. For instance, if an attributed variable *X{a:_, b:_, c:f(a)}* is unified with the attributed variable *Y{a:_, b:_, c:f(b)}*, the handlers for the attributes *a* and *b* should treat this as binding of two plain variables because their attributes were not involved. Only the handler for *c* has any work to do here. The library **suspend.pl** can be used as a template for writing attributed variable handlers.

The following operations invoke attributed variable handlers:

- **unify**: the usual unification. The handler procedure is

  unify_handler(+Term, ?Attribute)

  The first argument is the term that was unified with the attributed variable, it is either a nonvariable or an attributed variable. The second argument is directly the contents of the attribute slot corresponding to the extension, i.e. it is not the whole attributed variable. When this handler is invoked, the attributed variable is already bound to *Term*.

If an attributed variable is unified with a standard variable, the variable is bound to the attributed variable and no handlers are invoked. If an attributed variable is unified with another attributed variable or a non-variable, the attributed variable is bound (like a standard variable) to the other term **and** all handlers for the **unify** operation are invoked. Note that several attributed variable bindings can occur e.g. during a head unification and also during a single unification of compound terms. The handlers are only invoked at certain trigger points (usually before the next predicate call).

- **pre_unify**: this is also a handler which is invoked on normal unification, but it is called *before* the unification itself occurs. The handler procedure is

  pre_unify_handler(?AtrVar, +Term)

  The first argument is the attributed variable to be unfied, the second argument is the term it is going to be unified with. This handler is provided only for compatibility with SICStus Prolog and its use is not recommended, because it is less efficient than the **unify** handler and because its semantics is not quite clean, there may be cases where changes inside this handler may have unexpected effects.

- **test_unify**: the unification which is not supposed to trigger constraints propagation, it is used e.g. in the **not_unify/2** predicate. The handler procedure is

  test_unify_handler(+Term, ?Attribute)

  where the arguments are the same as for the unify handler. During the execution of the handler the attributed variable is bound to *Term*, however when all local handlers succeed, all bindings are undone.

- **compare_instances**: instance and variant tests. The handler procedure is

  instance_handler(-Res, ?TermL, ?TermR)

  and its arguments are similar to the **compare_instances/3** predicate. **Res** is the relation between the two terms *TermL* and *TermR*, it can be either = or < (the handler might directly fail if the result is >). All bindings made in the handler will be undone after processing the local handlers.

- **copy_term**: the **copy_term/2** predicate. The handler procedure is

  copy_handler(?Meta, ?Var)

  Meta is the attributed variable encountered in the copied term, *Var* is its corresponding variable in the copy. All extension handlers receive the same arguments. This means that if the attributed variable should be copied as an attributed variable, the handler must check if *Var* is still a free variable or if it was already bound to an attributed variable by a previous handler.

- **delayed_goals**: the **delayed_goals/2** predicate. The handler procedure is

  delayed_goals_handler(?Meta, ?GoalList, -GoalCont)

Meta is the attributed variable encountered in the term, *GoalList* is an open-ended list of all delayed goals in this attribute and *GoalCont* is the tail of this list.

- **delayed_goals_number**: the **delayed_goals_number/2** predicate. The handler procedure is

  delayed_goals_number_handler(?Meta, -Number)

  Meta is the attributed variable encountered in the term, *Number* is the number of delayed goals occurring in this attribute. Its main purpose is for the first-fail selection predicates, i.e. it should return the number of constraints imposed on the variable.

- **print**: attribute printing in **printf/2,3** when the **m** option is specified. The handler procedure is

  print_handler(?AttrVar, -Attribute)

  AttrVar is the attributed variable being printed, *Attribute* is the term which will be printed as a value for this attribute, prefixed by the attribute name. If no handler is specified for an attribute, it will not be printed.

16.7.1 Printing Attributed Variables

The different output predicates treat attributed variables differently. The **write/1** predicate just prints it as a variable, while **writeq/1** prints the whole attribute as well, so that the attributed variable can be read back. The **printf/2** predicate has two options to be combined with the **w** format: **M** forces the whole attributed variable to be printed together with all its attributes in the standard format, so that it can be read back in. With the **m** option the attributed variable is printed using the handlers defined for the **print** operation. If there is only one handled attribute, the attributed variable is printed as

X{Attr}

where *Attr* is the value obtained from the handler. If there are several handled attributes, all attributes are qualified like in

X{a:A, b:B, c:C}.

An attributed variable `X{m:a}` with **print** handler **=/2** can thus be printed in different ways, e.g.: [2]

```
printf("%w", [X{m:a}])   or write(X{m:a}):    X
printf("%vMw", [X{m:a}]) or writeq(X{m:a}):   _g246{suspend : _g242, m : a}
printf("%mw", [X{m:a}]):                      X{a}
printf("%Mw", [X{m:a}]):                      X{suspend : _g251, m : a}
printf("%Vmw", [X{m:a}]):                     X_g252{a}
```

Write macros for attributed variables are not allowed because one extension alone should not decide whether the other attributes will be printed or not.

---

[2] The attribute **suspend** is always present and defined by system coroutining.

16.8 Built-Ins and Attributed Variables

free(?Term) This type-checking predicate succeeds iff its argument is an ordinary free variable, it fails if it is an attributed variable.

meta(?Term) This type-checking predicate succeeds iff its argument is an attributed variable. For other type testing predicates an attributed variable behaves like a variable.

16.9 Examples of Using Attributed Variables

16.9.1 Variables with Enumerated Domains

As an example, let us implement variables of enumerable types using attributes. We choose to represent these variable as attributed variables whose attribute is a enum/1 structure with a list holding the values the variable may take, e.g.

```
X{enum([a,b,c])}
```

We have to specify now what should happen when such a variable is bound. This is done by writing a handler for the **unify** operation. The predicate unify_enum/2 defined below is this handler. Its first argument is the value that the attributed variable has been bound to, the second is the attribute that the bound attributed variable had (keep in mind that the system has already bound the attributed variable to the new value). We distinguish two cases:
First, the attributed variable has been bound to another attributed variable (1st clause of unify_enum/2). In this case, we form the intersection between the two lists of admissible values. If it is empty, we fail. If it contains exactly one value, we can instantiate the remaining attributed variable with this value. Otherwise, we bind it to a new attributed variable whose attribute represents the remaining admissible values.
Second, when the attributed variable has been bound to a non-variable, the task that remains for the handler is merely to check if this binding was admissible (2nd clause of unify_enum/2).

```
[eclipse 2]: module(enum).
warning: creating a new module in module(enum)
[enum 3]: [user].
:- meta_attribute(enum, [unify:unify_enum/2, print:print_enum/2]).
:- import setarg/3 from sepia_kernel.

% unify_enum(+Term, Attribute)
unify_enum(_, Attr) :-
    /*** ANY + VAR ***/
    var(Attr).                  % Ignore if no attribute for this extension
unify_enum(Term, Attr) :-
    compound(Attr),
    unify_term_enum(Term, Attr).

unify_term_enum(Value, enum(ListY)) :-
    nonvar(Value),              % The attributed variable was instantiated
    /*** NONVAR + META ***/
    memberchk(Value, ListY).
```

```
unify_term_enum(Y{AttrY}, AttrX) :-
    -?->
    unify_enum_enum(Y, AttrX, AttrY).

unify_enum_enum(_, AttrX, AttrY) :-
    var(AttrY),                         % no attribute for this extension
    /*** VAR + META ***/
    AttrX = AttrY.                      % share the attribute
unify_enum_enum(Y, enum(ListX), AttrY) :-
    nonvar(AttrY),
    /*** META + META ***/
    AttrY = enum(ListY),
     intersection(ListX, ListY, ListXY),
     ( ListXY = [Val] ->
            Y = Val
     ;
            ListXY \= [],
            setarg(1, AttrY, ListXY)
     ).

print_enum(enum(List), Attr) :-
    -?->
    Attr = List.
 user       compiled traceable 1188 bytes in 0.03 seconds

yes.
[enum 4]: A{enum([yellow, blue, white, green])}
                = B{enum([orange, blue, red, yellow])}.

A = B = A{[blue, yellow]}
yes.
[enum 5]: A{enum([yellow, blue, white, green])}
                = B{enum([orange, blue, red, black])}.

A = B = blue
yes.
[enum 6]: A{enum([yellow, blue, white, green])} = white.

A = white
yes.
[enum 7]: A{enum([yellow, blue, white, green])} = red.

no (more) solution.
```

Some further remarks on this code: The second clause of **unify_term_enum/2** is a **matching clause**, as indicated by the $-?->$ guard. A matching clause is the only way to decompose an attributed variable. Note that this clause matches only calls that have an attributed variable with nonempty **enum** attribute on the first argument position.

16.9.2 Coroutining

Another example of attributed variable use is the ECL^{i}PS^{e}implementation of coroutining. The corresponding code can be found in the libraries **suspend_simple.pl** and **suspend.pl**.

16.10 Attribute Specification

The **structures.pl** library (see User Manual, Appendix A) is used to define and access variable attributes and their arguments. This makes the code independent of the number of attributes and positions of their arguments. Wherever appropriate, the libraries described in this document describe their attributes in this way, e.g.

$$\mathbf{suspend\ with\ [inst : I, constrained : C, bound : B]}$$

says that the structure name is **suspend** and that it has (at least) three arguments with the corresponding names and macros defined in the **structures.pl** library can be used to access it.

Chapter 17

Advanced Control Features

17.1 Introduction

ECL^{i}PS^e provides a set of low-level primitives for suspending and waking goals. Explicit suspension handling together with the attributed variable data type represents an extremely powerful tool for implementing constraint propagation and similar algorithms. These primitives may not be the easiest ones to use, but they are instrumental in providing higher-level constraints with the desired operational behaviours.

17.2 The Resolvent

The term **resolvent** originates from Logic Programming. It is the set of all goals that need to be satisfied. The computation typically starts with a top-level goal, then gets successively transformed (by substituting goals that match a clause head with an instance of the clause body, ie. a sequence of sub-goals), and eventually terminates with one of the trivial goals **true** or **fail**. For example, given the program

```
p :- q,r.
q.
r :- q.
```

and the goal p, the resolvent goes through the following states before the goal is proven and the computation terminates:

```
p ----> q,r ----> r ----> q ----> {}
```

While in Prolog the resolvent is always processed from left to right, the resolvent in ECL^{i}PS^e is more structured, and can be manipulated in a much more flexible way. This is achieved by two basic mechanisms, **suspension** and **priorities**.

Suspended goals form the part of the resolvent which is currently not being considered. This is typically done when we know that we cannot currently infer any interesting information from them.

The remaining goals are ordered according to their **priority**. At any time, the system attempts to solve the most urgent subgoal first. ECL^{i}PS^e currently supports a fixed range of 12 different priorities, priority 1 being the most urgent and 12 the least urgent.

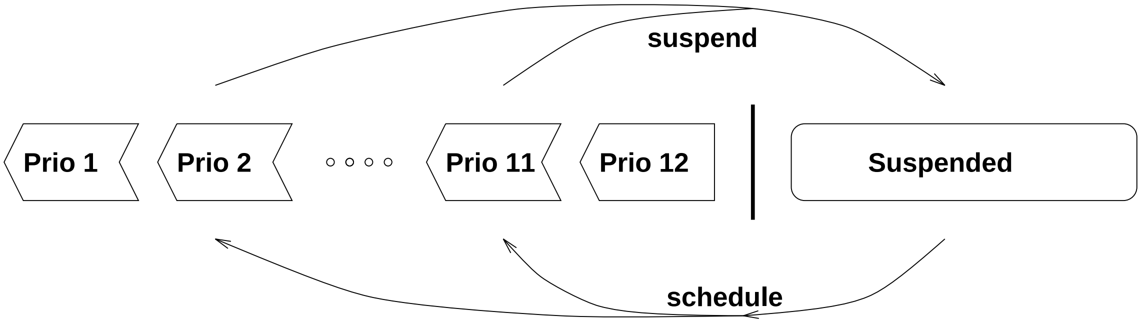


Figure 17.1: Structure of the resolvent

Figure 17.1 shows the structure of the resolvent. When a toplevel goal is launched, it has priority 12 and is the only member of the resolvent. As execution proceeds, active goals may be suspended, and suspended goals may be woken and scheduled with a particular priority.

17.3 Suspensions

ECL^{i}PS^e provides an opaque data type, called **suspension**, to represent suspended goals. Although usually a suspended goal waits for some waking condition in order to be reactivated, the primitives for suspension handling do not enforce this. To provide maximum flexibility of use, the functionalities of suspending and waking/scheduling are separated from the trigger mechanisms that cause the waking. We first describe the operations on suspensions.

17.3.1 What's in a Suspension?

A suspension represents a goal that is part of the resolvent. Apart from the goal structure proper, it holds information that is used for controlling its execution. The components of a suspension are:

The goal structure A term representing the goal itself, eg. $X > Y$.

The goal module The module from which the goal was called.

The goal priority The priority with which the goal will be scheduled when it becomes woken.

The state This indicates the current position of the suspension within the resolvent. It is either suspended (sleeping), scheduled or executed (dead).

Debugger data

17.3.2 Creating Suspended Goals

The most basic primitive to create a suspension is **make_suspension(Goal, Priority, Susp [, Module])** where **Goal** is the goal structure, **Priority** is a small integer denoting the priority with which the goal should be woken and **Susp** is the resulting suspension.
Note that usually **make_suspension/3,4** is not used directly, but implicitly via **suspend/3,4** (described below) which in addition attaches the suspension to a trigger condition.

A suspension which has not yet been scheduled for execution and executed, is called *sleeping*, a suspension which has already been executed is called *executed* or *dead* (since it disappears from the resolvent). A newly created suspension is always sleeping, however note that due to backtracking, an executed suspension can become sleeping again. Sometimes we use the term *waking*, which is less precise and denoted the process of both scheduling and eventual execution.

17.3.3 Operations on Suspensions

The following summarises the predicates that can be used to create, test, decompose and destroy suspensions.

make_suspension(Goal, Priority, Susp)

make_suspension(Goal, Priority, Susp, Module) Create a suspension **Susp** with a given priority from a given goal. The goal will subsequently show up as a *delayed goal.*

is_suspension(Susp) Succeeds if **Susp** is a sleeping or scheduled suspension, fails if it is a suspension that has been already executed.

type_of(S, goal) Succeeds if **S** is a suspension, no matter if it is sleeping, scheduled or executed.

get_suspension_data(Susp, Name, Value) Extract any of the information contained in the suspension: **Name** can be one of `goal`, `module`, `priority`, `state` or `invoc`. (**suspension_to_goal/3**] is an older version of this primitive which retrieves only goal and module).

set_suspension_data(Susp, Name, Value) The `priority` and `invoc` (debugger invocation number) fields of a suspension can be changed using this primitive. If the priority of a sleeping suspension is changed, this will only have an effect at the time the suspension gets scheduled. If the suspension is already scheduled, changing priority has no effect, except for future schedulings of demons (see 17.6).

kill_suspension(Susp) Convert the suspension **Susp** into an *executed* one, ie. remove the suspended goal from the resolvent. This predicate is meta-logical as its use may change the semantics of the program.

17.3.4 Examining the Resolvent

The system keeps track of all created suspensions and it uses this data e.g. in the built-in predicates **delayed_goals/1**, **suspensions/1**, **current_suspension/1**, **subcall/2** and to detect floundering of the query given to the ECL^{i}PS^e top-level loop. A query is said to *flounder* if it succeeds but leaves some unexecuted suspensions. In this case, the answer substitutions alone do not form a solution, the suspended goals have to be taken in to account as well.

17.4 Waking conditions for suspensions

The usual purpose of a suspension is to represent a suspended goal which is waiting for a particular event to occur. When this event occurs, the suspension is passed to the waking scheduler which puts it at the appropriate place in the priority queue of woken goals and as soon as it becomes first in the queue, the suspension is executed.

The event which causes a suspension to be woken is usually related to one or more variables, for example variable instantiation, or a modification of a variable's attribute. However, it is also possible to trigger suspension with symbolic events not related to any variable.
Suspensions which should be woken by the same event are grouped together in a *suspension list*. This is a normal Prolog list which contains suspensions. Suspension lists are either stored in an attribute of an attributed variable or attached to a symbolic trigger.

17.4.1 Attaching Suspensions to Variables

Suspensions are attached to variables by means of the attribute mechanism. For this purpose, a variable attribute needs to have one or more slots reserved for **suspension lists**. Suspensions can then be inserted into one or several of those lists using

insert_suspension(Vars, Susp, Index) Insert the suspension **Susp** into the **Index**'th suspension list of all attributed variables occurring in **Vars**. The current module specifies which of the attributes will be taken.

insert_suspension(Vars, Susp, Index, Module) Similar to **insert_suspension/3**, but it inserts the suspension into the attribute specified by **Module**.

For instance,

```
insert_suspension(Vars, Susp, inst of suspend, suspend)
```

inserts the suspension into the **inst**[1] list of the (system-predefined) **suspend** attribute of all variables that occur in **Vars**, and

```
insert_suspension(Vars, Susp, max of fd, fd)
```

would insert the suspension into the **max** list of the finite-domain attribute of all variables in **Vars**.
Note that both predicates find all attributed variables which occur in the general term **Vars** and for each of them, locate the attribute which corresponds to the current module or the **Module** argument respectively. This attribute must be a structure, otherwise an error is raised, which means that the attribute has to be initialised before calling **insert_suspension/4,3**. Finally, the **Index**'th argument of the attribute is interpreted as a suspension list and the suspension **Susp** is inserted at the beginning of this list. A more user-friendly interface to access suspension lists is provided by the **suspend/3** predicate.

User-defined Suspension Lists

Many important attributes and suspension lists are either provided by the suspend-attribute or by libraries like the Finite Domain library lib(fd). For those suspension lists, initialisation and waking is taken care of.
For the implementation of user-defined suspension lists, the following low-level primitives are provided:

init_suspension_list(+Position, +Attribute) Initialises argument Position of Attribute to an empty suspension list.

---

[1] We use the syntax provided by lib(structures): `inst of suspend` stands for a small integer indicating the number of the argument slot within the attribute structure.

merge_suspension_lists(+Pos1, +Attr1, +Pos2, +Attr2) Destructively appends the first suspension list (argument Pos1 of Attr1) to the end of the second (argument Pos2 of Attr2).

schedule_suspensions(+Position, +Attribute) Takes the suspension list on argument position **Position** within **Attribute**, and schedule them for execution. As a side effect, the suspension list within **Attribute** is updated, ie. suspensions which are no longer useful are removed destructively. See section 17.5 for more details on waking.

17.4.2 Attaching Suspensions to Global Triggers

A single suspension or a list of suspensions can be attached to a symbolic trigger by using **attach_suspensions(+Trigger, +Susps)**. A symbolic trigger can have an arbitrary name (an atom). To "pull the trigger" **schedule_suspensions(+Trigger)** is used which will submit all attached suspensions to the waking scheduler.

Postponed Goals

There is one system-defined trigger called **postponed**. It is provided as a way to postpone the triggering of a goal as much as possible. This trigger is pulled just before the end of certain encapsulated executions, like

- end of toplevel execution
- inside all-solution predicates (**findall/3**, **setof/3**)
- inside **min_max/2** and **minimize/2**

A suspension should be attached to the **postponed** trigger only when

- it might not have any other waking conditions left
- and it might at the same time have other waking conditions left that could make it fail during further execution
- and one does not want to execute it now, e.g. because it is known to succeed or re-suspend

An example is a goal that used to wait for modifications of the upper bound of an interval variable. If the variable gets instantiated to its upper bound, there is no need to wake the goal (since the bound has not changed), but the variable (and with it the waking condition) disappears and the goal may be left orphaned.

17.5 The Waking Mechanism

Suspended goals are woken by submitting at least one of the suspension lists in which they occur to the waking scheduler. The waking scheduler which maintains a global priority queue inserts them into this queue according to their priority (see figure 17.1).
A suspension list can be passed to the scheduler by either of the predicates **schedule_suspensions/1** or **schedule_suspensions/2**. A suspension which has been scheduled in this way and awaits its execution is called a *scheduled suspension*.
Note, however, that scheduling a suspension by means of **schedule_suspensions/1** or **schedule_suspensions/2** alone does not implicitly start the waking scheduler. Instead, execution

continues normally with the next goal in sequence after schedule_suspensions/1,2. The scheduler must be explicitly invoked by calling **wake/0**. Only then it starts to execute the woken suspensions.

The reason for having **wake/0** is to be able to schedule several suspension lists before the priority-driven execution begins[2].

17.6 Demon Predicates

A common pattern when implementing data-driven algorithms is the following:

```
report(X) :-
      suspend(report1(X), 1, X->constrained).

report1(X) :-
      var(X),
      suspend(report1(X), 1, X->constrained), % re-suspend
      writeln(constrained(X)).
report1(X) :-
      nonvar(X),
      writeln(instantiated(X)).   % die
```

Here we have a goal that keeps monitoring changes to its variables. To do so, it suspends on some or all of those variables. When a change occurs, it gets woken, does something, and re-suspends. The repeated re-suspending has two disadvantages: It can be inefficient, and the goal does not have a unique identifying suspension that could be easily referred to, because on every re-suspend a new suspension is created.

To better support this type of goals, ECL^{i}PS^e provides a special type of predicate, called a **demon**. A predicate is turned into a demon by annotating it with a **demon/1** declaration. A demon goal differs from a normal goal only in its behaviour on waking. While a normal goal disappears from the resolvent when it is woken, the demon remains in the resolvent. Delaratively, this corresponds to an implicit recursive call in the body of each demon clause. Or, in other words, the demon goal forks into one goal that remains in the suspended part of the resolvent, and an identical one that gets scheduled for execution.

With this functionality, our above example can be done more efficiently. One complication arises, however. Since the goal implicitly re-suspends, it now has to be explicitly killed when it is no longer needed. The easiest way to achieve this is to let it remember its own suspension in one of its arguments. This can then be used to kill the suspension when required:

```
% A demon that wakes whenever X becomes more constrained
report(X) :-
      suspend(report(X, Susp), 1, X->constrained, Susp).

:- demon(report/2).
report(X, _Susp) :-
      var(X),
      writeln(constrained(X)). % implicitly re-suspend
```

---

[2] This mechanism may be reconsidered in a future release

```
report(X, Susp) :-
      nonvar(X),
      writeln(instantiated(X)),
      kill_suspension(Susp).    % remove from the resolvent
```

17.7 More about Priorities

ECL^{i}PS^e uses an execution model which is based on goal *priorities* and which guarantees that a scheduled goal with a higher priority will be always executed before any goal with lower priority. Priority is a small integer number ranging from 1 to 12, 1 being the highest priority and 12 the lowest (cf. figure 17.1). All goals started from the ECL^{i}PS^e top-level loop or from the command line with the -e option have priority 12. Each suspension and each goal which is being executed therefore has an associated priority. The priority of the currently executing goal can be found out with **get_priority/1**.

Priority-based execution is driven by a scheduler: It picks up the scheduled suspension with the highest priority. If its priority is higher than the priority of the currently executing goal, then the execution of the current goal is interrupted and the new suspension is executed. This is repeated until there are no suspensions with priority higher than that of the current goal.

17.7.1 Changing Priority Explicitly

It is also possible to execute a goal with a given priority by means of **call_priority(Goal, Prio)** which calls **Goal** with the priority **Prio**. When a goal is called this way with high priority, it is effectively made atomic, ie. it will not be interrupted by goals that wake up while it executes. Those goals will all be deferred until exit from **call_priority/2**. This technique can sometimes improve efficiency. Consider for example the following program:

```
p(1).
report(Term) :- writeln(term=Term), suspend(report(Term),3,Term->inst).
```

and the execution

```
[eclipse 2]: report(f(X,Y,Z)), p(X),p(Y),p(Z).
term = f(X, Y, Z)
term = f(1, Y, Z)
term = f(1, 1, Z)
term = f(1, 1, 1)
```

report/1 is woken and executed three times, once for each variable binding. If instead we do the three bindings under high priority, it will only execute once after all bindings have already been done:

```
[eclipse 3]: report(f(X,Y,Z)), call_priority((p(X),p(Y),p(Z)), 2).
term = f(X, Y, Z)
term = f(1, 1, 1)
```

17.7.2 Choice of Priorities

Although the programmer is more or less free to specify which priorities to use, we strongly recommend to stick to the following scheme (from urgent to less urgent):

debugging (1) goals which don't contribute to the semantics of the program and always succeed, e.g. display routines, consistency checks or data breakpoints.

immediate goals which should be woken immediately and which do not do any bindings or other updates. Examples are quick tests which can immediately fail and thus avoid redundant execution.

quick fast deterministic goals which may propagate changes to other variables.

normal deterministic goals which should be woken after the **quick** class.

slow deterministic goals which require a lot of processing, e.g. complicated disjunctive constraints.

delayed nondeterministic goals or goals which are extremely slow.

toplevel goal (12) the default priority of the user program.

17.8 Printing Suspensions

The default format for the printing of a sleeping suspension is 'SUSP-_nn-susp':

```
[eclipse 2]: make_suspension(help(X), 2, S).
S = 'SUSP-_78-susp'
```

Scheduled and executed suspensions are by default printed as follows:

```
[eclipse 6]: make_suspension(true, 2, S), writeln(S),
             schedule_woken([S]), writeln(S).
'SUSP-_88-susp'
'SUSP-_88-sched'
S = 'SUSP-_88-dead'
```

The way suspensions are printed can be changed by defining a write macro for the type `goal`:

```
[eclipse 1]: [user].
 :- op(1090, fx, <), op(1100, xf, >).
 trs(S, <Goal>) :-
        suspension_to_goal(S, Goal, _).
 :- define_macro(type(goal), trs/2, [write]).
^D
yes.
[eclipse 2]: make_suspension(writeln(hello), 2, S).

S = < writeln(hello) >
```

17.9 The Standard suspend Attribute

ECL^{i}PS^e defines a standard attribute called **suspend** which provides a general coroutining mechanism which serves as a basis for various other extensions that use goal suspension and

waking. It defines an attribute and several suspension lists which are woken on very general conditions and which can be accessed by other extensions and also some general utility predicates.
The attribute has the name **suspend** and it is represented by the structure[3]

suspend with [inst : I, constrained : C, bound : B]

The lists have the following meaning:

- **inst** is woken only when the variable is instantiated, i.e. bound to a nonvariable.
- **bound** is woken whenever the variable is bound to another variable whose **suspend** attribute is not empty or when it is instantiated.
- **constrained** is woken whenever the variable becomes more constrained in the most general sense, including instantiation and binding to another variable which has a nonempty **suspend** attribute.

The **inst** and **bound** lists are woken implicitly by the unification routine. New extensions which want to be notified about these events are supposed to use this library instead of defining their own lists that should be woken on instantiation and binding.
The **constrained** list must be explicitly woken by every extension which imposes more constraints on the variable. For example, in case of the **fd** library this means that the domain of the variable was reduced, in the **r** library this is the case when a new equation containing this variable was inserted. As the system cannot itself decide whether the variable was constrained or not, it is the sole responsibility of each extension to notify the system of the constraining. This is done by calling the predicate

notify_constrained(Var)

whenever the variable **Var** becomes more constrained. As the **constrained** list is automatically woken when the variable is instantiated or bound, this predicate should be called only when the variable is constrained in an extension-specific way.
The suspend attribute is defined with handlers for the meta-term events

- **unify**
- **compare_instances**
- **delayed_goals**
- **delayed_goals_number**

As no printing handler is defined, the suspended lists are normally not visible.

17.9.1 The suspend/3,4 Predicates

The predicates **suspend/3** and **suspend/4** provide a simplified interface to suspend goals and wake them on specified conditions, which is easier to use than the low-level predicates **make_suspension/3** and **insert_suspension/3**. When

---

[3] We use the syntax provided by lib(structures).

suspend(Goal, Prio, CondList)

is called, *Goal* will be suspended with priority *Prio* and it will wake up as soon as one of the condition specified in the *CondList* is satisfied. This list contains specifications of the form

Vars – > Cond

to denote that as soon as one of the variables in the term *Vars* will satisfy the condition *Cond*, the suspended goal will be woken and then executed as soon as the program priority allows it. *CondList* can also be a single specification.
The condition *Cond* can be the name of a system-defined suspension list, e.g.

(X,Y) – > inst

means that as soon as one (or both) of the variables **X, Y** will be instantiated, the suspended goal will be woken. These variables are also called the *suspending variables* of the goal.
Cond can also be the specification of a suspension list defined in one of currently available attributes. E.g. when the finite domain library is loaded

f(A, B) – > fd:min

triggers the suspended goal as soon as the minimum element of the domains of **A** or **B** are updated (see Library Manual, Finite Domain Library).
Another admissible form of condition *Cond* is

trigger(Name)

which suspends the goal on the global trigger condition **Name** (see section 17.4.2).

17.9.2 Particularities of Waking by Unification

Goals that are suspended on the **inst** and **bound** lists of the **suspend** attribute are woken by unifications of their *suspending variables*. One suspending variable can be responsible for delaying several goals, on the other hand one goal can be suspended due to several suspending variables. This means that when one suspending variable is bound, several delayed goals may be woken. The order of waking suspended goals does not necessarily correspond to the order of their suspending. It is in fact determined by their priorities and is implementation-dependent within the same priority group.
The waking process never interrupts the unification and/or a sequence of simple goals. Simple goals are a subset of the built-ins and can be recognised by their **call_type** flag as returned by **get_flag/3**, simple goals having the type `external`. Note also that some predicates, e.g. **is/2**, are in-line expanded in the **nodbgcomp** mode and thus simple, whereas in **dbgcomp** mode they are *regular*.
ECL^{i}PS^e treats simple predicates (including unification) always as a block. Delayed goals are therefore woken only at the end of a successful unification and/or a sequence of simple goals. If a suspending variable is bound in a simple goal, the suspended goals are woken only at the end of the last consecutive simple goal or at the clause end. If the clause contains some simple goals at the beginning of its body, they are considered part of the head (*extended head*, or *prefix*) and if a suspending variable is bound in the head unification or in a simple predicate in the extended head, the corresponding delayed goals are woken at the end of the extended head.

A **cut** is always executed **before** waking any pending suspended goals. This is important especially in the situations where the cut acts like a guard, immediately after the clause neck or after a sequence of simple goals. If the goals woken by the head unification or by the extended head are considered as constraints on the suspending variables, the procedure will not behave as expected. For example

```
delay filter(_, Li, Lo) if var(Li), var(Lo).
filter(P,[],[]) :- !.
filter(P,[N|LI],[N|NLI]) :-
        N mod P =\= 0,
        !,
        filter(P,LI,NLI).
filter(P,[N|LI],NLI) :-
        N mod P =:= 0,
        filter(P,LI,NLI).

delay integers(_, List) if var(List).
integers(_, []).
integers(N, [N|Rest]) :-
        N1 is N + 1,
        integers(N1, Rest).

:- integers(2, L), filter(2, L, [N|R])
```

Here the call to **integers/2** delays, when **filter/3** is called, L is instantiated in the head unification, but **integers/2** will not be woken until the cut is executed, instead of waking it immediately and proceed to the third clause of **filter/3** after the woken goal fails. Therefore the call to =\\= also delays and the cut cuts the third clause. Only then the call to **integers/2** is woken, it fails and the whole goal fails instead of executing the third clause as expected. This is yet another example why cut should not be used together with coroutining. Note also that the ECL^{i}PS^e debugger will emit a warning due to the improper use of the cut.

The reason why delayed goals are woken **after** the cut and not before it is that neither of the two possibilities is always the intended or the correct one, however when goals are woken **before** the cut, there is no way to escape it and wake them after, and so if a nondeterministic goal is woken, it is committed by this cut which was most probably not intended. On the other hand, it is always possible to force waking before the cut by inserting a regular goal before it, for example **true/0**, so the sequence

true, !

can be viewed as a special cut type.

The example above should be correctly written using the if-then-else construct, which always forces waking suspended goals before executing the condition. This would also save trailing the second argument and creating a choice point:

```
filter(P,[],[]) :- !.
filter(P,[N|LI],LL) :-
        (N mod P =\= 0 ->
                LL = [N|NLI],
                filter(P, LI, NLI)
        ;
                filter(P,LI,LL)
        ).
```

17.10 The Top-Level Loop

The top-level loop normally reads the query from the user, executes it and prints the answer bindings. In case that a subgoal remains delayed[4], the answer bindings alone are not logically correct, they have to be considered in conjunction with the remaining delayed goals. To signal this, the top-level loop first prints the answer bindings and then it checks whether some goals are still delayed. If so, an exception is raised; the default handler for this exception prints the goals that are still delayed, so that the user is notified about this situation. This handler can as well print only the number of delayed goals, or ask the user for some action, e.g. force waking of some or all delayed goals.

17.11 The Cut and the Suspended Goals

It is very important to mention here the influence of non-logical predicates, especially the **cut**, on the execution of delayed goals. The cut relies on a fixed order of goal execution in that it discards some choice points if all goals preceding it in the clause body have succeeded. If some of these goals are delayed, or if the head unification of the clause with the cut wakes some nondeterministic delayed goals, the completeness of the resulting program is lost and there is no clean way to save it as long as the cut is used.
The user is strongly discouraged to use non-local cuts together with coroutining, or to be precisely aware of their scope. The danger of a cut is twofold:

- a cut can be executed if some calls preceding it in the clause (or children of these calls) delay and they fail when they are woken
- the head unification of a clause with cuts can wake some delayed goals. If they are nondeterministic, the cut in the body of the waking clause will commit even the woken goals

In order to detect these situations, the ECL^{i}PS^e debugger has an option to print a warning whenever a cut in one of the above two conditions is executed. These warnings can be toggled using the **P** command.

17.12 Delaying of Built-In Predicates

A number of built-in predicates can be configured to delay when their arguments are not sufficiently instantiated. This behaviour uses the more general mechanism introduced above and is enabled with

```
:- set_flag(coroutine, on).
```

---

[4] This is sometimes referred to as *floundering*.

As a consequence, these predicates will no longer raise instantiation faults, but delay instead. For example:

```
[eclipse 1]: X > 0.
instantiation fault in X > 0
[eclipse 2]: set_flag(coroutine, on).
yes.
[eclipse 3]: X > 0.

X = X
Delayed goals:
        X > 0
yes.
```

The delayed predicate will be executed as soon as the responsible variable is instantiated:

```
[eclipse 4]: X > 0, X=1.

X = 1
yes.
```

The following built-ins have this property:

</2 **>/2** **>=/2** **=</2**
=:=/2 **=\=/2** **plus/3** **times/3**.
is/2 **functor/3** **arg/3** **=../2**
atom_string/2 **number_string/2** **integer_atom/2** **char_int/2**
string_list/2 **atom_length/2** **string_length/2** **concat_atoms/3**
concat_atom/2 **concat_strings/3** **concat_string/2**

17.13 Obsolete Suspension Facilities

The facilities presented in this section are largely superseded by the more general mechanism described earlier in this chapter. However they are still supported and the information given here may give useful hints for porting or improving existing code.

17.13.1 Delay Clauses

For delaying calls to user-defined Prolog predicates, ECL^{i}PS^e provides *delay clauses*. Delay clauses are a declarative means (they are in fact meta-clauses) to specify the conditions under which the predicate should delay. The semantics of delay clauses is thus cleaner than many alternative approaches to delay primitives.
A delay clause is very similar to a normal Prolog clause. It has the form

```
delay <Head> if <Body>.
```

A Prolog predicate may have one or more delay clauses. They have to be textually **before** and **consecutive** with the normal clauses of the predicate they belong to. The simplest example for a delay clause is one that checks if a variable is instantiated:

```
delay report_binding(X) if var(X).
report_binding(X) :- printf("Variable has been bound to %w\n", [X]).
```

The operational semantics of the delay clauses is as follows: when a procedure with some delay clauses is called, then the delay clauses are executed before executing the procedure itself. If one of the delay clauses succeeds, the call is suspended, otherwise they are all tried in sequence and, if all delay clauses fail, the procedure is executed as usual.

The mechanism of executing a delay clause is similar to normal Prolog clauses with two exceptions:

- the unification of the call with the delay clause head is not the usual Prolog unification, but rather unidirectional pattern matching. This means that the variables in the call must not be bound by the matching, if such a binding would be necessary to perform the unification, it will fail instead. E.g. the head of the delay clause

  ```
  delay p(a, X) if var(X).
  ```

  does not match the call **p(A, b)** but it matches the goal **p(a, b)**.

- the delay clauses are deterministic, they leave no choice points. If one delay clause succeeds, the call is delayed and the following delay clauses are not executed. As soon as the call is resumed, all delay clauses that may succeed are re-executed.

The reason for using pattern matching instead of unification is to avoid a possible mixing of meta-level control with the object level, similarly to [3].

If the matching succeeds, the body of the delay clause is executed. If all the body subgoals succeed, the call is suspended. Otherwise, or if the head matching fails, the next delay clause is tried and if there is none, the call continues normally without suspending.

The form of the head of a delay clause is not restricted. For the body, the following conditions hold:

- the body subgoals must not bind any variable in the call and they must not delay themselves. The system does not verify these conditions currently.

- it should contain at least one of the following subgoals:

  - **var/1**
  - **nonground/1**
  - **nonground/2**
  - **\==/2**

  If this is not the case, then the predicate may delay without being linked to a variable, so it delays forever and cannot be woken again. The experience shows that the former four primitives suffice to express any usual conditions.

CAUTION: when the symbol **:-** is used instead of **if** in the delay clause, the resulting term is a clause of the procedure **delay/1**. To ease the detection of such mistakes, the procedure delay/1 is protected and so an exception is raised.

More Examples

- A predicate that checks if its argument is a proper list of integers. The delay conditions specify that the predicate should delay if the list is not terminated or if it contains variable elements. This makes sure that it will never generate list elements, but only acts as a test:

  ```
  delay integer_list(L) if var(L).
  delay integer_list([X|_]) if var(X).
  integer_list([]).
  integer_list([X|T]) :- integer(X), integer_list(T).
  ```

- Delay if the first two arguments are identical and the third is a variable:

  ```
  delay p(X, X, Y) if var(Y).
  ```

- Delay if the argument is a structure whose first subterm is not ground:

  ```
  delay p(X) if compound(X), arg(1, X, Y), nonground(Y).
  ```

- Delay if the argument term contains 2 or more variables:

  ```
  delay p(X) if nonground(2, X).
  ```

- The predicate as a delaying condition is useful mainly The \ **==/2** in predicates like X + Y = Z which need not be delayed if X == Z. Y can be directly bound to 0, provided that X is later bound to a number (or it is not bound at all) The condition **X \== Y** makes sense only if X or Y are nonground: a delay clause

  ```
  delay p(X, Y) if X \== Y.
  ```

  executed with the call ?- p(a, b) of course succeeds and the call delays forever, since no variable binding can wake it.

17.13.2 Simulating other delay primitives with delay clauses

It is relatively easy to simulate similar constructs from other systems by using delay clauses, for example, MU-Prolog's *sound negation* predicate ~/1 can be in ECL^{i}PS^e simply implemented as

```
delay ~ X if nonground(X).
~ X :- \+ X .
```

MU-Prolog's *wait declarations* can be in most cases simulated using delay clauses. Although it is not possible to convert all wait declarations to delay clauses, in the real life examples this can usually be achieved. The *block* declarations of SICStus Prolog can be easily expressed as delay clauses with **var/1** and **nonground/1** conditions. The *freeze/2* predicate (e.g. from SICStus Prolog, same as *geler/2* in Prolog-II) can be expressed as

```
delay freeze(X, _) if var(X).
freeze(_, Goal) :- call(Goal).
```

The transcription of *when declarations* from NU_Prolog basically involves negating them: for instance, the when declarations

```
?- flatten([], _) when ever.
?- flatten(A._, _) when A.
```

can be rewritten as

```
delay flatten(A, _) if var(A).
delay flatten([A|_], _) if var(A).
```

Note however, that in contrast to when declarations, there are no syntactic restrictions on the head of a delay clause, in particular it can contain any compound terms and repeated variables. The semantics of the delay clauses is also clearer than it is the case for other comparable constructs - by defining when the call has to *delay* the user naturally expresses the necessary condition. If the user specifies when the call should *not* be delayed, this condition is no longer quite straightforward - if there is no condition or if the condition does not match the call it would mean that the call should wait forever, which is certainly not the intended semantics.

Chapter 18

More About Suspension

The fundamentals of goal suspension and waking were described in the previous chapter. This chapter looks at some applications and examples in greater detail.

18.1 Waiting for Instantiation

Goals that are to be woken when one or more variables become instantiated use the **inst** list. For instance, a predicate **freeze(Term, Goal)** which delays and is woken as soon as any variable in **Term** becomes instantiated can be implemented as follows:

```
freeze(Term, Goal) :-
    suspend(Goal, 3, Term->inst).
```

or equivalently by

```
freeze(Term, Goal) :-
    make_suspension(Goal, 3, Susp),
    insert_suspension(Term, Susp, inst of suspend, suspend).
```

When it is called with a nonground term, it produces a delayed goal and when one variable is instantiated, the goal is woken:

```
[eclipse 2]: freeze(X, write(hello)).

X = X

Delayed goals:
        write(hello)
yes.
[eclipse 3]: freeze(p(X, Y), write(hello)), X=Y.

X = X
Y = X

Delayed goals:
        write(hello)
```

```
yes.
[eclipse 4]: freeze(p(X, Y), write(hello)), Y=1.
hello
X = X
Y = 1
yes.
```

However, if its argument is ground, it will still produce a suspended goal which may not be what we expect:

```
[eclipse 5]: 8.
freeze(a, write(hello)).



Delayed goals:
        write(hello)
yes.
```

To correct this problem, we can test this condition separately:

```
freeze(Term, Goal) :-
    nonground(Term),
    !,
    suspend(Goal, 3, Term->inst).
freeze(_, Goal) :-
    call(Goal).
```

and get the expected results:

```
[eclipse 8]: freeze(a, write(hello)).
hello
yes.
```

Another possibility is to wait until a term becomes ground, i.e. all its variables become instantiated. In this case, it is not necessary to attach the suspension to *all* variables in the term. The **Goal** has to be called when the last variable in **Term** is instantiated, and so we can pick up any variable and attach the suspension to it. We may then save some unnecessary waking when other variables are instantiated before the selected one. To select a variable from the term, we can use the predicate **term_variables/2** which extracts all variables from a term. However, when we already have all variables available, we can in fact dispose of **Term** which may be huge and have a complicated structure. Instead, we pick up one variable from the list until we reach its end:

```
wait_for_ground(Term, Goal) :-
    term_variables(Term, VarList),
    wait_for_var(VarList, Goal).

wait_for_var([], Goal) :-
    call(Goal).
wait_for_var([X|L], Goal) :-
```

```
    (var(X) ->
        suspend(wait_for_var([X|L], Goal), 3, X->inst).
    ;
    nonground(X) ->
        term_variables(X, Vars),
        append(Vars, L, NewVars),
        wait_for_var(NewVars, Goal)
    ;
        wait_for_var(L, Goal)
    ).
```

We can use the debugger to see when a goal is delayed:

```
[eclipse 18]: wait_for_ground(p(A, B, C), write(hello)),
        C = 1, A = 1, B = 1.
  (1) 0  CALL   wait_for_ground(p(A, B, C), write(hello)) (dbg)?- skip
  (1) 0  EXIT   wait_for_ground(p(A, B, C), write(hello)) (dbg)?- delayed goals

Delayed goals:
(5) wait_for_var([B, A], write(hello))

  (1) 0  EXIT   wait_for_ground(p(A, B, C), write(hello)) (dbg)?- creep
C (6) 0  CALL   C = 1 (dbg)?- creep
C (6) 0  EXIT   1 = 1 (dbg)?- creep
  (5) 0  RESUME wait_for_var([B, A], write(hello)) (dbg)?- skip
  (5) 0  EXIT   wait_for_var([B, A], write(hello)) (dbg)?- creep
C (9) 0  CALL   A = 1 (dbg)?- delayed goals

Delayed goals:
(8) wait_for_var([A], write(hello))

C (9) 0  CALL   A = 1 (dbg)?- creep
C (9) 0  EXIT   1 = 1 (dbg)?- creep
C (10) 0  CALL   B = 1 (dbg)?- creep
C (10) 0  EXIT   1 = 1 (dbg)?- creep
  (8) 0  RESUME wait_for_var([1], write(hello)) (dbg)?- skip
hello  (8) 0  EXIT   wait_for_var([1], write(hello)) (dbg)?- creep

C = 1
A = 1
B = 1
yes.
```

18.2 Waiting for Binding

Sometimes we want a goal to be woken when a variable is bound to another one, e.g. to check for subsumption or disequality. As an example, let us construct the code for the built-in predicate

~= /2 . This predicate imposes the disequality constraint on its two arguments. It works as follows:

1. It scans the two terms. If they are identical, it fails.
2. If it finds a pair of different arguments at least one of which is a variable, it suspends. If both arguments are variables, the suspension is placed on the **bound** suspended list of both variables. If only one is a variable, the suspension is placed on its **inst** list, because in this case the constraint may be falsified only if the variable is instantiated.
3. Otherwise, if it finds a pair of arguments that cannot be unified, it succeeds.
4. Otherwise it means that the two terms are equal and it fails.

The code looks as follows. **equal_args/3** scans the two arguments. If it finds a pair of unifyable terms, it returns them in its third argument. Otherwise, it calls **equal_terms/3** which decomposes the two terms and scans recursively all their arguments.

```
dif(T1, T2) :-
    (equal_args(T1, T2, Vars) ->
        (nonvar(Vars) ->
            (Vars = inst(V) ->
                suspend(dif(T1, T2), 3, V->inst)
            ;
                suspend(dif(T1, T2), 3, Vars->bound)
            )
        ;
            fail     % nothing to suspend on, they are identical
        )
    ;
        true         % the terms are different
    ).

equal_args(A1, A2, Vars) :-
    (A1 == A2 ->
        true
    ;
    var(A1) ->
        (var(A2) ->
            Vars = bound(A1, A2)
        ;
            Vars = inst(A1)
        )
    ;
    var(A2) ->
        Vars = inst(A2)
    ;
        equal_terms(A1, A2, Vars)
    ).
```

```
equal_terms(R1, R2, Vars) :-
    R1 =.. [F|Args1],
    R2 =.. [F|Args2],
    equal_lists(Args1, Args2, Vars).

equal_lists([], [], _).
equal_lists([X1|A1], [X2|A2], Vars) :-
    equal_args(X1, X2, Vars),
    (nonvar(Vars) ->
        true     % we have already found a variable
    ;
        equal_lists(A1, A2, Vars)
    ).
```

Note that **equal_args/3** can yield three possible outcomes: success, failure and delay. Therefore, if it succeeds, we have to make the distinction between a genuine success and delay, which is done using its third argument. The predicate **dif/2** behaves exactly as the built-in predicate **~= /2**:

```
[eclipse 26]: dif(X, Y).

X = X
Y = Y

Delayed goals:
        dif(X, Y)
yes.
[eclipse 27]: dif(X, Y), X=Y.

no (more) solution.
[eclipse 28]: dif(X, Y), X=f(A, B), Y=f(a, C), B=C, A=a.

no (more) solution.
[eclipse 29]: dif(X, Y), X=a, Y=b.

X = a
Y = b
yes.
```

Note also that the scan stops at the first variable being compared to a different term. In this way, we scan only the part of the terms which is absolutely necessary to detect failure – the two terms can become equal only if this variable is bound to a matching term.
This approach has one disadvantage, though. We always wake the **dif/2** call with the original terms as arguments. Each time the suspension is woken, we scan the two terms from the beginning and thus repeat the same operations. If, for instance, the compared terms are lists with thousands of elements and the first 10000 elements are ground, we spend most of our time checking them again and again.

The reason for this handling is that the system cannot suspend the execution of **dif/2** while executing its subgoals: it cannot freeze the state of all the active subgoals and their arguments. There is however a possibility for us to do this explicitly: as soon as we find a variable, we stop scanning the terms and return a list of continuations for all ancestor compound arguments. In this way, **equal_args** returns a list of pairs and their continuations which will then be processed step by step:

- **equal_args/4** scans again the input arguments. If it finds a pair of unifyable terms, it inserts it into a difference list.
- **equal_lists/4** processes the arguments of compound terms. As soon as a variable is found, it stops looking at following arguments but it appends them into the difference list.
- **diff_pairs/2** processes this list. If it finds an identical pair, it succeeds, the two terms are different. Otherwise, it suspends itself on the variables in the matched pair (here the suspending is simplified to use only the **bound** list).
- The continuations are just other pairs in the list, so that no special treatment is necessary.
- When the variables suspended upon are instantiated to compound terms, the new terms are again scanned by **equal_arg/4**, but the new continuations are prepended to the list. As a matter of fact, it does not matter if we put the new pairs at the beginning or at the end of the list, but tracing is more natural when we use the fifo format.
- If this list of pairs is exhausted, it means that no potentially non-matching pairs were found, the two terms are identical and thus the predicate fails. note that this is achieved by a matching clause for **diff_pairs/2** which fails if its first argument is a free variable.
- Note the optimisation for lists in **equal_terms/4**: If one term is a list, we pass it directly to **equal_lists/4** instead of decomposing each element with **functor/3**. Obviously, this optimisation is applicable only if the input terms are known not to contain any pairs which are not proper lists.

```
dif2(T1, T2) :-
    equal_args(T1, T2, List, Link),
    !,
    diff_pairs(List, Link).
d2if(_, _).                 % succeed if already different

equal_args(A1, A2, L, L) :-
    A1 == A2,
    !.
equal_args(A1, A2, [A1-A2|Link], Link) :-
    (var(A1);var(A2)),
    !.
equal_args(A1, A2, List, Link) :-
    equal_terms(A1, A2, List, Link).

equal_terms(T1, T2, List, Link) :-
    T1 = [_|_],
        T2 = [_|_],
        !,
        equal_lists(T1, T2, List, Link).
    equal_terms(T1, T2, List, Link) :-
        T1 =.. [F|Args1],
        T2 =.. [F|Args2],
        equal_lists(Args1, Args2, List, Link).

    equal_lists([], [], L, L).
    equal_lists([X1|A1], [X2|A2], List, Link) :-
        equal_args(X1, X2, List, L1),
        (nonvar(List) ->
            L1 = [A1-A2|Link]
        ;
            equal_lists(A1, A2, L1, Link)
        ).

    diff_pairs([A1-A2|List], Link) :-
        -?->
        (A1 == A2 ->
            diff_pairs(List, Link)
        ;
        (var(A1); var(A2)) ->
            suspend(diff_pairs([A1-A2|List], Link), 3, A1-A2->bound)
        ;
        equal_terms(A1, A2, NewList, NewLink) ->
            NewLink = List,             % prepend to the list
            diff_pairs(NewList, Link)
        ;
            true
        ).
```

Now we can see that compound terms are processed up to the first potentially matching pair and then the continuations are stored:

```
[eclipse 30]: dif2(f(g(X, Y), h(Z, 1)), f(g(A, B), h(2, C))).

X = X
...
Delayed goals:
        diff_pairs([X - A, [Y] - [B], [h(Z, 1)] - [h(2, C)]|Link], Link)
yes.
```

When a variable in the first pair is bound, the search proceeds to the next pair:

```
[eclipse 31]: dif2(f(g(X, Y), h(Z, 1)), f(g(A, B), h(2, C))), X=A.

Y = Y
```

```
...
Delayed goals:
        diff_pairs([Y - B, [] - [], [h(Z, 1)] - [h(2, C)]|Link], Link)
yes.
```

dif2/2 does not do any unnecessary processing, so it is asymptotically much better than the built-in **~= /2**.

This predicate, however, can be used only to *impose* a constraint on the two terms (i.e. it is a *tell* constraint only). It uses the approach of *eager failure* and *lazy success*. Since it does not process the terms completely, it sometimes does not detect success:

```
[eclipse 55]: dif2(f(X, a), f(b, b)).

X = X


Delayed goals:
        diff_pairs([X - b, [a] - [b]|Link], Link)
yes.
```

If we wanted to write a predicate that suspends if and only if the disequality cannot be decided, we have to use a different approach. The easiest way would be to process both terms completely each time the predicate is woken. There are, however, better methods. We can process the terms once when the predicate **dif/2** is called, filter out all possibly matching pairs and then create a suspension for each of them. As soon as one of the suspensions is woken and it finds an incompatible binding, the **dif/2** predicate can succeed. There are two problems:

- How to report the success? There are N suspensions and each of them may be able to report success due to its bindings. All others should be disposed of.

  This can be solved by introducing a new variable which will be instantiated when the two terms become non-unifyable. Any predicate can then use this variable to ask or wait for the result. At the same time, when it is instantiated, all suspensions are woken and finished.

- How to find out that the predicate has failed? We split the whole predicate into N independent suspensions and only if all of them are eventually woken and they find identical pairs, the predicate fails. Any single suspension does not know if it is the last one or not.

  To cope with this problem, we can use the *short circuit* technique: Each suspension will include two additional variables, the first one being shared with the previous suspension and the second one with the next suspension. All suspensions are thus chained with these variables. The first variable of the first suspension is instantiated at the beginning. When a suspension is woken and it finds out that its pair of matched terms became identical, it binds those additional variables to each other. When all suspensions are woken and their pairs become identical, the second variable of the last suspension becomes instantiated and this can be used for notification that the predicate has failed.

```
dif3(T1, T2, Yes, No) :-
    compare_args(T1, T2, no, No, Yes).
compare_args(_, _, _, _, Yes) :-
    nonvar(Yes).
compare_args(A1, A2, Link, NewLink, Yes) :-
    var(Yes),
    (A1 == A2 ->
        Link = NewLink            % short-cut the links
    ;
    (var(A1);var(A2)) ->
        suspend(compare_args(A1, A2, Link, NewLink, Yes), 3,
    [[A1|A2]->bound, Yes->inst])
    ;
        compare_terms(A1, A2, Link, NewLink, Yes)
    ).

compare_terms(T1, T2, Link, NewLink, Yes) :-
    T1 =.. [F1|Args1],
    T2 =.. [F2|Args2],
    (F1 = F2 ->
        compare_lists(Args1, Args2, Link, NewLink, Yes)
    ;
        Yes = yes
    ).

compare_lists([], [], L, L, _).
compare_lists([X1|A1], [X2|A2], Link, NewLink, Yes) :-
    compare_args(X1, X2, Link, L1, Yes),
    compare_lists(A1, A2, L1, NewLink, Yes).
```

The variable **Yes** is instantiated as soon as the constraint becomes true. This will also wake all pending suspensions which then simply succeed. The argument **No** of **dif3/4** becomes instantiated to `no` as soon as all suspensions are woken and their matched pairs become identical:

```
[eclipse 12]: dif3(f(A, B), f(X, Y), Y, N).

Y = Y
...

Delayed goals:
        compare_args(A, X, no, L1, Y)
        compare_args(B, Y, L1, N, Y)
yes.
[eclipse 13]: dif3(f(A, B), f(X, Z), Y, N), A = a, X = b.

Y = yes
N = N
...
yes.
[eclipse 14]: dif3(f(A, B), f(X, Z), Y, N), A=X, B=Z.
```

```
Y = Y
N = no
...
yes.
```

Now we have a constraint predicate that can be used both to impose disequality on two terms and to query it. For instance, a condition "if T1 = T2 then X = single else X = double" can be expressed as

```
cond(T1, T2, X) :-
    dif3(T1, T2, Yes, No),
    cond_eval(X, Yes, No).

cond_eval(X, yes, _) :- -?->
    X = double.
cond_eval(X, _, no) :- -?->
    X = single.
cond_eval(X, Yes, No) :-
    var(Yes),
    var(No),
    suspend(cond_eval(X, Yes, No), 2, Yes-No->inst).
```

This example could be further extended, e.g. to take care of shared variables, occur check or propagating from the answer variable (e.g. imposing equality on all matched argument pairs when the variable **Y** is instantiated). We leave this as a (rather advanced) exercise to the reader.

18.3 Waiting for other Constraints

The **constrained** list in the **suspend** attribute is used for instance in generic predicates which have to be notified about the possible change of the state of a variable, especially its unifyability with other terms. Our example with the **dif** predicate could be for instance extended to work with finite domain or other constrained variables. The modification is fairly simple:

- When a variable in one term is matched against a subterm of the other term, it might not necessarily be unifyable with it, because there might be other constraints imposed on it. Therefore, **not_unify/2** must be used to test it explicitly.
- The suspension should be woken not only on binding, but on any constraining and thus the **constrained** list has to be used.

The predicate **compare_args/5** is thus changed as follows:

```
compare_args(_, _, _, _, Yes) :-
    nonvar(Yes).
compare_args(A1, A2, Link, NewLink, Yes) :-
    var(Yes),
    (A1 == A2 ->
        Link = NewLink
    ;
    (var(A1);var(A2)) ->
        (not_unify(A1, A2) ->
            Yes = yes
        ;
            suspend(compare_args(A1, A2, Link, NewLink, Yes), 3,
[[A1|A2]->constrained, Yes->inst])
        )
    ;
        compare_terms(A1, A2, Link, NewLink, Yes)
    ).
```

Now our **dif3/4** predicate yields correct results even for constrained variables:

```
[eclipse 1]: dif3(A, B, Y, N), A::1..10, B::20..30.

Y = yes
N = N
A = A{[1..10]}
B = B{[20..30]}
yes.
[eclipse 2]: dif3(A, B, Y, N), A::1..10, B = 5, A ## 5.

Y = yes
N = N
B = 5
A = A{[1..4, 6..10]}
yes.
[eclipse 18]: dif3(A, B, Y, N), A + B $= 1, A $= 1/2.

Y = Y
N = no
B = 1 / 2
A = 1 / 2
yes.
```

Chapter 19

Memory Organisation And Garbage Collection

19.1 Introduction

This chapter may be skipped on a first reading. Its purpose is to give the advanced user a better understanding of how the system uses memory resources. In a high level language like Prolog it is often not obvious for the programmer to see where the system allocates or frees memory.
The sizes of the different memory areas can be queried by means of the predicate **statistics/2** and **statistics/0** prints a summary of all these data. Here is a sample output:

```
[eclipse 1]: statistics.

times:                  [0.133333, 0.65, 304.809] seconds
global_stack_used:      724 bytes
global_stack_allocated: 102400 bytes
global_stack_peak:      1576960 bytes
trail_stack_used:       32 bytes
trail_stack_allocated:  16384 bytes
trail_stack_peak:       16384 bytes
control_stack_used:     392 bytes
control_stack_allocated:4096 bytes
control_stack_peak:     4096 bytes
local_stack_used:       288 bytes
local_stack_allocated:  4096 bytes
local_stack_peak:       4096 bytes
code_heap_allocated:    326888 bytes
code_heap_used:         326888 bytes
general_heap_allocated: 291608 bytes
general_heap_used:      278432 bytes
gc_number:              0
gc_collected:           0 bytes
gc_area:                0 bytes
gc_ratio:               100.0 %
gc_time:                0.0 seconds
```

```
dictionary_entries:     2494
dict_hash_usage:        1498 / 8191
dict_hash_collisions:   155 / 1498
dict_gc_number:         0
dict_gc_time:           0.0 seconds
```

The **used**-figures indicate the actual usage at the moment the statistics built-in was called. The **allocated** value is the amount of memory that is reserved for this area and actually occupied by the ECL^{i}PS^e process. The **peak** value indicates what was the maximum allocated amount during the session. In the following we will discuss the six memory areas mentioned. The gc-figures are described in section 19.2.

19.1.1 The Code Heap

The code heap is used to store compiled Prolog code and external predicates. Consequently its size is increased by the various **compile**-predicates, the **assert**-family and by **load/1**.
Space on the code heap is freed when single clauses (**retract**) or whole predicates (**abolish**) are removed from the system. Note that space reclaiming is usually delayed in these cases, since the removed code may still be under execution. Erasing a module also reclaims all the memory occupied by the module's predicates.

19.1.2 The General Heap

The general heap is used to store a variety of data:

- **internal database:** Term that are recorded using the **record**-family of builtins are stored on the general heap. The space is freed when the terms are erased.

- **arrays and global variables:** Space for storing global variables and arrays (**make_array**, **setval**) is allocated from the general heap. When a global variable is overwritten , the space for the old value is reclaimed. All memory related to a global variable or array is reclaimed when it is destroyed using **erase_array/1**.

- **dictionary:** The *dictionary* is the system's table of atoms and functors. The dictionary grows whenever the system encounters an atom or functor that has not been mentioned so far. The dictionary shrinks on dictionary garbage collections, which are triggered automatically after a certain number of new entries has been made. The dictionary is designed to hold several thousands of entries, the current number of entries can be queried with **statistics/0,2**.

- **various descriptors:** The system manages a number of other internal tables (for modules, predicates, streams, operators, etc.) that are also allocated on the general heap. This space is reclaimed when the related Prolog objects cease to exist.

- **I/O-buffers:** When streams are opened, the system allocates buffers from the general heap. They are freed when the stream is closed.

- **allocation in C-externals:** If external predicates written in C call malloc() or related C library functions, this space is also allocated from the general heap. It is the user's responsibility to free this space if it becomes unused.

19.1.3 The Local Stack

The Local Stack is very similar to the call/return stack in procedural languages. It holds Prolog variables and return addresses. Space on this stack is allocated during execution of a clause and deallocated before the last subgoal is called (due to tail recursion / last call optimisation). This deallocation can not be done when the clause exits nondeterministically (this can be checked with the debugger or the profiling facility). However, if a deallocation has been delayed due to nondeterminism, it is finally done when a cut is executed or when execution fails beyond the allocation point. Hence the ways to limit growth of the local stack are

- use tail recursion where possible
- avoid unnecessary nondeterminism (cf. 19.1.4)

19.1.4 The Control Stack

The main use of the Control Stack is to store so-called *choicepoints*. A choicepoint is a description of the system's state at a certain point in execution. It is created when more than one clause of a predicate apply to a given goal. Should the first clause fail, the system will backtrack to the place where the choice was made, the old state will be restored from the choicepoint and the next clause will be tried. Disjunctions (**;/2**) also create choicepoints.

The only way to reduce Control Stack usage is to avoid unnecessary nondeterminism. This is done by writing deterministic predicates in such a way that they can be recognised by the system. The debugger can help to identify nondeterministic predicates: When it displays an `*EXIT` port instead of `EXIT` then the predicate has left a choicepoint behind. In this case it should be checked whether the nondeterminism was intended. If not, the predicate can often be made deterministic by

- writing the clause heads such that a matching clause can be more easily selected by *indexing*
- using the if-then-else construct (`.. -> .. ; ..`)
- deliberate insertion of (green) cuts

19.1.5 The Global Stack

The Global Stack holds Prolog structures, lists, strings and long numbers. So the user's selection of data structures is largely responsible for the growth of this stack (cf. 6.4). In coroutining mode, delayed goals also consume space on the Global Stack. It also stores source variable names for terms which were read in with the flag **variable_names** being `on`. When this feature is not needed, it should be turned off so that space on the global stack is saved.

The global stack grows while a program creates data structures. It is popped only on failure. ECL^{i}PS^e therefore provides a garbage collector for the Global Stack which is called when a certain amount of new space has been consumed. See section 19.2 for how this process can be controlled. Note again that unnecessary nondeterminism reduces the amount of garbage that can be reclaimed and should therefore be avoided.

19.1.6 The Trail Stack

The Trail Stack is used to record information that is needed on backtracking. It is therefore closely related to the Control Stack. Ways to reduce Trail Stack consumption are

- avoid unnecessary nondeterminism
- supply **mode** declarations

The Trail Stack is popped on failure and is garbage collected together with the Global Stack.

19.2 Garbage collection

The four stacks grow an shrink as needed[1]. In addition, ECL^{i}PS^e provides an incremental garbage collector for the global and the trail stack. It is also equipped with a dictionary garbage collector that frees memory that is occupied by obsolete atoms and functors. Both collectors are switched on by default and are automatically invoked from time to time. Nevertheless, there are some predicates to control their action. The following predicates affect both collectors:

set_flag(gc, on). Enable the garbage collector (the default).

set_flag(gc, verbose). The same as 'on', but print a message on every collection (the message goes to toplevel_output):

```
GC ... global: 96208 - 88504 (92.0 %), trail: 500 - 476 (95.2 %), time: 0.017
```

It displays the area to be searched for garbage, the amount and percentage of garbage, and the time for the collection. The message of the dictionary collector is as follows:

```
DICTIONARY GC ... 2814 - 653, (23.2 %), time: 0.033
```

It displays the number of dictionary entries before the collection, the number of collected entries, the percentage of reduction and the collection time.

set_flag(gc, off). Disable the garbage collector (and risk an overflow), eg. for time-critical execution sequences.

Predicates related to the stack collector are:

set_flag(gc_interval, Nbytes). Specify how often the collector is invoked. Roughly, Nbytes is the number of bytes that your program can use up before a garbage collection is triggered. There may be programs that create lots of (useful) lists and structures while leaving few garbage. This will cause the garbage collector to run frequently while reclaiming little space. If you suspect this, you should call statistics/0 and check the garbage ratio. If it is very low (say below 50%) it may make sense to increase the `gc_interval`, thus reducing the number of garbage collections.

garbage_collect. Request an immediate collection (only if enabled). The use of this predicate should be restricted to situations where the automatic triggering performs badly. It should then be inserted in a place where you know for sure that you have just created a lot of garbage, eg. before the tail-recursive call in something like

---

[1] provided that the underlying operating system supports this

```
cycle(OldState) :-
        transform(OldState, NewState),  /* long computation     */
        !,
        garbage_collect,                /* OldState is obsolete */
        cycle(NewState).
```

statistics(gc_number, N). The number of stack garbage collections performed during this ECL^{i}PS^e session.

statistics(gc_collected, Bytes). The amount of global stack space reclaimed by all the garbage collections in bytes.

statistics(gc_area, Bytes). The average global stack area that was scanned by each garbage collection. This number should be close to the selected `gc_interval`, if it is much larger, `gc_interval` should be increased.

statistics(gc_ratio, Percentage). The average percentage of garbage found and reclaimed by each garbage collection. If this ratio is low, `gc_interval` should be increased.

statistics(gc_time, Seconds). The total cputime spent during all garbage collections.

Predicates related to the dictionary collector are:

set_flag(gc_interval_dict, N). Specify that the dictionary collector should be invoked after N new dictionary entries have been made.

statistics(dict_gc_number, N). The number of dictionary garbage collections performed during this ECL^{i}PS^e session.

statistics(dict_gc_time, Seconds). The total cputime spent by all dictionary garbage collections.

Chapter 20

Operating System Interface

20.1 Introduction

The ECL^{i}PS^e system has been developed under the UNIX operating system. The interface to the file system, process communication and environment are closely related to the corresponding UNIX facilities. Unfortunately a few features are only available on UNIXes of the Berkeley family, not on System-V. However they were considered useful enough not to leave them out for portability reasons. Some features are therefore not supported on some System-V versions. Since there is a trend to unify the different UNIXes, these incompatibilities will hopefully disappear in the future.
ECL^{i}PS^e's operating system interface consists of a collection of built-in predicates and some global flags that are accessed with **set_flag/2**, **get_flag/2** and **env/0**. They are described in the following sections.

20.2 Environment Access

A number of predicates and global flags is provided to get more or less useful information from the operating system environment.

20.2.1 Command Line Arguments

Arguments provided on the UNIX command line are accessed by the builtins **argc/1** which gives the number of command line arguments (including the command name itself) and **argv/2** which returns a requested positional argument in string form. The following example makes a list of all command line arguments:

```
collect_args(List) :-
        argc(N),
        collect_args(N, [], List).
collect_args(0, List, List) :- !.
collect_args(N, List0, List) :-
        N1 is N-1,
        argv(N1, Arg),
        collect_args(N1, [Arg|List0], List).
```

20.2.2 Environment Variables

UNIX environment variables are another way to pass information to the ECL^{i}PS^e process. Their string value can be read using **getenv/2**:

```
[eclipse 1]: getenv('HOME', Home).

Home = "/usr/octopus"
yes.
```

20.2.3 Exiting ECL^{i}PS^e

When ECL^{i}PS^e is exited, it can give a return code to the operating system. This is done by using **exit/1**. It exits ECL^{i}PS^e and returns its integer argument to the operating system.

```
[eclipse 1]: exit(99).
csh% echo $status
99
```

Note that **halt** is equivalent to **exit(0)**.

20.2.4 Time and Date

The current date can be obtained in the form of a UNIX date string:

```
[eclipse 1]: date(Today).

Today = "Tue May 29 20:49:39 1990\n"
yes.
```

The library **calendar** contains a utility predicate to convert this string into a Prolog structure. Another way to access the current time and date is the global flag **unix_time**. It returns the current time in the traditional UNIX measure, i.e. in seconds since 00:00:00 GMT Jan 1, 1970:

```
[eclipse 1]: get_flag(unix_time, Now).

Now = 644008011
yes.
```

Other interesting timings concern the resource usage of the running ECL^{i}PS^e. The **statistics/2** builtin gives three different times, the user cpu time, the system cpu time and the elapsed real time since the process was started (all in seconds):

```
[eclipse 1]: statistics(times, Used).

Used = [0.916667, 1.61667, 2458.88]
yes.
```

The first figure (user cpu time) is the same as given by **cputime/1**.

20.2.5 Host Computer

Access to the name and unique identification of the host computer where the system is running can be obtained by the two global flags **hostname** and **hostid**, accessed via **get_flag/2** or **env/0**. These flags might not be available on all machines, **get_flag/2** fails in these cases.

20.2.6 Calling C Functions

Other data may be obtained with the predicate **call_c/2** which allows to call directly any C function which is linked to the Prolog system. Functions which are not linked can be loaded dynamically with the **load/1** predicate.

20.3 File System

A number of built-in predicates is provided for dealing with UNIX files and directories. Here we consider only the file as a whole, for opening files and accessing their contents refer to chapter 12.

20.3.1 Current Directory

The current working directory is an important notion in UNIX. It can be read and changed within the ECL^{i}PS^e system by using **getcwd/1** and **cd/1** respectively. The current working directory is accessible as a global flag as well. Reading and writing this flag is equivalent to the use of **getcwd/1** and **cd/1**:

```
[eclipse 1]: getcwd(Where).

Where = "/usr/name/prolog"
yes.
[eclipse 2]: cd(..).

yes.
[eclipse 3]: get_flag(cwd, Where)

Where = "/usr/name"
yes.
```

All ECL^{i}PS^e built-ins that take file names as arguments accept absolute pathnames as well as relative pathnames starting at the current directory.

20.3.2 Looking at Directories

To look at the contents of a directory, **read_directory/4** is available. It takes a directory pathname and a filename pattern and returns a list of subdirectories and a list of files matching the pattern. The following metacharacters are recognised in the pattern: * matches an arbitrary sequence of characters, ? matches any single character, [] matches one of the characters inside the brackets unless the first one is a ^ in which case it matches any character but those inside the brackets.

```
[eclipse 1]: read_directory("/usr/john", "*", Dirlist, Filelist).
Dirlist = ["subdir1", "subdir2"]
Filelist = ["one.c", "two.c", "three.pl", "four.pl"]
yes.
```

20.3.3 Checking Files

For checking the existence of files, **exists/1** is used. For accessing any file properties there is **get_file_info/3**. It can return file permissions, type, owner, size, inode, number of links as well as creation, access and modification times (as defined by the UNIX system call *stat(2)*), and accessibility information. It fails when the specified file does not exist. Refer to the BIP book or **help/1** for details.

20.3.4 Renaming and Removing Files

For these basic operations with files, **rename/2** and **delete/1** are provided.

20.3.5 Filenames

The utilities **pathname/3**, **pathname/2** and **suffix/2** are provided to ease the handling of filenames. **pathname/3** takes a full pathname and cuts it into the directory pathname and the filename proper. It also expands symbolic pathnames, starting with ~, ~user or $var. **suffix/2** unifies its second argument with the suffix of the filename, i.e. the part beginning with the last dot in the string.

```
[eclipse 1]: Name = "~octopus/prolog/file.pl",
        pathname(Name, Path, File),
        suffix(Name, Suffix).

Path = "/usr/octopus/prolog/"
File = "file.pl"
Name = "~octopus/prolog/file.pl"
Suffix = ".pl"
yes.
```

20.4 Creating Communicating Processes

ECL^{i}PS^e provides all the necessary built-ins needed to create UNIX processes and establish communication between them. A ECL^{i}PS^e process can communicate with other processes via streams and by sending and receiving signals.

20.4.1 Process creation

The built-ins of the **exec** group and **sh/1** fork a new process and execute the command given as the first argument. Sorted by their versatility, there are:

- **sh(Command)**
- **exec(Command, Streams)**

- **exec(Command, Streams, ProcessId)**
- **exec_group(Command, Streams, ProcessId)**

With **sh/1** (or its synonym **system/1**) it is possible to call and execute any UNIX command from withing ECL^{i}PS^e. However it is not possible to communicate with the process. Moreover, the ECL^{i}PS^e process just waits until the command has been executed.
The **exec** group makes it possible to set up communication links with the child process by specifying the `Streams` argument. It is a list of the form

```
[Stdin, Stdout, Stderr]
```

and specifies which ECL^{i}PS^e stream should be connected to the *stdin*, *stdout* or *stderr* of the child respectively. Unless `null` is specified, this will establish pipes to be created between the ECL^{i}PS^e process and the child. On Berkeley UNIX systems the streams can be specified as *sigio(Stream)* which will setup the pipe such that the signal *sigio* is issued every time new data appears on the pipe. Thus, by defining a suitable interrupt handler, it is possible to service this stream in a completely asynchronous way.

20.4.2 Process control

The **sh/1** and **exec/2** built-ins both block the ECL^{i}PS^e process until the child has finished. For more sophisticated applications, the processes have to run in parallel and be synchronised explicitly. This can be achieved with **exec/3** or **exec_group/3**. These return immediately after having created the child process and unify its process identifier (*Pid*) with the their argument. The Pid can be used to

- send signals to the process, using the built-in **kill(Pid, Signal)**
- wait for the process to terminate and obtain its return status **wait(Pid, Status)**

The difference between **exec/3** and **exec_group/3** is that the latter creates a new process group for the child, such that the child does not get the interrupt, hangup and kill signals that are sent to the parent.
The process identifier of the running ECL^{i}PS^e and of its parent process are available as the global flags **pid** and **ppid** respectively. They can be accessed using **get_flag/2** or **env/0**.
Here is an example of how to connect the UNIX utility **bc** (the arbitrary-precision arithmetic language) to a ECL^{i}PS^e process. We first create the process with two pipes for the child's standard input and output. Then, by writing and reading these streams, the processes can communicate in a straightforward way. Note that it is usually necessary to flush the output after writing into a pipe:

```
[eclipse 1]: exec(bc, [in,out], P).

P = 9759
yes.
[eclipse 2]: writeln(in, "12345678902321 * 2132"), flush(in).

yes.
[eclipse 3]: read_string(out, "\n", _, Result).
```

```
Result = "26320987419748372"
yes.
```

In this example the child process can be terminated by closing its standard input (in other cases it may be necessary to send a signal). The built-in **wait/2** is then used to wait for the process to terminate and to obtain its exit status. Don't forget to close the ECL^{i}PS^e streams that were opend by **exec/3**:

```
[eclipse 4]: close(in), wait(P,S).

P = 9759
S = 0     More? (;)
yes.
[eclipse 5]: at_eof(out), close(out).

yes.
```

20.4.3 Interprocess Signals

The UNIX signals are all mapped to ECL^{i}PS^e interrupts. Their names and numbers may vary on different machines. Refer to the operating system documentation for details.
The way to deal with incoming signals is to define a Prolog or external predicate and declare it as the interrupt handler for this interrupt (using **set_interrupt_handler/2**). Interrupt handlers can be established for all signals except those that are not allowed to be caught by the process (like e.g. the *kill* signal 9). For a description of event handling in general see chapter 14.
For explicitly sending signals to other processes **kill/2** is provided, which is a direct interface to the UNIX system call *kill(2)*. Note that some signals can be set up to be raised automatically, e.g. `sigio` can be raised when data arrives on a pipe.

20.4.4 Internal Signals

Sometimes it is useful to explicitly sent signals to the own process. An interesting case is timer signals. A simple interface to the corresponding operating system facilities is provided with the built-ins

alarm(+Time) this will send a timeout signal (usually `alrm`/14) after a specified number of seconds

set_timer(+Timer, +Interval) activate different operating system timers to send signals in regular intervals

These built-ins may not be available on all machines.

Chapter 21

Interprocess Communication

ECL^{i}PS^e contains built-in predicates that support interprocess communications using sockets. Sockets, available in the BSD Unix, implement bidirectional channels that can connect multiple processes on different machines in different networks. The socket predicates are directly mapped to the system calls and therefore detailed information can be found in the Unix manuals.

Sockets in general allow a networked communication among many processes, where each packet sent by one process can be sent to different address. In order to limit the number of necessary built-in predicates, ECL^{i}PS^e supports only point-to-point communication based on stream or datagram sockets, or many-to-one communication based on datagrams. Broadcasting as well as using one socket to send data to different addresses is not supported, except that datagram sockets can be re-connected, so that the same socket is directed to another address. Below we describe the basic communication types that are available in ECL^{i}PS^e.

Note that the sockets streams in ECL^{i}PS^e are buffered like all other streams, and so it is necessary to flush the buffer in order to actually send the data to the socket. This can be done either with the **flush/1** predicate or with the option **%b** in **printf/2, 3**.

21.1 Socket Domains

Currently there are two available domains, `unix` and `internet`. The communication in the `unix` domain is limited to a single machine, and the sockets are associated to files in this machine's file system.

The `internet` domain can be used to connect any two machines which are connected through the network. The address of a socket is then identified by the host name and the port number. The host name is the same as obtained e.g. with the **get_flag(hostname, Host)**. The port identifies the channel on the host which is used for the communication.

21.2 Stream Connection on a Single Machine

This type of communication is very similar to pipes, the stream communication is reliable and there are no boundaries between the messages. Stream sockets always require explicit connection from both communicating processes.

After a socket is created with the **socket/3** predicate, one of the processes, the server, gives it a name and waits for a connection. The other process uses the same name when connecting to the former process. After the connection is established, both processes can read and write on the socket and so the difference between the server and the client disappears:

```
server:
    [eclipse 10]: socket(unix, stream, s), bind(s, '/tmp/sock').

    yes.
    [eclipse 11]: listen(s, 1), accept(s, _, news).
    <blocks waiting for a connection>

client:
    [eclipse 26]: socket(unix, stream, s), connect(s, '/tmp/sock').

    yes.
    [eclipse 27]: printf(s, "%w. %b", message(client)), read(s, Msg).

server:
    [eclipse 12]: read(news, Msg), printf(news, "%w. %b", message(server)).

    Msg = message(client)
    yes.

client:
    Msg = message(server)
    yes.
```

21.3 Datagram Connection on a Single Machine

The datagram connection is based on packets sent from one process to another, similar to usual LAN's, e.g. the Ethernet. The communication protocol does not guarantee that the message will always be delivered, but normally it will be. Every packet represents a message which is read separately at the system level, however at the Prolog level the packet boundaries are not visible[1]. The difference to stream communication is that there is no obligatory connection between the server and the client. First the socket has to be created, and then the process which wants to read from the it binds the socket to a name. Any other process can then connect directly to this socket using the **connect/2** predicate and send data there. This connection can be temporary, and after writing the message to the socket the process can connect it to another socket, or just disconnect it by calling **connect(Socket, 0)**.
Such datagram connection works only in one direction, namely from the process that called **connect/2** to the process that called **bind/2**, however the connection in the other direction can be established in the same way:

```
server:
    % Make a named socket and read two terms from it
    [eclipse 10]: socket(unix, datagram, s), bind(s, '/tmp/sock').
```

---

[1] The packet boundaries are not of much interest in Prolog because every Prolog term represents itself a message with clear boundaries.

```
    yes.
    [eclipse 11]: read(s, X), read(s, Y).

process1:
    % Connect a socket to the server and write one term
    [eclipse 32]: socket(unix, datagram, s), connect(s, ’/tmp/sock’).

    yes.
    [eclipse 33]: printf(s, "%w. %b", message(process1)).

process2:
    % Connect a named socket to the server and write another term
    [eclipse 15]: socket(unix, datagram, s), connect(s, ’/tmp/sock’),
        bind(s, ’/tmp/socka’).

    yes.
    [eclipse 16]: printf(s, "%w. %b", message(process2)).

    yes.
    % And now disconnect the output socket from the server
    [eclipse 17]: connect(s, 0).

    yes.

server:
    % Now the server can read the two terms
    X = message(process1)
    Y = message(process2)
    yes.
    % and it writes one term to the second process on the same socket
    [eclipse 12]: connect(s, ’/tmp/socka’),
        printf(s, "%w. %b", message(server)).

process2:
    %
    [eclipse 18]: read(s, Msg).

    Msg = message(server)
    yes.
```

21.4 Stream Connection Between Two Machines

The sequence of operations is the same as for the Unix domain, however in the Internet domain the socket addresses are not related to the file system, they contain the host name and the port number. Since one port number identifies the socket on a given host, the process cannot itself specify the port number it wants to use because it can be already in use by another process. Therefore, the safe approach is to use the default and let the system specify the port number, which is achieved by leaving the port uninstantiated. Since the host is always known, it can also be left uninstantiated. The client, however, has to specify both the host name and the port number:

```
server:
    [eclipse 10]: socket(internet, stream, s), bind(s, X).

    X = acrab5 / 3789
    yes.
    [eclipse 11]: listen(s, 1), accept(s, From, news).
    <blocks waiting for a connection>

client:
    [eclipse 26]: socket(internet, stream, s), connect(s, acrab5/3789).

    yes.
    [eclipse 27]: printf(s, "%w. %b", message(client)), read(s, Msg).

server:
    From = acrab4 / 1627
    yes.
    [eclipse 12]: read(news, Msg), printf(news, "%w. %b", message(server)).

    Msg = message(client)
    yes.

client:
    Msg = message(server)
    yes.
```

21.5 Datagram Connection with Multiple Machines

This type of communication, which is the most general one offered by ECL^{i}PS^e, is similar to the datagram connection on a single machine with the exception that instead of the `unix` domain the `internet` domain is used and that any machine which is reachable over the network can participate in the communication.

Since ECL^{i}PS^e does not provide a link to the system call *sendto()*, the address where the packet should be sent to can be specified only using **connect/2**. If the next packet is to be sent to a different address, a new **connect/2** call can be used. The socket can be disconnected by calling **connect(s, 0/0)**.

The functionality of *recvfrom()* is not available, i.e. the sender has to identify itself explicitly in the message if it wants the receiver to know who the sender was.

Below is an example of a program that starts ECL^{i}PS^e on all available machines which are not highly loaded and accepts a hello message from them. Note the use of *rsh* to invoke the process on the remote machine and pass it the host name and port address.

```
% Invoke ECLiPSe on all available machines and accept a hello message
% from them.
connect_machines :-
    machine_list(List),        % make a list of machines from ruptime
    socket(internet, datagram, sigio(s)), % signal when data comes
    bind(s, Address),
    set_interrupt_handler(io, io_handler/0),
    connect_machines(List, Address).

% As soon as a message arrives to the socket, the io signal will
% be sent and the handler reads the message.
io_handler :-
    set_flag(enable_interrupts, off),
    read_string(s, "\n", _, Message),
    writeln(Message),
    set_flag(enable_interrupts, on).

% Invoke eclipse on all machines with small load and let them execute
% the start/0 predicate
connect_machines([info(RHost, UpTime, Users, L1, _, _)|Rest], Host/Port) :-
    UpTime > 0,         % it is not down
    L1 < 0.5,           % load not too high
    Users < 3,          % not too many users
    !,
    concat_string(['rsh ', RHost, ' eclipse ', Host, ' ', Port,
        ' -b /home/lp/micha/sepia4/up.pl -e start'], Command),
    printf("sending to %s\n%b", RHost),
    exec(Command, [], _),
    connect_machines(Rest, Host/Port).
connect_machines([_|Rest], Address) :-
    connect_machines(Rest, Address).
connect_machines([], _).

% ECLiPSe on remote hosts is invoked with
%          eclipse host port -b file.pl -e start
% It connects to the socket of the main process,
% sends it a hello message and exits.
start :-
    is_built_in(socket/3),    % to ignore non-BSD machines
    argv(1, SHost),
    argv(2, SPort),
    atom_string(Host, SHost),
    number_string(Port, SPort),
    get_flag(hostname, LHost),
    socket(internet, datagram, s),   % create the socket
    connect(s, Host/Port),           % connect to the main process
    printf(s, "hello from %s\n%b", LHost).

% Invoke ruptime(1) and parse its output to a list of accessible
% machines in the form
%    info(Host, UpTime, Users, Load1, Load2, Load3).
machine_list(List) :-
    % exec/2 cannot be used as it could overflow
    % the pipe and then block
    exec('ruptime -l', [null, S], P),
    parse_ruptime(S, List),
    close(S),
    wait(P, _),
    !.

% Parse the output of ruptime
parse_ruptime(S, [Info|List]) :-
    parse_uptime_record(S, Info),
    !,
    parse_ruptime(S, List).
parse_ruptime(_, []).

% parse one line of the ruptime output
parse_uptime_record(S, info(Host, Time, Users, Load1, Load2, Load3)) :-
    read_token(S, Host, _),
    Host \== end_of_file,
    read_token(S, Up, _),
    (Up == up ->
        read_time(S, Time),
        read_token(S, ',', _),
        read_token(S, Users, _),
        read_token(S, _, _),
        read_token(S, ',', _),
        read_token(S, load, _),
        read_token(S, Load1, _),
        read_token(S, ',', _),
        read_token(S, Load2, _),
        read_token(S, ',', _),
        read_token(S, Load3, _)
    ;
        read_time(S, _),
        Time = 0
    ).

% Parse the up/down time and if the machine is down, return 0
read_time(S, Time) :-
    read_token(S, T1, _),
    (read_token(S, +, _) ->
            Days = T1,
            read_token(S, Hours, _),
            read_token(S, :, _)
        ;
            Days = 0,
            Hours = T1
        ),
        read_token(S, Mins, _),
        Time is ((24 * Days) + Hours) * 60 + Mins.
```

and here is a script of the session:

```
[eclipse 1]: [up].
up.pl      compiled traceable 4772 bytes in 0.08 seconds

yes.
[eclipse 2]: connect_machines.
sending to mimas3
sending to mimas8
sending to acrab23
sending to europa1
sending to europa5
sending to regulus2
sending to miranda5
sending to mimas2
sending to triton6
sending to europa2
sending to acrab7
sending to europa3
sending to sirius
sending to miranda6
sending to charon6
sending to acrab13
sending to triton1
sending to acrab20
sending to triton4
sending to charon2
sending to triton5
sending to acrab24
sending to acrab21
sending to scorpio
sending to acrab14
sending to janus5

yes.
[eclipse 3]: hello from mimas3
eclipse: Command not found.     % eclipse not installed here
hello from regulus2
hello from mimas8
hello from acrab20
hello from europa1
hello from mimas2
hello from miranda6
hello from miranda5
hello from europa3
hello from charon6
hello from charon2
hello from acrab24
hello from triton5
hello from acrab21
hello from janus5
hello from triton4
hello from triton6
hello from europa2
hello from europa5
hello from acrab23
hello from triton1
hello from acrab14
hello from acrab13
hello from acrab7
```

Chapter 22

Profiling Prolog Execution

22.1 Introduction

The profiling tool[1] helps to find "hot spots" in a program that are worth optimising. It can be used any time with any compiled Prolog code, it is not necessary to use a special compilation mode or set any flags. When

:- profile(Goal).

is called, the profiler executes the *Goal* in the profiling mode, which means that every 0.01s the execution is interrupted and the profiler remembers the currently executing procedure. When the goal succeeds or fails, the profiler prints so and then it prints the statistics about the time spent in every encountered procedure:

```
[eclipse 5]: profile(boyer).
rewriting...
proving...
goal succeeded

                PROFILING STATISTICS
                --------------------

Goal:             boyer
Total user time:  10.65s

Predicate             Module        %Time  Time
-------------------------------------------------
rewrite          /2  eclipse       52.3%  5.57s
garbage_collect  /0  sepia_kernel  23.1%  2.46s
rewrite_args     /2  eclipse       16.6%  1.77s
equal            /2  eclipse        4.7%  0.50s
remainder        /3  eclipse        1.5%  0.16s
```

---

[1] The profiler requires a small amount of hardware/compiler dependent code and may therefore not be available on all platforms.

```
...
plus              /3  eclipse         0.1%  0.01s

yes.
```

The profiler prints the predicate name and arity, its definition module, percentage of total time spent in this predicate and the absolute time. Some of auxiliary system predicates are printed under a common name without arity, e.g. *arithmetic* or *all_solutions*. Predicates which are local to locked modules are printed together on one line that contains only the module name. By default only predicates written in Prolog are profiled, i.e. if a Prolog predicate calls an external or built-in predicate written in C, the time will be assigned to the Prolog predicate.
The predicate **profile(Goal, Flags)** can be used to change the way profiling is made, *Flags* is a list of flags. Currently only the flag `simple` is accepted and it causes separate profiling of simple predicates, i.e. those written in C:

```
[eclipse 6]: profile(boyer, [simple]).
rewriting...
proving...
goal succeeded

                PROFILING STATISTICS
                --------------------

Goal:             boyer
Total user time:  10.55s

Predicate             Module        %Time  Time
-------------------------------------------------
=..               /2  sepia_kernel  31.1%  3.28s
garbage_collect   /0  sepia_kernel  23.5%  2.48s
rewrite           /2  eclipse       21.6%  2.28s
rewrite_args      /2  eclipse       17.2%  1.82s
equal             /2  eclipse        4.1%  0.43s
remainder         /3  eclipse        0.9%  0.10s
...
plus              /3  eclipse        0.1%  0.01s

yes.
```

Appendix A

Libraries

A.1 Calendar Library

This library contains a set of predicates to assist with the handling of dates and times. It is loaded using

```
:- use_module(library(calendar)).
```

The library represents time points as *Modified Julian Dates* (MJD). Julian Dates (JD) and Modified Julian Dates (MJD) are a consecutive day numbering scheme widely used in astronomy, space travel etc. That means that every day has a unique integer number, and consecutive days have consecutive numbers. Note that you can also use fractional MJDs to denote the time of day. Then every time point has a unique floating point representation! With this normalised representation, distances between times are obviously trivial to compute, and so are weekdays (by simple mod(7) operation).
The predicates provided are

date_to_mjd(+D/M/Y, -MJD) converts a Day/Month/Year structure into its unique integer MJD number.

mjd_to_date(+MJD, -D/M/Y) converts an MJD (integer or float) into the corresponding Day/Month/Year.

time_to_mjd(+H:M:S, -MJD) returns a float MJD <1.0 encoding the time of day (UTC/GMT). This can be added to an integral day number to obtain a full MJD.

mjd_to_time(+MJD, -H:M:S) returns the time of day (UTC/GMT) corresponding to the given MJD as Hour:Minute:Seconds structure, where Hour and Minute are integers and Seconds is a float.

mjd_to_weekday(+MJD, -DayName) returns the weekday of the specified MJD as atom monday, tuesday etc.

mjd_to_dow(+MJD, -DoW) returns the weekday of the specified MJD as an integer (1 for monday up to 7 for sunday).

mjd_to_dow(+MJD, +FirstWeekday, -DoW) as above, but allows to choose a different starting day for weeks, specified as atom monday, tuesday etc.

mjd_to_dy(+MJD, -DoY/Y), dy_to_mjd(+DoY/Y, -MJD) convert MJDs to or from a DayOfYear/Year representation, where DayOfYear is the relative day number starting with 1 on every January 1st.

mjd_to_dwy(+MJD, -DoW/WoY/Y), dwy_to_mjd(+DoW/WoY/Y, -MJD) convert MJDs to or from a DayOfWeek/WeekOfYear/Year representation, where DayOfWeek is the day number within the week (1 for monday up to 7 for sunday), and WeekOfYear is the week number within the year (starting with 1 for the week that contains January 1st).

mjd_to_dwy(+MJD, +FirstWeekday, -DoW/WoY/Y)

dwy_to_mjd(+DoW/WoY/Y, +FirstWeekday, -MJD) As above, but allows to choose a different starting day for weeks, specified as atom monday, tuesday etc.

unix_to_mjd(+UnixSec, -MJD) convert the UNIX time representation into a (float) MJD.

mjd_now(-MJD) returns the current date/time as (float) MJD.

jd_to_mjd(+JD, -MJD), mjd_to_jd(+MJD, -JD) convert MJDs to or from JDs. The relationship is simply MJD = JD-2400000.5.

The library code is valid for dates between 1 Mar 0004 = MJD -677422 = JD 1722578.5 and 22 Jan 3268 = MJD 514693 = JD 2914693.5.

A.1.1 Examples

What day of the week was the 29th of December 1959?

```
[eclipse 1]: lib(calendar).
[eclipse 2]: date_to_mjd(29/12/1959, MJD), mjd_to_weekday(MJD,W).
MJD = 36931
W = tuesday
```

What date and time is it now?

```
[eclipse 3]: mjd_now(MJD), mjd_to_date(MJD,Date), mjd_to_time(MJD,Time).
Date = 19 / 5 / 1999
MJD = 51317.456238425926
Time = 10 : 56 : 59.000000017695129
```

How many days are there in the 20th century?

```
[eclipse 4]: N is date_to_mjd(1/1/2001) - date_to_mjd(1/1/1901).
N = 36525
```

The library code does not detect invalid dates, but this is easily done by converting a date to its MJD and back and checking whether they match:

```
[eclipse 5]: [user].
valid_date(Date) :-
        date_to_mjd(Date,MJD),
        mjd_to_date(MJD,Date).

[eclipse 6]: valid_date(29/2/1900).   % 1900 is not a leap year!
no (more) solution.
```

A.2 CIO

This library provides C-Prolog compatible I/O predicates. It is included in the C-Prolog compatibility package, but may as well be used independently. It is loaded using

```
:- lib(cio).
```

The following global predicates are defined by this library:

see/1

seeing/1

seen/1

skip/1

skip/2

tab/1

tab/2

tell/1

telling/1

told/1

The predicates change ECL^{i}PS^e's **input** or **output** stream, respectively. Other streams, like **toplevel_input**, are not affected, so the behaviour is similar as in Quintus Prolog.

A.3 Crossreference Checking

A.3.1 Checking of Loaded Predicates

This library is loaded by

```
:- lib(check).
```

The global predicate **check/0** defined in this library will make extensive checks of all the Prolog code in the Prolog database and it will print warning for the following predicates:

- predicates whose visibility has been declared but that are not defined,
- predicates that have been declared as a tool (with **tool/1**) but have not been linked to a tool body,
- predicates whose type (external or b_external) has been declared but that are not defined or declared (for the visibility),
- tool interfaces whose tool body is not defined.
- predicates that have been referenced (e.g. compiled a call to it) but that are not defined and have no declaration of any type,
- local predicates that are not declared nor referenced (and so probably redundant).

check/1 will check a specified module.

A.3.2 Checking Files

The library **xref.pl** defines the global predicate **xref(FileList)**. When called, this predicate will examine the given file or list of file names and print separately predicates which are defined and those that are not defined in the given file(s).

A.4 HTTP Library

The HTTP library contains an extensible server and a client for the Hyper Text Transfer Protocol. The library is entirely written in ECL^{i}PS^e.
Typical use of the client is for building WWW "Worms", WWW "Robots" or customized WWW browsers. Typical use of the server is for building customized servers, e.g. dynamic generation of HTML pages. The server and the client can typically be used together to build proxy servers.
Limitations and Bugs:

- The current version of the server is sequential.
- The MIME and HTTP grammar is not complete and may fail parsing some sentences generated by existing browsers and servers.

The library provides two predicates: http_client/7, and http_server/1.

A.4.1 Client

```
http_client(+Method, +Uri, +ObjectBody, +HttpParams,
                    -RespError, -RespParam, -RespObjectBody)
        Metod and Uri and ObjectBody and String are strings
        HttpParams is a list of terms defined in the DCG
        Error is a term error(ErrorCode, ErrorPhrase)
                ErrorCode: the error code contained in the response
                ErrorCode: the error phrase contained in the response
        RespParam is a list of terms defined in the DCG
        RespObjectBody is the object body of the response
```

The client does:

- encoding of a request
- sending of a request
- reception of the response
- decoding of the response

Example

The following example illustrates the use of a client.
The predicate http_client/7 is used to access the HTML pages, given their URI (the method GET is applied).

```
/*******************************************************************
 *  Web client example
 *******************************************************************/

> eclipse
ECRC Common Logic Programming System [sepia opium megalog parallel]
Version 3.5.2, Copyright ECRC GmbH, Wed Jan  3 12:54 1996
```

```
[eclipse 1]: use_module(http).
http_grammar.pl compiled traceable 25048 bytes in 0.38 seconds
http_client.pl compiled traceable 5916 bytes in 0.47 seconds
http_server.pl compiled traceable 5304 bytes in 0.07 seconds
http.pl    compiled traceable 0 bytes in 0.57 seconds

yes.
[eclipse 2]:  http_client("GET", "http://www.ecrc.de/staff/", "", [],
Status, Param, Resp).

Status = error(200, "Document follows ")
Param = [date, server, contentType(mt(text, html))]

Resp = "<HTML>...</HTML>"

yes.
```

A.4.2 Server

```
http_server(+Port)
        Port is an integer (usually superior to the last protected TCP port)
```

The server does:

- creation of a socket, bind it to current Host and given Port and listen
- accept a connection on the socket
- reception of a request
- decoding of the request (method + url + http param init)
- call the predicate http_method in module http_method
- encoding of the response (depending on server function)
- send the response on the socket

NOTE: The predicate http_server/1 requires that a module http_method is defined that contains a predicate http_method/6. This predicate is used by the programmer to customize the server. For instance the method GET can be simply implemented. The programmer can define its own methods.

Example

A simple example of server is the implementation of the method GET. A module is created that contains the predicate http_method/6 that implements the method GET: a read on the file identified by its URL. The file is returned if it is found, otherwise an error parameter is returned.

```
[eclipse 1]: [user].

/*****************************************************************
 *  test (server)
 *****************************************************************/

:- module(http_method).

:- set_error_handler(170, fail/0).
:- set_error_handler(171, fail/0).

/*
http_method(+Method, +Url, +ObjectBody, -Output, -StatusCode, -Parameter)
executes the method on the object and returns:
- the output of the method (possibly empty)
- a status code for the response status line
- a list of http parameters (in particular the length of the object body).

*/


http_method("GET", Url, _, Contents, 200, [contentLength(CL)]):-
        append_strings("/", FileName, Url),
        getContents(FileName, Contents), !,
        string_length(Contents, CL).
http_method("GET", _, _, "", 404, []).


getContents(Url, Contents):-
        open(Url, read, s),
        read_string(s, "", _, Contents),
        close(s).

^D

yes.

[eclipse 2]: use_module(http).
http_grammar.pl compiled traceable 25048 bytes in 0.27 seconds
http_client.pl compiled traceable 6052 bytes in 0.28 seconds
http_server.pl compiled traceable 5564 bytes in 0.03 seconds
http.pl    compiled traceable 0 bytes in 0.35 seconds

yes.
[eclipse 3]: use_module(http_method).

yes.
```

```
[eclipse 4]: http_server(8000).
```

This simple program can be used to test HTML pages. Viewers such as Netscape provide a view code option that signalizes syntax errors in the HTML code. This simple program can be used as a light weight testing tool, possibly launched from the directory where the HTML page resides.

A.4.3 HTTP Grammar

The structure of the HTTP messages is precisely described in the specification document (http://www.w3.org/pub/
An augmented BNF is provided for each component of the header. We have used the DCG (Definite Clause Grammar) mechanism of ECLiPSe to encode the grammar, that we use for both parsing (from HTTP messages into Prolog terms) and pretty printing (from prolog terms into HTTP messages).
This DCG grammar may have to be modified with the evolutions of the HTTP protocol (standard modification and available client or server implementations).

A.4.4 File Structure

http.pl exports the predicates that the user can manipulate: http_client/6, and http_server/1.

http_grammar.pl contains the grammar that allows to parse and pretty print the headers of the HTTP messages.

http_client.pl implements the HTTP client.

http_server.pl implements the HTTP server.

A.4.5 Authors

This library was designed and implemented by Ph. Bonnet, S. Bressan and M. Meier.

A.5 ISO Standard Prolog Compatibility Package

This library provides some degree of compatibility with the definition of Standard Prolog as defined in ISO/IEC 13211-1 (Information Technology, Programming Languages, Prolog, Part 1: General Core, 1995). The areas where the library is not yet complete are mainly I/O and directives. However it should be sufficient for many applications. The library is provided in source form.

The compatibility package is loaded by issuing the directive

```
:- use_module(library(iso)).
```

The effect of the compatibility library is local to the module where it is loaded, ie. the directive must be given in every module that wants to use the compatibility mode. Other modules are unaffected, which means that code written in different language variants can be mixed as long as it is organised into different modules.

A.6 C-Prolog Compatibility Package

One of the requirements during the development of ECL^{i}PS^e has been the aim of minimising the work required to port C-Prolog programs to ECL^{i}PS^e. To this end, many of the non standard predicates in C-Prolog have also been included in ECL^{i}PS^e. It is of course impossible to achieve total compatibility between the two systems. To assist in making the changes necessary to run a C-Prolog program on the current version of ECL^{i}PS^e, this appendix describes the predicates available in the C-Prolog compatibility library and it summarises the principal differences between ECL^{i}PS^e Prolog and C-Prolog.

Most of the C-Prolog predicates are also ECL^{i}PS^e built-in predicates and so they can be always accessed.

Please note that this appendix does not detail the functionality of C-Prolog, refer to the C-Prolog documentation for this information.

A.6.1 Using the C-Prolog compatibility package

The C-Prolog compatibility package can be accessed by issuing the directive

```
:- use_module(library(cprolog)).
```

The effect of the compatibility library is local to the module where it is loaded (except for the redefinition of some global event handlers). The directive must therefore be given in every module that wants to use the compatibility mode.

The C-Prolog compatibility package includes the **cio** and the **scattered** library.

A.6.2 C-Prolog compatibility predicates

The following predicates are available in the compatibility package. They are exported from the module **cprolog** and automatically imported by use_module/1 or lib/1.

consult/1

reconsult/1 The above two are implemented by simply calling the ECL^{i}PS^e predicate **compile/1**. By default all compiled procedures are static. Procedures on which **assert/1** etc. will be applied, have to be declared as *dynamic* using **dynamic/1**. The notation **[-File]** for **reconsult/1** is not supported.

current_functor/2

current_predicate/2

db_reference/1

erased/1

get/1 This is similar to the ECL^{i}PS^e predicate **get/1**, but control characters and blank spaces are skipped.

get0/1

fileerrors/0

heapused/1 Needed for evaluating **heapused** in arithmetic expressions. It returns the sum of code heap and general heap usage.

instance/2 Note that this compatibility predicate redefines the ECL^{i}PS^e builtin of the same name but different meaning (which is no longer available in C-Prolog mode). It is implemented using the ECL^{i}PS^e predicate **referenced_record/2**.

leash/1 The ECL^{i}PS^e leash built-in provides more functionality than the C-Prolog equivalent.

log/2

log10/2 These are not predicates in C-Prolog (arithmetic functors), but in ECL^{i}PS^e they are needed for evaluating log/1 and log10/1 in arithmetic expressions.

nofileerrors/0

primitive/1

prompt/2

put/1 This is similar to the ECL^{i}PS^e predicate **put/1**, but it first applies arithmetic evaluation to its argument.

sh/0

see/1

seeing/1

seen/0

skip/1

tab/1

tell/1

telling/1

told/0 The predicates of the **see/tell** family are defined in the **cio** library and can also be loaded independently.

ttyput/1 corresponds to the DEC-10 Prolog predicate

A.6.3 C-Prolog Predicates not available in ECL^{i}PS^e

The following C-Prolog predicates are not available in ECL^{i}PS^e, or the corresponding predicates have a different semantics:

assert/2

asserta/2

assertz/2

clause/3 ECL^{i}PS^e does not support database references for clauses.

expand_term/2 This is not supported. ECL^{i}PS^e provides the macro facility for transforming input terms (see chapter 13).

'LC'/0

'NOLC'/0 These are not supported in ECL^{i}PS^e.

A.6.4 Differences Between C-Prolog and ECL^{i}PS^e

The following differences remain even with the compatibility package:

Database References ECL^{i}PS^e provides database references only for terms in the indexed database, not for program clauses.

Numbers C-Prolog has a tendency to "prefer" integers over real numbers. For instance, under C-Prolog when the call `X is 4.0/2.0` is made, X is instantiated to an integer. This behaviour does not occur in ECL^{i}PS^e. The order of integers and reals in the standard order is different.

Operators In C-Prolog there is a bug regarding the operator **not** — it binds closer than its precedence declaration.

Strings Strings are simulated in C-Prolog by lists. Under C-Prolog mode, ECL^{i}PS^e provides this functionality — double-quoted strings are parsed as lists of integers. This can cause confusion when pure ECL^{i}PS^e predicates are used in C-Prolog mode, e.g. **substring/3** will not accept double-quoted items, since they are lists, not ECL^{i}PS^e strings. The built-in **string_list/2** converts between both representations.

A.6.5 Syntax differences

The list below describes the syntax differences between ECL^{i}PS^e and C-Prolog. The following C-Prolog properties are simulated by the compatibility package:

- A blank is not allowed between the functor and the opening bracket.
- single (resp. double) quote must be doubled between single (resp. double) quotes.
- newline is allowed between quotes.
- `$` is a normal character.

The following properties of original C-Prolog are *not* simulated by the compatibility package:

- the symbol `|` is not an atom.
- a clause can not be ended by end of file.
- based integers are not accepted.
- comments are not a delimiter (just ignored).
- `{}` is not an atom.
- `[]` can not be a functor.

A.7 I/O Redirection

This library is loaded with

```
:- lib(fromonto).
```

and defines several primitives that allow to easily redirect the standard user input or output from or to a given file, stream or string. The available primitives are

- Goal **from_file** File
- Goal **onto_file** File
- Goal **from_stream** Stream
- Goal **onto_stream** Stream
- Goal **from_string** String
- Goal **onto_string** String

All input (output) in *Goal* is taken from (to) the specified medium. Example:

```
[eclipse 1]: write(hello) onto_file scratch.

yes.
[eclipse 2]: read(X) from_file scratch.

X = hello
yes.
[eclipse 3]: read(X) from_string "s(a,2,[3])".

X = s(a, 2, [3])
yes.
[eclipse 4]: (write(hello), put(0' ), write(world)) onto_string S.

S = "hello world"
yes.
```

A.8 The Mode Analyser

This library provides a static mode analysis tool for ECL^{i}PS^e programs. It takes as input a Prolog program and a goal pattern, and performs an analysis of the modes that the predicates in the program will be called with, given the specified goal pattern. The mode analyser is loaded with

```
:- lib(modes).
```

The usage is simple. The predicate **read_source/1** is used to read in the source file to be analysed. The predicate **analyze/1** starts the analysis with a given call pattern and prints the results in the form of a compilable mode declaration:

```
[eclipse 2]: read_source(qsort).
reading qsort

yes.
[eclipse 3]: analyze(qsort(++,-)).

% Analysed 3 call patterns of 3 predicates in 0.05 sec
% Number of pending goals: 29
% Global Stack used:       12892
:- mode partition(++, ++, -, -).
:- mode qsort(++, -).
:- mode qsort(++, -, ++).

yes.
[eclipse 4]: analyze(qsort(+,-)).

% Analysed 4 call patterns of 3 predicates in 0.13 sec
% Number of pending goals: 54
% Global Stack used:       35968
:- mode partition(?, ?, -, -).
:- mode qsort(+, -).
:- mode qsort(+, -, +).

yes.
```

Restrictions: Programs that use coroutining features cannot be analysed.

A.9 Parallel Utilities

This library contains parallel versions of common sequential predicates. It is loaded using

```
:- lib(par_util).
```

and it currently contains the following predicates:

par_member(?Element, +List) Parallel version of **member/2**, i.e. selects elements from the given list in parallel. Note that it cannot work backwards and generate lists like **member/2** can, the list must be a proper list.

par_delete(?Element, ?List, ?Rest) Parallel version of **delete/3**.

par_between(+From, +To, ?I) Parallel version of **between/3**. Generates integers between *From* and *To* in parallel. See also **fork/2**, on which it is based.

par_maplist(+Pred, +In, ?Out) Parallel version of **maplist/3**. The semantics is not exactly the same as **maplist/3**: It does not work backwards and it does not cope with aliasing between the *In* and the *Out* list, since it is implemented on top of **findall/3**. There will only be a performance gain if the mapping predicate does enough computation to make the overhead pay off.

Goal1 & Goal2 Parallel AND operator implemented on top of OR-parallelism. This will only pay off for sufficiently coarse-grained computations in *Goal1* and *Goal2*.

A.10 Ptags

This library provides a program that checks the source form of a Prolog program and creates a tags file for use with the UNIX editors ex and vi, similar to *ctags(1)*. The library is loaded using

```
:- lib(ptags).
```

and the predicates **ptags/1**, **ptags/2** and **tags/2** become global. The utility is invoked by

```
:- ptags(+File)
```

or

```
:- ptags(+File, +TagsFile)
```

+TagsFile is the name of the tags file. If *+TagsFile* is omitted, it defaults to `tags`.
The tags file created by the **ptags/1, 2** predicates can be used as a tags file for *vi* or *ex*. A procedures specified as *Name/Arity* can be found using the command

```
:ta Name
```

If there are several procedures with the same name and different arity, the above command will find only one of them. In this case the command

```
:ta Name/Arity
```

should be used. If the clauses for the procedure are not consecutive or if the procedure occurs in more than one file, only one occurrence will be put into the tags file. Which one it will be depends on the file name and on the contents of the line that is being sought by the :ta command.

A.11 Quintus Prolog Compatibility Package

ECL^{i}PS^e includes a Quintus Prolog compatibility package to ease the task of porting Quintus Prolog applications to ECL^{i}PS^e Prolog. This package does not provide the user with a system completely compatible to Quintus Prolog, however it provides most of the Quintus built-in predicates, moreover some of the Quintus library predicates are available in the ECL^{i}PS^e library. This package includes the C-Prolog compatibility package (see Appendix A.6).
Please note that this appendix does not detail the functionality of Quintus Prolog, refer to the Quintus Prolog documentation for this information.

A.11.1 Using the Quintus Prolog compatibility package

The Quintus Prolog compatibility package can be accessed by issuing the directive

```
:- use_module(library(quintus)).
```

The effect of the compatibility library is local to the module where it is loaded (except for the redefinition of some global event handlers). The directive must therefore be given in every module that wants to use the compatibility mode. The Quintus package includes the C-Prolog one.

A.11.2 The Quintus compatibility predicates

The following Quintus predicates which differ from ECL^{i}PS^e predicates are available:

'C'/3 for grammar rules

absolute_file_name/2

atom_chars/2

character_count/2

current_input/1

current_key/2

current_module/2

current_output/1

expand_term/2 This predicate is dummy, since the ECL^{i}PS^e macro facility works on every input term, provided that the flag `macro_expansion` is set to `on`.

float/1

flush_output/1

format/2, 3

gc/0

get0/2 This predicate is identical to **get/2** in ECL^{i}PS^e.

help/1 This is the normal ECL^{i}PS^e **help/1** predicate.

incore/1

integer/2

is_digit/1

is_lower/1

is_upper/1

line_count/2

manual/0

meta_predicate/1 This predicate is not available, as Quintus' method of passing the module information to meta predicates differs substantially from the ECL^{i}PS^e more general concept of *tools* (see Section 9.6.4). Only the operator definition for this functor is available.

module/2

multifile/1 This is implemented by declaring the predicates as dynamic, so to obtain more efficient programs it is better to put all clauses of the same procedure into one file (or to concatenate all files where multifile predicates occur).

no_style_check/1

nogc/0

nospyall/0

number_chars/2

numbervars/3

open_null_stream/1

otherwise/0

portray_clause/1

predicate_property/2 The property *interpreted* is not provided. The property *exported* is returned if the predicate is exported or global. Use of **get_flag/3** should be preferred.

prolog_flag/2, 3 There are some differences in the flags, as they are mostly simulated with the ECL^{i}PS^e flags:

- not all the character escapes used in the Quintus Prolog are available
- **gc_margin** is taken as the ECL^{i}PS^e flag **gc_interval** (see Section 19.2)
- setting **gc_trace** to **on** sets also **gc** to **on**

public/1 synonym for **global/1**

put/2

put_line/1

retractall/1

set_input/1

set_output/1

source_file/1, 2

statistics/0, 2 these predicates are slightly different than in Quintus, see the description of the ECL^{i}PS^e **statistics/2** predicate. The predicate **statistics/2** also accepts all Quintus values in the Quintus mode, but for `stack_shifts` is always returns zeros. **statistics/0** returns only Quintus values when in Quintus mode.

stream_code/2

stream_position/2, 3

style_check/1

term_expansion/2

ttyflush/0, ttyget/1, ttyget0/1, ttynl/0, ttyput/1, ttyskip/1, ttytab/1 these predicates work with the **stdout** stream

unix/1

unknown/2

version/0

Note that **line_position/2** is not implemented. To perform sophisticated output formatting, printf/2,3 or string streams can be used.

A.11.3 Syntax differences

The list below describes the syntax differences between ECL^{i}PS^e and Quintus Prolog. The following properties of Quintus Prolog are simulated by the compatibility package:

- A blank is not allowed between the functor and the opening bracket.
- single (resp. double) quote must be doubled between single (resp. double) quote.
- The symbol `|` (bar) is recognised as an alternative sign for a disjunction and it acts as an operator.
- newline is allowed between quotes.

The following Quintus properties are *not* simulated:

- the symbol `|` is not an atom.
- a clause can not be ended by end of file.

- signed numbers: explicitly positive numbers are structures.
- a real with an exponent must have a floating point.
- a space is allowed before the decimal point and the exponent sign.
- the definition of the escape sequence is more extended than in ECL^{i}PS^e.
- **spy/1** and **nospy/1** accept as arguments lists, rather than comma-separated terms like in ECL^{i}PS^e.

A.12 Scattered

The ECL^{i}PS^e compiler does not allow the clauses for *static* predicates being non-consecutive, i.e. interleaved with clauses for other predicates. The event 134 ”procedure clauses are not consecutive” is raised in such a case. This library provides a means to handle such programs. When compiling from a file, it is enough just to load the library before the compilation. It redefines that handler for the event 134 in such a way that the procedures with non-consecutive procedures are recompiled in one chunk after encountering the end of file.
When not compiling from a file, the non-consecutive clauses have to be declared using the directive **scattered/1**. This declaration has to precede any clauses of the predicate.

```
:- lib(scattered).
:- scattered p/3, q/1.
```

Note that this applies to predicates whose clauses are non-consecutive, but in a single file. Predicates that are spread over multiple files still have to be declared as dynamic

A.13 SICStus Prolog Compatibility Package

ECL^{i}PS^e includes a SICStus Prolog compatibility package to ease the task of porting SICStus Prolog applications to ECL^{i}PS^e Prolog. This package includes the C-Prolog compatibility package (see Appendix A.6) and the Quintus-Prolog compatibility package (see Appendix A.11). Please note that this appendix does not detail the functionality of SICStus Prolog, refer to the SICStus Prolog documentation for this information.

A.13.1 Using the SICStus Prolog compatibility package

The SICStus Prolog compatibility package can be accessed by issuing the directive

```
:- use_module(library(sicstus)).
```

The effect of the compatibility library is local to the module where it is loaded (except for the redefinition of some global event handlers). The directive must therefore be given in every module that wants to use the compatibility mode.

A.13.2 The SICStus compatibility predicates

The following predicates are available (in addition the the ones from the quintus library):

block/1

call_residue/2 This is only approximate, the variables in the second argument are dummies.

dif/2

fcompile/1

freeze/2

frozen/2

load/1

module/2

on_exception/3

raise_exception/1

when/2

A.13.3 Sockets library

A sockets library is provided for compatibility with the sockets manipulation predicates of SICStus. To use these predicates, the sockets library has to be loaded:

```
:- use_module(library(sockets)).
```

For SICStus 3.0, the sockets predicates are also in a `sockets` library, so no changes are needed to load the library. However, for older versions of SICStus, the predicates are available as built-ins, and no library has to be loaded. So if the code is written for older versions of SICStus, then the above line has to be added.
The sockets library can be used independently of the `sicstus` library. Note also that ECL^{i}PS^e also provides its own socket manipulation predicates (see Chapter 21) that provides similar functionalities to the sockets library.
All the predicates in SICStus' socket library are supported:

socket/2

socket_bind/2

socket_connect/3

socket_listen/2

socket_accept/2

socket_select/5

current_host/1 This does not attempt to always return the qualified name of the host machine.

stream_select/3 This is a built-in for SICStus, and not part of its socket library. It is currently supported in ECL^{i}PS^e as part of the sockets library.

A.13.4 Syntax differences

Since the SICStus package contains the Quintus one, the syntax differences are the same, see Section A.11.3.

A.14 Utility Libraries

These libraries contain various predicates which were not included into ECL^{i}PS^e built-in predicates but which are often useful for the user.

A.14.1 Util

The library is loaded using

```
:- lib(util).
```

and it current contains the following predicates:

add_path(+Directory) The argument must be a string which is the name of a directory. This directory will be added at the beginning of the *library* path.

add_suffix(+Suffix) The argument must be a string which denotes a file name suffix that will be added at the beginning of the *prolog_suffix* list.

between(+From, +To, ?I) Succeeds if *From* and *To* are integers and *I* unifies with a number between the two. On backtracking it generates all values for *I* starting from *From* onwards.

compiled List all currently compiled files and indicate if they have been modified since they were compiled.

list_error(+String, -ErrNo, -ErrMsg) Find the event number whose message contains the specified substring.

ptags_all Make a ptags file for all .pl files in the current directory.

read_line(-String) Defined as

```
read_line(String) :-
        read_string(input, "\n", Length, String).
```

It reads a line from the `input` stream into a string.

read_line(+Stream, -String) Like the previous but reads from a specified stream.

stream(+Stream) List all information about the specified I/O stream.

streams List all currently opened streams.

time(+Goal) Call the goal *Goal*, measure its runtime (cputime) and print the result after success or failure.

write_history Writes the current history into the *.eclipse_history* file.

A.14.2 Define

This library allows to define macro transformation similar to the #define command of the C preprocessor.

A.14.3 Numbervars

Implements the **numbervars(Term, From, To)** predicate of C-Prolog. *Term* is any term, *From* and *To* are integer numbers. All variables in *Term* are instantiated to terms of the form

$VAR(N)

where N is an integer number. The first encountered variable will be coded by the number *From*, on exit *To* is instantiated to the next unused number.
This predicate can thus be used to encode nonground term using a ground representation. Note that metaterms can be used for the same purpose, but their use is both more efficient and more general, because the variables are not actually instantiated and so they can be used again as variables when needed.

A.14.4 Apply

Implements the apply/2 predicate.

A.14.5 Lips

Measure the system's speed in logical inferences per second, using the infamous naive reverse program.

A.14.6 Anti_unify

Defines the predicate **anti_unify/3** which computes the least common generalisation of two terms.

A.14.7 Spell

The effect of loading this library is to modify the event handler for calling an undefined procedure. A spelling correction algorithm is used to see if the cause was a misspelling of an existing predicate.

A.14.8 Rationals

This library should be loaded when working with rational arithmetic. It defines macros to accept and print rationals in the form Num/Den (in addition to the standard syntax Num_Den). Also, it sets the global flag `prefer_rationals`, so that the arithmetic division function / yields a rational rather than a floating point result.

Appendix B

Syntax

B.1 Introduction

This chapter provides a definition of the syntax of the ECL^{i}PS^e Prolog language. A complete specification of the syntax is provided and comparison to other commercial Prolog systems are made. The ECL^{i}PS^e syntax is based on that of Edinburgh Prolog ([1]).

B.2 Notation

The following notation is used in the syntax specification in this chapter:

- a term_h is a term which is the head of the clause.
- a term_h(N) is a term_h of maximum precedence N.
- a term_g is a term which is a goal (body) of the clause.
- a term_g(N) is a term_g of maximum precedence N.
- a term_a is a term which is an argument of a compound term or a list.
- a term(N) can be any term (term_h, term_a or term_h) of maximum precedence N.
- fx(N) is a prefix operator of precedence N which is not right associative.
- fy(N) is a prefix operator of precedence N which is right associative.
- similar definitions apply for infix (xfx, xfy, yfx) and postfix (xf, yf) operators.

B.2.1 Character Classes

The following character classes exist:

Character Class	**Notation Used**	**Valid Characters**
upper_case	UC	all upper case letters
underline	UL	`_`
lower_case	LC	all lower case letters
digit	N	digits
blank_space	BS	space, tab and nonprintable ASCII characters
end_of_line	NL	carriage return and line feed
atom_quote	AQ	`'`
string_quote	SQ	`"`
list_quote	LQ	
radix	RA	
ascii	AS	
solo	SL	`( ) ] }`
special	SP	`! , ; [ { |`
line_comment	CM	`%`
escape	ESC	`\`
first_comment	CM1	`/`
second_comment	CM2	`*`
symbol	SY	`# + - . : < = > ? @ ^ ` ` ` ~ $ &`

The character class of any character can be modified by the built-in predicate **set_chtab/2**. Tokens can be read with the predicate **read_token/3**.

B.2.2 Groups of characters

Group Type	**Notation**	**Valid Characters**
alphanumerical	ALP	UC UL LC N
delimiter	DE	`) } ] , |`
any character	ANY	
non escape or newline	NEN	any character except escape and newline
sign	SGN	`+ -`

B.2.3 Valid Tokens

The valid tokens are described below :

Constants

1. **atoms**

```
ATOM    = (LC ALP*)
        | (SY | CM1 | CM2 | ESC)+
        | (AQ (NEN | ESC ANY)* AQ)
        | |
        | ;
        | []
```

```
          | {}
          | !
```

2. **numbers**

   (a) **integers**

   ```
   INT = [SGN] BS* N+
   ```

   (b) **based integers**

   ```
   INTBAS = N+ (AQ | RA) (N | LC | UC)+
   ```

   The base must be an integer between 1 and 36 included, the value being valid for this base.

   (c) **ASCII codes**

   ```
   ASCII = 0 (AQ | RA) ANY | AS ANY
   ```

   The value of the integer is the ASCII code of the last character.

   (d) **rationals**

   ```
   RAT = [SGN] BS* N+ UL N+
   ```

   (e) **reals**

   ```
   REAL = [SGN] BS* N+ . N+ [ (e | E) [SGN] N+ | Inf ]
        | [SGN] BS* N+        (e | E) [SGN] N+
   ```

   checks are performed that the numbers are in a valid range.

3. **strings**

   ```
   STRING = SQ (NEN | ESC ANY | SQ BS* SQ)* SQ
   ```

4. **lists of ASCII codes**

   ```
   LIST = LQ (NEN | ESC ANY)* LQ
   ```

Variables

```
VAR = (UC | UL) ALP*
```

End of clause

```
EOCL = . (BS | NL | <end of file>) | <end of file>
```

B.2.4 Escape Sequences within Strings and Atoms

Within atoms and strings, the escape sequences (ESC ANY) are interpreted : if ANY is one of the characters described in the table below, the sequence ESC ANY generates a special character. Otherwise, the lexical analyser just ignores the escape character (ie `"\a"` is the same as `"a"`).

Escape Character	**Result**
`b`	backspace
`f`	line feed
`n`	newline
`r`	carriage return
`t`	tabulation
newline	ignored
three octal digits	character whose ASCII code is the octal value (any 8-bit value)

B.3 Formal definition of clause syntax

What follows is the specification of the syntax. The terminal symbols are written in UPPER CASE or as the character sequence they consist of.

```
program                 ::=     clause EOCL
                          |     clause EOCL program

clause                  ::=     head
                          |     head rulech goals
                          |     rulech goals

head                    ::=     term_h

goals                   ::=     term_g
                          |     goals , goals
                          |     goals ; goals
                          |     goals -> goals
                          |     goals -> goals ; body

term_h                  ::=     term_h(0)
                          |     term(1200)

term_g                  ::=     term_g(0)
                          |     term(1200)

term(0)                 ::=      VAR              /* not a term_h */
                          |      metaterm         /* not a term_h */
                          |      ATOM
                          |      structure
                          |      subscript
                          |      list
```

```
                        |        STRING         /* not a term_h nor a term_g */
                        |        number         /* not a term_h nor a term_g */
                        |        bterm

term(N)                 ::=     term(0)
                        |       prefix_expression(N)
                        |       infix_expression(N)
                        |       postfix_expression(N)

prefix_expression(N)    ::=     fx(N) term(N-1)
                        |       fy(N) term(N)

infix_expression(N)     ::=     term(N-1)       xfx(N)  term(N-1)
                        |       term(N)         yfx(N)  term(N-1)
                        |       term(N-1)       xfy(N)  term(N)

postfix_expression(N)   ::=     term(N-1)       xf(N)
                        |       term(N) yf(N)

metaterm                ::=     VAR { meta_attributes }

meta_attributes         ::=     attribute
                        |       attribute , meta_attributes

attribute               ::=     qualified_attribute
                        |       nonqualified_attribute

qualified_attribute     ::=     ATOM : nonqualified_attribute

nonqualified_attribute  ::=     term_a

structure               ::=     functor ( termlist )

subscript               ::=     structure list
                        |       VAR list

termlist                ::=      term_a
                        |        term_a , termlist

list                    ::=     [ listexpr ]
                        |       .(term_a, term_a)

listexpr                ::=     term_a
                        |       term_a | term_a
                        |       term_a , listexpr

term_a                  ::=     term(1200)
```

```
                            /* Note: it depends on syntax_options */

number               ::=    INT
                      |     INTBAS
                      |     RAT
                      |     REAL

bterm                ::=    ( clause )
                      |     { clause }

functor              ::=    ATOM                    /* arity > 0 */

rulech               ::=    :-
                      |     ?-
```

B.3.1 Comments

Comments can be full line comments, that is enclosed by CM1-CM2 and CM2-CM1, or end of line comments, that is enclosed by CM and NL. They behave as separators.

B.3.2 Operators

In Prolog, the user is able to modify the syntax dynamically by explicitly declaring new operators. The builtin **op/3** performs this task. As in Edinburgh Prolog, a lower precedence value means that the operator binds stronger (1 strongest, 1200 weakest).
Multiple definitions are handled in the following way:

- The multiple definitions prefix/infix and prefix/postfix are allowed.
- The multiple definition infix/postfix is allowed. The resulting ambiguity is resolved as follows. When such an operator is followed by a delimiter, it is parsed as postfix operator, which is always correct. Otherwise, it is taken to be the infix operator regardless of associativity or precedence. This may yield an unexpected effect, in which case parentheses must be used for disambiguation.
- In case two definitions for the same associativity class are made, the last one erases the previous one.

When entering ECL^{i}PS^e, some operators are pre-defined (see Appendix C on page 229). They may be redefined if desired.
Note that parentheses are used to build expressions with precedence zero and thus to override operator declarations. Quotes, on the other hand, are used to build atoms from characters with different or mixed character classes; they do not change the precedence of operators.
Operators are by default local to a module unless they are declared with **global_op/3**. The operator visible in a module is either the local one (if any) or the global one.

B.3.3 Ambiguity

The prolog syntax allows some ambiguity when parsing operators. To solve it, we could do a lookahead(N) or a backtracking. ECL^{i}PS^e uses a lookahead(1), which allows to solve the main cases of ambiguity.
If a prefix operator is followed by an infix operator the system decides which token is an atom and which is the operator by examining the precedence. The token with the higher precedence becomes the operator. If the precedences are the same, the associativity of the operators is examined. If the prefix operator is *fy*, it becomes the operator. If the infix operator is *yfx* or *yfy* it becomes the operator. Otherwise an error is raised.

B.4 Syntax Differences between ECL^{i}PS^e and other Prologs

B.4.1 Properties of ECL^{i}PS^e

Some particularities exist in the default syntax of ECL^{i}PS^e. Most of these properties can be configured and are in fact changed by the compatibility packages.

- Syntax for the special types of ECL^{i}PS^e: metaterms and rationals.
- A blank space is allowed between the functor and the following opening parenthesis (see below)
- end of file is accepted as fullstop
- operators with precedence higher than 1000 are allowed in a comma-separated list of terms, i.e. structure arguments and lists (see below). The ambiguity is resolved by considering commas as argument separators rather than operators inside the term. Thus e.g.

  ```
  p(a :- b, c)
  ```

  is accepted and parsed as **p/2**.
- double-quoted items are parsed as strings, not as lists (controlled via **set_chtab/2**)
- empty brackets are not parsed as the atom '[]' when there are layout charactes between the brackets (controlled by syntax option **blanks_in_nil**)
- newline is not allowed inside quotes (controlled by syntax option **nl_in_quotes**)

B.4.2 Changing the Parser behaviour

Some of these properties can be changed by selecting one of the following syntax options using **set_flag/2**. The following options exist:

blanks_in_nil allow blanks between the brackets in [].

limit_arg_precedence do not allow terms with a precedence higher than 999 as structure arguments, unless parenthesised.

nested_comments allow bracketed comments to be nested.

nl_in_quotes allow newlines to occur inside quotes.

no_blanks do not allow blanks between functor an opening parenthesis

no_other_quotes do not allow string quotes inside atom quotes and vice versa.

$VAR terms of the form '$VAR'(N) are printed in a special way by all the predicates that obey operator declarations (i.e. write, writeq, print and partly printf). '$VAR'(0) is printed as A, '$VAR'(25) as Z, '$VAR'(26) as A1 and so on. When the argument is an atom or a string, just this argument is printed.

based_bignums Allow base notation even to write integers longer than the wordsize (i.e. they are always positive).

Syntax option settings are local to the module where they were performed. They are switched on and off as follows:

```
:- set_flag(syntax_option, nl_in_quotes).
:- set_flag(syntax_option, not no_blanks).
```

Appendix C

Operators

The following table summarises the predefined global operators in ECL^{i}PS^e. They can be redefined or erased on a per-module basis by hiding them with a user-defined local operator using **op/3**. It is not recommended (but possible) to use **global_op/3** to change their global definition.

Prec	Assoc	Operators
1200	xfx	[-->, :-, ?-, if]
1200	fx	[:-, ?-]
1190	fy	[help]
1190	fx	[delay]
1180	fx	[-?->]
1170	fx	[if]
1160	xfx	[else]
1150	xfx	[then]
1100	xfy	[;, do, '|']
1050	xfy	[->]
1050	xfx	[from]
1050	fy	[import]
1000	xfy	[,]
1000	fy	[abolish, demon, dynamic, export, global, listing, local, mode, nospy, parallel, skipped, spy, traceable, unskipped, untraceable]
900	fy	[\+, not, once, ~]
700	xfx	[#<, #<=, #=, #=<, #>, #>=, #\=, ::, <, =, =.., =:=, =<, ==, =\=, >, >=, @<, @=<, @>, @>=, \=, \==, is, ~=]
650	xfx	[@, of, with]
600	xfy	[:]
600	xfx	[..]
500	yfx	[+, -, /\, \/]
500	fx	[+, -]
400	yfx	[*, /, //, <<, >>]
300	xfx	[mod]

```
200   xfy   [^]
200    fx   [\]
```

Appendix D

Events

We list here the ECL^{i}PS^e event types together with the default event handlers and their description. Unless otherwise specified, the arguments that the system passes to the event handler are

First Argument	Second Argument	Third Argument
Event number	Culprit goal	Caller Module

If the caller module is unknown, a free variable is passed.

D.1 Event Types

D.1.1 Argument Types and Values

Event	Event Type	Default Event Handler
1	general error	error_handler / 2
2	term of an unknown type	error_handler / 2
4	instantiation fault	error_handler / 2
5	type error	error_handler / 2
6	out of range	error_handler / 2
7	string contains unexpected characters	error_handler / 2
8	bad argument list	error_handler / 2

D.1.2 Arithmetic, Environment

Event	Event Type	Default Event Handler
15	creating parallel choice point	fail / 0
16	failing to parallel choice point	fail / 0
17	recomputation failed	error_handler / 2
20	arithmetic exception	error_handler / 2
21	undefined arithmetic expression	error_handler / 3
23	comparison trap	compare_handler / 3
24	number expected	error_handler / 2
25	integer overflow	integer_overflow_handler / 2
30	trying to write a read-only flag	error_handler / 2
31	arity limit exceeded	error_handler / 2
32	no handler for event	warning_handler / 2
33	event queue overflow	error_handler / 2

D.1.3 Data and Memory Areas, Predicates, Operators

Event	Event Type	Default Event Handler
40	stale object handle	error_handler / 2
41	array or global variable does not exist	undef_array_handler / 3
42	redefining an existing array	make_array_handler / 3
43	multiple definition postfix/infix	error_handler / 2
44	local record already exists	error_handler / 2
45	local record does not exist	error_handler / 2
60	accessing an undefined procedure from	error_handler / 3
61	redefining a tool procedure	warning_handler / 2
62	inconsistent procedure redefinition	error_handler / 3
63	procedure not dynamic	error_handler / 3
64	procedure already dynamic	dynamic_handler / 3
65	procedure already defined	error_handler / 3
66	trying to modify a system predicate	error_handler / 3
67	procedure is not yet loaded	error_handler / 3
68	calling an undefined procedure	call_handler / 3
69	autoload event	autoload_handler / 3
70	accessing an undefined dynamic procedure	undef_dynamic_handler / 3
71	procedure already parallel	error_handler / 2
72	accessing an undefined operator	error_handler / 2
73	redefining an existing operator	true / 0
74	hiding an existing global operator	true / 0
75	using an obsolete built-in	obsolete_handler / 3
76	predicate declared but not defined	warning_handler / 3
77	predicate used but not declared or defined	warning_handler / 3

D.1.4 Modules, Visibility

Event	Event Type	Default Event Handler
80	not a module	error_handler / 2
81	module/1 can appear only as a directive	error_handler / 2
82	trying to access a locked module	locked_access_handler / 2
83	creating a new module	warning_handler / 2
87	procedure is already local	warning_handler / 2
88	procedure is already exported	warning_handler / 2
89	procedure is already global	warning_handler / 2
91	not a tool procedure	error_handler / 2
92	procedure is already local	error_handler / 2
93	procedure is already exported	error_handler / 2
94	procedure is already imported	error_handler / 2
95	procedure is already global	error_handler / 2
96	procedure is already imported	warning_handler / 2
97	module already exists	error_handler / 2
98	key not correct	error_handler / 2
99	name clash with other imported module(s)	warning_handler / 2
100	accessing a procedure defined in another module	undef_dynamic_handler / 3
101	trying to erase a module from itself	error_handler / 2

D.1.5 Syntax Errors, Parsing

Event	Event Type	Default Event Handler
110	syntax error:	parser_error_handler / 2
111	syntax error: list tail ended improperly	parser_error_handler / 2
112	syntax error: illegal character in a quoted token	parser_error_handler / 2
113	syntax error: precedence too high	parser_error_handler / 2
114	syntax error: unexpected token	parser_error_handler / 2
115	syntax error: unexpected end of file	parser_error_handler / 2
116	syntax error: numeric constant out of range	parser_error_handler / 2
117	syntax error: bracket necessary	parser_error_handler / 2
118	syntax error: unexpected fullstop	parser_error_handler / 2
119	syntax error: postfix/infix operator expected	parser_error_handler / 2
120	syntax error: wrong solo char	parser_error_handler / 2
121	syntax error: space between functor and open bracket	parser_error_handler / 2
122	syntax error: variable with multiple attributes	parser_error_handler / 2
124	syntax error : prefix operator followed by infix operator	parser_error_handler / 2
125	syntax error : unexpected closing bracket	parser_error_handler / 2
126	syntax error : grammar rule head is not valid	parser_error_handler / 2
127	syntax error : grammar rule body is not valid	parser_error_handler / 2
128	syntax error : in source transformation	parser_error_handler / 2
129	syntax error: source transformation floundered	parser_error_handler / 2

D.1.6 Compilation, Run-Time System, Execution

Event	Event Type	Default Event Handler
130	syntax error: illegal head	compiler_error_handler / 2
131	syntax error: illegal goal	compiler_error_handler / 2
132	syntax error: term of an unknown type	compiler_error_handler / 2
133	loading the library	message_handler / 2
134	procedure clauses are not consecutive	compiler_error_handler / 2
135	trying to redefine a protected procedure	compiler_error_handler / 2
136	redefining a system predicate	warning_handler / 2
137	trying to redefine a procedure with another type	compiler_error_handler / 2
138	singleton local variable in do-loop	singleton_in_loop / 2
139	compiled or dumped file message	compiled_file_handler / 3
140	undefined instruction	error_handler / 2
141	unimplemented functionality	error_handler / 2
142	built-in predicate not available on this system	error_handler / 2
143	compiled query failed	compiler_error_handler / 2
144	a cut is not allowed in a condition	compiler_error_handler / 2
145	procedure being redefined in another file	redef_other_file_handler / 2
146	start of compilation	true / 0
147	compilation aborted	compiler_abort_handler / 2
148	bad pragma	compiler_error_handler / 2

The handlers for these events receive the following arguments:

Event	Second Argument	Third Argument
130	Culprit clause	Module
131	Culprit clause	Module
132	Culprit clause	Module
133	Library name (string)	undefined
134	Procedure Name/Arity	Module
135	Procedure Name/Arity	Module
136	Procedure Name/Arity	Module
137	Procedure Name/Arity	Module
139	(File, Size, Time), see below	Module
140	'Emulate'	undefined
141	Goal	Module
142	Goal	Module
143	Goal	Module
144	Goal (if an execution error) or Culprit clause (if compiler error)	Module
145	(Name/Arity, OldFile, NewFile)	Module
146	File	Module
147	File	

The second argument for the event 139 depends on the predicate where it was raised:

- **compile/1, 2** - (file name, code size, compile time)
- **dump/1** - (input file, dump file, 'dumped')
- **compile_term/1** - ('term', code size, compile time)
- **compile_stream/1** - ('string', code size, compile time) with a string stream
- **compile_stream/1** - (file name, code size, compile time) with a stream associated to a file

D.1.7 Top-Level

Event	Event Type	Default Event Handler
150	start of eclipse execution	sepia_start / 0
151	eclipse restart	true / 0
152	end of eclipse execution	sepia_end / 0
153	toplevel: print prompt	sepia_toplevel_prompt / 2
154	toplevel: start of query execution	true / 0
155	toplevel: print values	sepia_print_values / 3
156	toplevel: print answer	sepia_answer / 2
157	error exit	error_exit / 0
158	toplevel: entering break level	sepia_start_break / 3
159	toplevel: leaving break level	sepia_end_break / 3

These events are not errors but rather hooks to allow users to modify the behaviour of the ECL^{i}PS^e toplevel. Therefore the arguments that are passed to the handler are not the erroneous goal and the caller module but defined as follows:

Event	Second Argument	Third Argument
150	A free variable. If the handler binds the variable to an atom, this name is used as the toplevel module name	undefined
151	undefined	undefined
152	The argument is the number that ECL^{i}PS^e will return to the operating system	undefined
153	current toplevel module	current toplevel module
154	a structure of the form *goal(Goal, VarList, NewGoal, NewVarList)*, where Goal is the goal that is about to be executed and VarList is the list that associates the variables in Goal with their names (like in **readvar/3**). NewGoal and NewVarList are free variables. If the handler binds NewVarList then the toplevel will use NewGoal and NewVarList to replace Goal and VarList in the current query.	current toplevel module
155	A list associating the variable names with their values after the query has been executed.	current toplevel module
156	An atom stating the answer to the query that was just executed. The possible values are: `yes`, `last_yes` or `no` if the query had no variables, `more_answers`, `last_answer` if the query contained variables and bindings were printed, `no_answer` if a query containing variables failed.	current toplevel module
157	undefined	undefined
158	break level	current toplevel module
159	break level	current toplevel module

When the handler for event 152 "end of eclipse execution" fails, ECL^{i}PS^e is not exited, but re-started. This can prevent accidental exits from the system.

D.1.8 Macro Transformation Errors, Lexical Analyser

Event	Event Type	Default Event Handler
160	global macro transformation already exists	error_handler / 3
161	macro transformation already defined in this module	macro_handler / 3
162	no macro transformation defined in this module	warning_handler / 2
163	single atom or string quote character may not be redefined	error_handler / 2
164	toplevel: print banner	sepia_banner / 2

The event 164 is raised whenever the toplevel loop is restarted.

Event	Second Argument	Third Argument
164	the banner string	

D.1.9 I/O, Operating System, External Interface

Event	**Event Type**	**Default Event Handler**
170	system interface error	system_error_handler / 2
171	File does not exist :	error_handler / 2
172	File is not open :	error_handler / 2
173	library not found	error_handler / 2
174	child process terminated due to signal	error_handler / 2
175	child process stopped	error_handler / 2
176	message passing error	error_handler / 2
190	end of file reached	eof_handler / 3
191	output error	output_error_handler / 2
192	illegal stream mode	error_handler / 2
193	illegal stream specification	error_handler / 2
194	too many symbolic names of a stream	error_handler / 2
195	yield on flush	io_yield_handler / 2
196	trying to modify a system stream	close_handler / 2
197	use 'input' or 'output' instead of 'user'	error_handler / 2
198	reading past the file end	past_eof_handler / 2
210	Remember() not inside a backtracking predicate	error_handler / 2
211	External function does not exist	error_handler / 2
212	External function returned invalid code	error_handler / 2

D.1.10 Advanced Features, Extensions, Debugging

Event	**Event Type**	**Default Event Handler**
230	exiting to an undefined tag	error_handler / 2
231	default help/0 message	fail / 0
250	trying to change the debugger model while debugging	error_handler / 2
251	debugger user event	debug_handler / 3
252	debugger port event	trace_line_handler_tty / 2
253	debugger call event	ncall / 2
254	debugger exit event	nexit / 1
255	debugger redo event	redo / 5
256	debugger delay event	ndelay / 2
257	debugger wake event	resume / 2
258	debugger notify event	true / 0
260	unexpected end of file	error_handler / 2
261	invalid saved state	error_handler / 2
262	can not allocate required space	error_handler / 2
263	can not save or restore from another break level than level 0	error_handler / 2
264	not a dump file	compiled_file_handler / 3
265	bad dump file version	compiled_file_handler / 3
267	predicate not implemented in this version	error_handler / 2
268	predicate not supported in parallel session	error_handler / 2

These handlers receive special arguments:

Event	**Second Argument**	**Third Argument**
251	trace(Invoc, Depth, Port, Goal)	Module
252	trace(Invoc, Depth, Port, Goal)	Module
264	(File, [], [])	undefined
265	(File, [], [])	undefined

Event	Event Type	Default Event Handler
270	undefined variable attribute	error_handler / 2
271	bad format of the variable attribute	error_handler / 2
272	delay clause may cause indefinite delay	warning_handler / 2
273	delayed goals left	delayed_goals_handler / 3
274	stack of woken lists empty	error_handler / 2
280	Found a solution with cost	error_handler / 2

The handlers for these events receive the following arguments:

Event	Second Argument	Third Argument
272	Culprit clause	Module
273	list of sleeping suspensions	undefined
280	Cost, Goal	undefined

D.2 ECL^{i}PS^e Fatal Errors

A fatal error cannot be caught by the user. When they occur, the system performs a warm restart. The following fatal errors may be generated by ECL^{i}PS^e:

***** Fatal error: Out of memory - no more swap space** The available memory (usually swap space) on the computer has been used up either by the application or some external process.

***** Fatal error: Internal error - memory corrupted** This signals an inconsistency in the system's internal data structures. The reason can be either a bug in the ECL^{i}PS^e system itself or in an external predicate provided by the user.

D.3 User-Defined Events

Currently the user defined events start at 340 (this may change with future releases). User event numbers are returned using **define_error/2**. The default handler is **error_handler/2**. See section 14.2.3 for more details.

D.4 System Event Handlers

In the tables above the default event handlers for all the events are given. Here follows a short description of these handlers. Some of them only print error massages. Those can easily be redefined by the user. Others do more complex things to achieve a certain behaviour of the ECL^{i}PS^e system. If those are redefined by the user, this may have unexpected results. Note that the default handler can always be called (even while the active handler is redefined) by using

```
error(default(Number), Goal [, Module])
```

It is therefore not necessary to know the names of the default handlers. Moreover, most of them are not accessible for the user.

close_handler/2 prevents system stream from being closed. If an attempt is made to close a stream that a system stream is redirected to, the system stream is first reset to its standard value.

autoload_handler/3 Load the appropriate library and recall the autoloaded goal.

compare_handler/3 applies arithmetic evaluation to the first two arguments of the goal, then re-calls the culprit goal with the evaluated arguments.

compiler_error_handler/2 prints error message with relevant line, then fails.

compiled_file_handler/2 prints the message about compiled or dumped file, possible with size and time indication

compiler_abort_handler/2 prints the error message with the file name. If it can find out the line number, it is printed as well.

compiler_warning_handler/2 prints error message with relevant line, then succeeds.

delayed_goals_handler/2 prints a list of delayed goals and succeeds.

dynamic_handler/3 retracts all clauses of the predicate that is already dynamic and succeeds.

eof_handler/2 takes the appropriate action for reaching end of file, depending on the culprit goal, e.g. binding the result to **end_of_file** if the goal was **read/1**. The handler fails for unknown goals.

error_exit/0 Calls **user_exit/0** if it exists globally, else aborts.

error_handler/2 prints the error message and the culprit. Then it raises the event 157 (error exit) which by default aborts via **exit_block(abort)**.

error_handler/3 used for errors inside tools. It is like **error_handler/2** but it also prints the module used by the culprit.

integer_overflow_handler/2 redo an overflowed word-sized-integer arithmetic operation with bignums.

locked_access_handler/2 allows certain goals to be executed in spite of the module lock.

macro_handler/3 prints a warning and redefines a macro.

make_array_handler/3 If the error number is 42 (redefining an existing array), it prints the warning, erases the existing array and replaces it by a new one. Otherwise, it calls the default handler.

meta_unify/1 The default handler for metaterm unification. It wakes the delayed goals.

sepia_start_break/3 prints the message "Entering break level N".

sepia_end_break/3 prints the message "Leaving break level N" and resets the toplevel module in case it was changed inside the break level.

sepia_banner/2 prints its second argument on the `toplevel_output` stream, flushes the stream and succeeds.

message_handler/2 prints the error message followed by the second argument onto toplevel_output and succeeds. Note that the second argument is not necessarily the culprit goal, but rather just a string to be printed. This handler is used for events which are not errors.

output_error_handler/2 closes a related stream if necessary and calls **system_error_handler/2**.

parser_error_handler/1 prints the faulty input line and the corresponding error message, then fails. Used when the culprit is not important and when no abort should occur.

past_eof_handler/2 closes the stream that has been read past end of file, then calls **error_handler/2**.

sepia_answer/2 Issues the yes/no/more answer and prompts for input in the latter case.

sepia_end/0 Calls **user_end/0**, if it exists globally, else does nothing.

sepia_print_values/3 Standard ECL^{i}PS^e results printing routine, i.e. prints variable bindings and delayed goals.

sepia_start/0 Calls **user_start/0**, if it exists globally, else does nothing.

sepia_toplevel_prompt/2 Prints the standard ECL^{i}PS^e toplevel prompt.

system_error_handler/2 gets the operating system error number (from `errno`) and prints the corresponding error message. Then it raises the event 157 (error exit) which by default aborts via **exit_block(abort)**.

undef_array_handler/3 If the culprit was **setval/2** with an atom as first argument, a global variable of that name is created using **make_array/1** and the culprit is re-called. Otherwise like **error_handler/2**.

undef_dynamic_handler/3 when a non-dynamic clause has been asserted, it makes it dynamic (if possible), then asserts it.

warning_handler/2 prints the error message and the culprit and succeeds.

D.5 System Interrupt Handlers

Some of the signals (interrupts) are handler by built-in predicates **halt/0**, **default/0**, **abort/0** and **true/0**, others have special handlers:

interrupt_prolog/0 asks the user what to do - abort, start a break level, debug, continue or exit.

it_handler/0 only prints the signal number to the `error` stream.

it_overflow/0 prints the message "Segmentation violation - maybe machine stack overflow" and makes a warm restart.

it_reset/0 makes a warm restart.

Appendix E

Protected Procedures

Thanks to the module system, the user can normally redefine every global system built-in at any time. However, some built-ins are treated specially by the compiler. They are classified as **protected** and listed below. The compiler has to be informed of their redefinition before compiling the first call to such a predicate. A **local**-declaration at the beginning of the module is sufficient for this purpose. Otherwise, an attempt to redefine one of the following predicates results in an error message.

!/0
,/2
- >/2
:/2
;/2
</2
=/2
=:=/2
=</2
==/2
= \ =/2
>/2
>=/2
\+/1
\ =/2
\ ==/2
add_attribute/2
add_attribute/3
arg/3
atom/1
atomic/1
call/2
call_explicit/2
compound/1
delay/1
fail/0
fail_if/1
free/1
insert_suspension/3
insert_suspension/4
integer/1
is/2
is_suspension/1
make_suspension/3
meta/1
nonground/1
nonvar/1
not/1
number/1
once/1
rational/1
real/1
string/1
true/0
var/1
~=/2

Appendix F

Global Flags

ECL^{i}PS^e has a lot of options and different modes of execution, some of which are controlled by the global flags described below. The current value of every flag can be retrieved using get_flag(+FlagName, ?Value), writeable flags can be set using set_flag(+FlagName, +NewValue). The built-in **env/0** prints a list of all the flags and their current setting.

all_dynamic **Access mode :** read/write

Type : Atomic constant `on` or `off`

Default : `off`

Description : Specifies whether all clauses compiled are dynamic or not.

break_level **Access mode :** read/write

Type : Integer

Description : Specifies current nesting level of recursive top-level loops.

coroutine **Access mode :** read/write

Type : Atomic constant `on` or `off`

Default : configuration dependent

Description : Specifies whether built-in predicates delay or whether they raise instantiation faults.

cwd **Access mode :** read/write

Type : String

Description : Specifies the name of the current working directory. May also be set using **cd/1** or read using **getcwd/1**.

debug_compile **Access mode :** read/write

Type : Atomic constant `on` or `off`

Default : `on`

Description : Specifies whether clauses are compiled for debugging or not. May be set on by **dbgcomp/0** or off by **nodbgcomp/0**.

debugging **Access mode :** read-only

Type : Atomic constant `nodebug`, `creep`, `leap`, `skip`, `jump_invoc`, `jump_level`, `var_skip`, `woke_skip`, `backtrack_skip` or `same_port_skip`

Default : `nodebug`

Description : Specifies whether debugging is disabled (`nodebug` value) or enabled. **trace/0** sets the `creep` mode, **debug/0** sets the `leap` mode, the other modes are set with the debugger options.

debugger_model Access mode : read/write

Type : Atomic constant `sepia`, `byrd` or `kcm`

Default : `sepia`

Description : select one of the available debugger models. They differ in memory consumption, retained information about exited subgoals and instantiations shown at FAIL ports. Refer to the debugger description for details.

dfid_compile Access mode : read/write

Type : Atomic constant `on` or `off`

Default : `off`

Description : When on, the compiler will generate code that keeps track of the number of ancestors of each goal. This is used by **library(dfid)** to perform bounded depth-first search and iterative deepening.

enable_interrupts Access mode : read/write

Type : Atomic constant `on` or `off`

Default : `on`

Description : If `on`, interrupts are recognised and processed as they occur. If `off`, interrupts are entered into a delay queue and processed only when the flag is switched back to `on`. Interrupts should be disabled only for short periods of time in order to keep the system's interrupt response time short.

extension Access mode : read-only

Type : Atomic constant.

Default : configuration dependent

Description : Specifies which extensions are available in the system. This flag may contain multiple values and will return them on backtracking.

float_precision Access mode : read/write

Type : Atomic constant `single` or `double`

Default : `single`

Description : Specifies whether the system uses single or double precision floating point numbers. Should be set at the beginning of a session.

gc Access mode : read/write

Type : Atomic constant `on`, `verbose` or `off`

Default : `on`

Description : Specifies whether garbage collection is enabled (`on`), disabled (`off`) or enabled and reports every collection on `toplevel_output` (`verbose`).

gc_interval Access mode : read/write

Type : Integer

Default : 1/8 of global stack size

Description : Specify how often the collector is invoked. Roughly, the argument specifies how many bytes a program can use up before a garbage collection is triggered. If the garbage collector runs frequently while reclaiming little space it may make sense to increase `gc_interval`, thus reducing the number of garbage collections.

gc_interval_dict Access mode : read/write

Type : Integer

Default : 960

Description : Specify after how many new dictionary entries the dictionary garbage collector is invoked.

goal_expansion Access mode : read/write

Type : Atomic constant `on` or `off`

Default : `on`

Description : Specifies whether goal expansion is done by the compiler. This includes goal macros and other compiler inlining. Can be disabled for debugging purposes.

hostarch Access mode : read-only

Type : String

Description : String identifying the host processor and operating system. It is also the name of the machine-dependent subdirectories in the ECL^{i}PS^e installation.

hostid Access mode : read-only

Type : Integer or string

Description : The unique identification of the host hardware the system is running on.

hostname Access mode : read-only

Type : String

Description : The name of the current host machine.

ignore_eof Access mode : read/write

Type : Atomic constant `on` or `off`

Default : Depends on host architecture

Description : Specifies whether ECL^{i}PS^e is exited when end-of-file is typed to the toplevel prompt or whether this is ignored. If ignored, ECL^{i}PS^e can be exited by calling **halt/0**.

installation_directory Access mode : read-only

Type : String

Description : The name of the toplevel directory of the running ECL^{i}PS^e installation. All ECL^{i}PS^e library, documentation and other directories are below this one.

last_errno **Access mode :** read-only

Type : Integer

Description : The error code that the most recent failed operating system call returned.

library_path **Access mode :** read/write

Type : List of strings

Default : the contents of the user's ECLIPSELIBRARYPATH environment variable, followed by the system library directories

Description : Specifies the list of pathnames used by the system to search for library files. The library path is used by **lib/1,2**, for autoloading, and to expand library/1 structures in pathnames.

loaded_library **Access mode :** read/write

Type : Atom

Description : Returns the names of the currently loaded libraries. This flag may contain multiple values and will return them on backtracking.

macro_expansion **Access mode :** read/write

Type : Atomic constant `on` or `off`

Default : `on`

Description : Specifies whether macro expansion used by the source transformation facility is enabled or disabled. Should be disabled only for debugging purposes.

max_global_trail **Access mode :** read-only

Type : Integer

Description : The total memory available to the global and the trail stack in bytes.

max_local_control **Access mode :** read-only

Type : Integer

Description : The total memory available to the local and control stack in bytes.

max_predicate_arity **Access mode :** read-only

Type : Integer

Default : 255

Description : Returns the maximum number of arguments allowed for an ECL^{i}PS^e predicate.

object_suffix **Access mode :** read-only

Type : String

Description : Returns the suffix of the external object files that can be loaded using **load/1**. It is usually "so" for systems that support shared libraries and "o" for the others.

occur_check **Access mode :** read/write

Type : Atomic constant `on` or `off`

Default : `off`

Description : If the flag is on, occur check is performed on uncompiled unifications and the compiler generates code for unifications that will perform the occur check when necessary.

output_mode **Access mode :** read/write

Type : String

Default : "QPm"

Description : The value is a control string that is recognised by the **%w** format of **printf/3**. This format is used to output results on the toplevel loop and to print debugger trace lines.

pid **Access mode :** read-only

Type : Integer

Description : Returns the process identifier of the current ECL^{i}PS^e.

ppid **Access mode :** read-only

Type : Integer

Description : Returns the process identifier of the current ECL^{i}PS^e's parent process.

prefer_rationals **Access mode :** read/write

Type : Atomic constant `on` or `off`

Default : `off`

Description : Specifies if the result of a /-division of two integers gives a rational or a floating point result. Similar for the result of raising an integer to a negative integral power.

print_depth **Access mode :** read/write

Type : Integer

Default : 20

Description : Specifies the print depth bound for printing compound terms. It can also be set by the '<' option of the debugger. It is not taken into account by **writeq/1, 2**, other I/O built-ins obey this flag.

prolog_suffix **Access mode :** read/write

Type : List of strings

Default : `["", ".sd", ".pl"]`

Description : Specifies the Prolog source file suffix(es) used when compiling files or loading libraries. The system tries to find the file by appending the given suffixes in the order provided.

statistics **Access mode :** read/write

Type : Atomic constant `off`, `some`, `all` or `mode`

Default : `off`

Description : Used to control the operation of the profiling facility. `off` disables profiling, `all` enables counting of all traceable procedure ports, `some` restricts this to the ones that have been selected with **set_flag/3** or **statistics/2**. `mode` is like `all` but it also records the modes of procedure calls.

syntax_option Access mode : read/write

Type : Atom

Description : Returns the names of the currently enabled syntax options. This flag may contain multiple values and will return them on backtracking. A syntax option is reset using set_flag(syntax_option, not(Flag)). Most compatibility packages affect these flags as well. The following options are available:

- `based_bignums` - Allow base notation even for integers longer than the wordsize (i.e. they are always positive).
- `blanks_in_nil` - allow blanks between the brackets in `[]`.
- `limit_arg_precedence` - do not allow terms with a precedence higher than 999 as structure arguments, unless parenthesised.
- `nested_comments` - allow bracketed comments to be nested.
- `nl_in_quotes` - allow newlines to occur inside quotes.
- `no_blanks` - do not allow blanks between functor an opening parenthesis
- `no_other_quotes` - do not allow string quotes inside atom quotes and vice versa.
- `$VAR` - terms of the form '$VAR'(N) are printed in a special way by all the predicates that obey operator declarations (i.e. write, writeq, print and partly printf). '$VAR'(0) is printed as A, '$VAR'(25) as Z, '$VAR'(26) as A1 and so on. When the argument is an atom or a string, just this argument is printed.

toplevel_module Access mode : read/write

Type : Atom

Description : The name of the current top-level module. This is the caller module for all queries entered in the top-level loop. By default, this is also shown in the top-level prompt.

unix_time Access mode : read-only

Type : Integer

Description : Return the time as given by the UNIX time(3) function, ie. seconds since 00:00:00 GMT, Jan 1 1970.

variable_names Access mode : read/write

Type : Atomic constant `on`, `off` or `check_singletons`

Default : **check_singletons**

Description : Controls the ability to retain the source name of variables. This flag affects the reading process only, i.e. when a variable is read (or compiled) with the flag set to **on**, it will keep its name even when the flag is switched off later.

check_singletons is like **on** but additionally the compiler emits warnings about source variables which occur only once in a clause and whose name does not start with an underscore. The source variable names are being created during the term input if this flag is not **off**, and then they are kept independently of the value of this flag.

version Access mode : read-only

Type : Atom

Description : Returns the current version number of ECL^{i}PS^e.

worker Access mode : read-only

Type : Integer

Description : In a parallel session it returns a positive number, identifying the worker on which the flag inquiry was executed. In a sequential session 0 is returned.

Appendix G

Restrictions and Limits

The ECL^{i}PS^e implementation tries to impose as few limits as possible. The existing limits are:

1. The maximum arity of a predicate in ECL^{i}PS^e is 255 (this value can be queried using get_flag(max_predicate_arity,X)). Note however that the arity of compound terms is unlimited.

2. The maximum arity of external predicates in the current implementation of ECL^{i}PS^e is 16.

3. The stack and heap sizes have virtual memory limits which can be changed using the -g, -l, -s and -p command line options or the ec_set_option function in case of an embedded ECL^{i}PS^e.

4. When the occur check is disabled and cyclical structures are created, e.g. in such situations as the unification of *X* and *g(X)* in

   ```
   X = g(X)
   ```

   which will result in a cyclic structure:

   ```
    X = g(g(g(g(g(...
   ```

   ECL^{i}PS^e is not able to copy such terms to the heap and so it is the programmer's responsibility to ensure that the cyclical terms are avoided in

   - Values of arrays and global variables.
   - Asserted and recorded terms.

Index

Bibliography

[1] D.L. Bowen. DEC-10 Prolog User's Manual. D.A.I. occasional paper 27, University of Edinburgh, December 1981.

[2] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[3] Mehmet Dincbas and Jean-Pierre Le Pape. Metacontrol of Logic Programs in METALOG. In ICOT, editor, *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*, pages 361–370, 1984.

[4] *ECLiPSe 3.4 Extensions User Manual*, 1994.

[5] H. A. Grant (ITC), A. Cunningham (ITC), and P. Kay (ECRC). KEGI User Manual. — SEP/UM/024, ITC, February 1990.

[6] Micha Meier. Event handling in Prolog. In *Proceedings of the North American Conference on Logic Programming*, Cleveland, October 1989.

[7] Micha Meier. Compilation of compound terms in Prolog. In *Proceedings of the NACLP'90*, Austin, October 1990.

[8] Micha Meier, Abderrahmane Aggoun, David Chan, Pierre Dufresne, Reinhard Enders, Dominique Henry de Villeneuve, Alexander Herold, Philip Kay, Bruno Perez, Emmanuel van Rossum, and Joachim Schimpf. SEPIA - an extendible Prolog system. In *Proceedings of the 11th World Computer Congress IFIP'89*, pages 1127–1132, San Francisco, August 1989.

[9] Micha Meier, Philip Kay, Emmanual van Rossum, and Hugh Grant. Sepia programming environment. In *Proceedings of the NACLP'89 Workshop on Logic Programming Environments: The Next Generation*, pages 82–86, Cleveland, October 1989.

[10] Micha Meier and Joachim Schimpf. An architecture for prolog extensions. In *Proceedings of the 3rd International Workshop on Extensions of Logic Programming*, pages 319–338, Bologna, 1992.

[11] Micha Meier, Joachim Schimpf, and Emmanual van Rossum. A guide to SEPIA customization and advanced programming. Technical Report TR-LP-50, ECRC, June 1990.

[12] Stefano Novello and Joachim Schimpf. *ECLiPSe Embedding and Interfacing Manual*, 1999.

[13] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[14] D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI, October 1983.