ECL^{i}PS^e

User Manual

Release 7.1

Abderrahamane Aggoun (ECRC)
David Chan (ECRC)
Pierre Dufresne (ECRC)
Eamon Falvey (ICL-ITC)
Hugh Grant (ICL-ITC)
Warwick Harvey (IC-Parc and CrossCore)
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, IC-Parc, CrossCore and Cisco)
Kish Shen (IC-Parc, CrossCore and Cisco)
Periklis Andreas Tsahageas (ECRC)
Dominique Henry de Villeneuve (ECRC)

September 3, 2022

Trademarks

Contents

Chapter 1

Introduction

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

ECL^{i}PS^e (ECL^{i}PS^e Constraint Logic Programming System) is an open-source software system whose aim is to serve as a platform for integrating various Logic Programming extensions. It is used in particular for the cost-effective development and deployment of constraint programming applications, e.g. in the areas of planning, scheduling, resource allocation, timetabling, transport etc. It is also ideal for teaching most aspects of combinatorial problem solving, e.g. problem modelling, constraint programming, mathematical programming, and search techniques. It contains several constraint solver libraries, a high-level modelling and control language, interfaces to third-party solvers, an integrated development environment and interfaces for embedding into host environments.
The ECL^{i}PS^e programming language has been developed from the Prolog language ([3]), more specifically the Edinburgh family of Prologs and more recently the ISO Prolog standard ([1]). ECL^{i}PS^e retains backward compatibility by supporting several language dialects.
In terms of implementation technology, ECL^{i}PS^e is built around an incremental compiler which compiles ECL^{i}PS^e source into WAM-like code [14], and a runtime system comprising an emulator of this abstract code, automatic memory management, I/O system and built-in predicates.

1.2 Overview

The ECL^{i}PS^e logic programming system was originally an integration of ECRC's SEPIA, Mega-Log and (parts of the) CHIP systems. It was then further developed into a Constraint Logic Programming system with a focus on hybrid problem solving and solver integration. The documentation is organised as follows:

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

The Constraint 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.

The Reference Manual contains detailed descriptions of all the Built-in predicates and the libraries. This information is also available from the development system's help/1 command

and the tkeclipse library browser.

The Visualisation Manual describes the facilities for the visualisation of constraint propagation and search.

All the documentation can be accessed using an html browser (refer to the eclipse installation directory under doc/index.html).

1.3 Further Information

ECL^{i}PS^e was initially developed at the European Computer-Industry Research Centre (ECRC) in Munich, and then at IC-Parc, Imperial College in London until the end of 2005. It is now an open-source project, with the support of Cisco Systems. Up-to-date information can be obtained from the ECL^{i}PS^e web site

`http://www.eclipseclp.org`

or from the Sourceforge site under the project name **eclipse-clp**

`http://www.sourceforge.net/projects/eclipse-clp`

which also hosts the main source repository. There you can also subscribe to the ECL^{i}PS^e user group mailing list or access its archives.

`eclipse-clp-users@lists.sf.net`

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 submit problem reports via the web site

`http://eclipseclp.org/bugs.html`

or by e-mail to

`eclipse-clp-bugs@lists.sf.net`

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 predicate is called.

++X This denotes a ground argument. Such an argument can be complex, but must be fully instantiated, i.e., not contain any variables.

–X This denotes an output argument. Such an argument is allowed to be uninstantiated at call time. When this mode is used in the description of a built-in or library predicate, it is only *descriptive*. This means that the predicate can be called with an instantated argument, but it will behave as if were called with an uninstantiated variable which is then unified with the actual argument after returning from the call (e.g. `atom_length(abc,3)` behaves the same as `atom_length(abc,L),L=3`). If this mode is used in a **mode/1** declaration, it is *prescriptive*, i.e. it is taken as a promise that the predicate will always be called with an uninstantiated variable, and that the compiler is allowed to make corresponding optimizations. Violating this promise will lead to unexpected failures.

?X This denotes an input or an output argument. Such an argument may be either instantiated or not when the predicate 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 by giving its name and 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. It can be written as a conventional identifier (beginning with a lower-case letter), or a character sequnce enclosed in single quotes.

Atomic An atom, string or a number. A term 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).

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

Clause See *program clause* or *goal clause.*

Compound Term Compound terms are of the form

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

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

Constant An *atom*, a *number* or a *string.*

Determinism The determinism specification of a built-in or library predicate says how many solutions the predicate can have, and whether it can fail. The six determinism groups are defined as follows:

```
            |   Maximum number of solutions
Can fail?   |   0               1               > 1
------------+-----------------------------------------
no          |   erroneous       det             multi
yes         |   failure         semidet         nondet
```

This classification is borrowed from the Mercury programming language, but in ECL^{i}PS^e only used for the purpose of documentation. Note that the determinism of a predicate usually depends on its calling mode.

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 Procedure 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.*

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 *head*.

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 clause 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 | []]]]
```

Mode A predicate mode is a particular instantiation pattern of its arguments at call time. Such a pattern is usually written as a predicate template, e.g.,

```
p(+,-)
```

where the symbols `+`, `++`, `-` and `?` represent instantiated, ground, uninstantiated and unknown arguments respectively.

Name/Arity The notation *Name/Arity* is used to specify a functor by giving its name and arity.

Number A number literal denotes a number, more or less like in all programming languages.

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 *Name/Arity*. Some built-ins allow the arity to be omitted and to specify the name only: this stands for all (visible) predicates with that name and any arity.

Program Clause A program clause (or simply *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 a *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.
```

The first of these two forms is often called a *directive*.

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 by a compound term (or atom) with this functor.

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

```
name_1/arity_1, name_2/arity_2, ..., name_k/arity_k.
```

The SpecList notation is used in many built-ins, for example, to specify a list of procedures in the **export/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 predefined stream identifiers (`input`, `output`, `error`, `warning_output`, `log_output`, `null`) or a user defined stream name (defined using **set_stream/2** or **open/3**).

String A string is similar to those found in all other programming languages. A string is enclosed in double quotes.

Structure 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* or a *compound term.*

Variable A variable is more similar to a mathematical variable than to a variable in some imperative language. It can be free, or instantiated to a term, but once instantiated it becomes indistinguishable from the term to which it was instantiated: in particular, it cannot become free again (except upon backtracking through the point of instantiation). The name of a variable is written in the form of an identifier that begins with an upper-case letter or with an underscore. A single underscore represents an *anonymous variable* that has only one occurrence (i.e., another occurrence of this name represents another variable).

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 How do I install the ECL^{i}PS^e system?

Please see the installation notes that came with ECL^{i}PS^e. For Unix/Linux systems, these are in the file `README_UNIX`. For Windows, they are in the file `README_WIN.TXT`.
Please note that choices made at installation time can affect which options are available in the installed system.

3.2 How do I run my ECL^{i}PS^e programs?

There are two ways of running ECL^{i}PS^e programs. The first is using `tkeclipse`, which provides an interactive graphical user interface to the ECL^{i}PS^e compiler and system. The second is using `eclipse`, which provides a more traditional command-line interface. We recommend you use TkECL^{i}PS^e unless you have some reason to prefer a command-line interface.

3.3 How do I use `TkECL`^{i}`PS`^{e}?

3.3.1 Getting started

To start TkECL^{i}PS^e, either type the command `tkeclipse` at an operating system command-line prompt, or select TkECL^{i}PS^e from the program menu on Windows. This will bring up the TkECL^{i}PS^e top-level, which is shown in Figure 3.1.
Note that help on TkECL^{i}PS^e and its component tools is available from the `Help` menu in the top-level window. If you need more information than can be found in this manual, try looking in the `Help` menu.

3.4 How do I write an ECL^{i}PS^e program?

You must use an editor to write your programs. ECL^{i}PS^e does not come with an editor, but any editor that can save plain text files can be used. Save your program as a plain text file, and you can then compile the program into ECL^{i}PS^e and run it.
With TkECL^{i}PS^e, you can specify the editor you want to use, and this editor will be started by TkECL^{i}PS^e, e.g., when you select a file in the 'Edit' option under the File menu. The

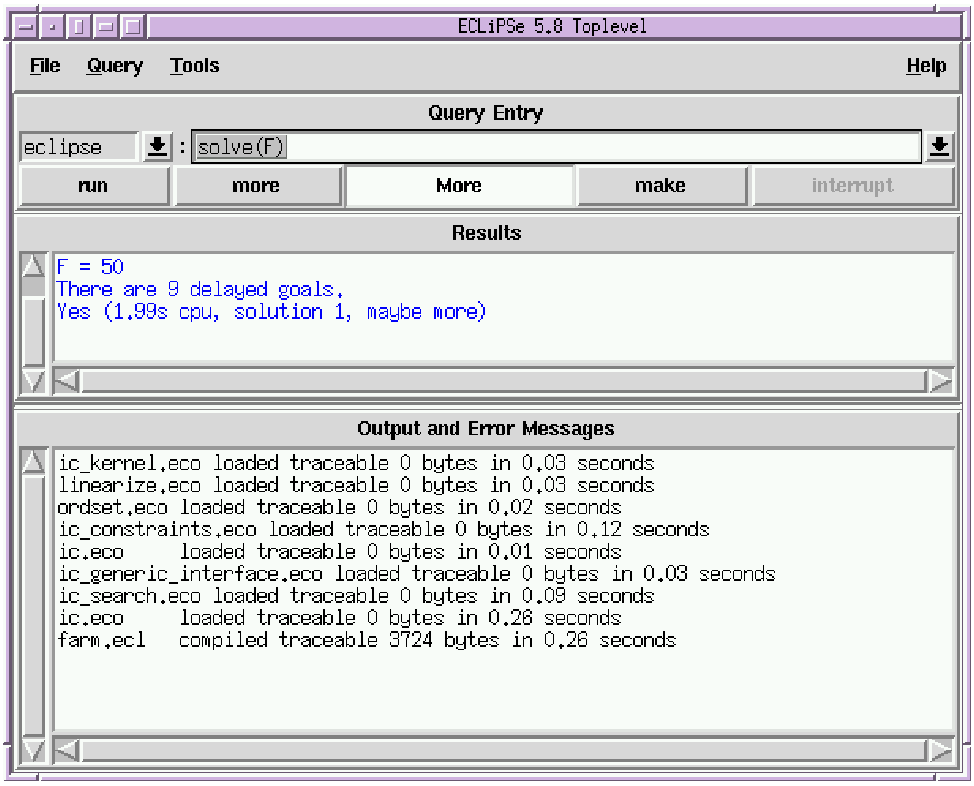


Figure 3.1: TkECL^{i}PS^e top-level

default values are the value of the VISUAL environment variable under Unix, or Wordpad under Windows. This can be changed with the Preference Editor under the Tools menu.

3.4.1 Compiling a program

From the `File` menu, select the `Compile ...` option. This will bring up a file selection dialog. Select the file you wish to compile, and click on the `Open` button. This will compile the file and any others it depends on. Messages indicating which files have been compiled and describing any errors encountered will be displayed in the bottom portion of the TkECL^{i}PS^e window (`Output and Error Messages`).
If a file has been modified since it was compiled, it may be recompiled by clicking on the `make` button. This recompiles any files which have become out-of-date.
For more information on program compilation and the compiler, please see chapter 6.

3.4.2 Executing a query

To execute a query, first enter it into the `Query Entry` text field. You will also have to specify which module the query should be run from, by selecting the appropriate entry from the drop-down list to the left of the `Query Entry` field. Normally, the default selection of `eclipse` will be fine; this will allow access to all ECL^{i}PS^e built-ins and all predicates that have not explicitly been compiled into a different module. Selecting another module for the query is only needed if you wish to call a predicate which is not visible from the `eclipse` module, in which case you

must select that module. (For more information about the module system, please see chapter 8.)
To actually execute the query, either hit the `Enter` key while editing the query, or click on the `run` button. TkECL^{i}PS^e maintains a history of commands entered during the session, and these may be recalled either by using the drop-down list to the right of the `Query Entry` field, or by using the up and down arrow keys while editing the `Query Entry` field.
If ECL^{i}PS^e cannot find a solution to the query, it will print `No` in the `Results` section of the TkECL^{i}PS^e window. If it finds a solution and knows there are no more, it will print it in the `Results` section, and then print `Yes`. If it finds a solution and there may be more, it will print the solution found as before, print `More`, and enable the `more` button. Clicking on the `more` button tells ECL^{i}PS^e to try to find another solution. In all cases it also prints the total time taken to execute the query.
Note that a query can be interrupted during execution by clicking on the `interrupt` button.

3.4.3 Editing a file

If you wish to edit a file (e.g., a program source file), then you may do so by selecting the `Edit ...` option from the `File` menu. This will bring up a file selection dialog. Select the file you wish to edit, and click on the `Open` button.
When you have finished editing the file, save it. After you've saved it, if you wish to update the version compiled into ECL^{i}PS^e (assuming it had been compiled previously), simply click on the `make` button.
You can change which program is used to edit your file by using the TkECL^{i}PS^e Preference Editor, available from the `Tools` menu.

3.4.4 Debugging a program

To help diagnose problems in ECL^{i}PS^e programs, TkECL^{i}PS^e provides the tracer. This can be invoked by selecting the `Tracer` option from the `Tools` menu. The next time a goal is executed, the tracer window will become active, allowing you to step through the program's execution and examine the program's state as it executes.
The tracer displays the current call stack, the program source, and a trace log. By using the left mouse button in the `Call Stack` region of the tracer window, you can bring up a menu of additional operations you can perform on that goal, such as inspecting it, or setting a spy point on the predicate in question. Source breakpoints can be set by marking the corresponding line in the tracer's source display. Selecting `Configure filter ...` from the `Options` menu of the tracer will launch the conditional filter. This filter allows you to specify conditions on which the tracer should stop at a debug port. This can be very useful for skipping over unwanted debug ports.
For more information on using the tracer, please see the online help, available by selecting `Tracer Help` from the `Help` menu.
Other TkECL^{i}PS^e tools which are useful while using the tracer are:

- the predicate browser (available by selecting the `Predicate Browser` option from the `Tools` menu), which is useful for setting or removing spy points on predicates, or for setting the `start_tracing` flag which activates the tracer when a particular predicate is called for the first time; and

- the term inspector (available by double left clicking on a term from the stack window, or by selecting the `Inspector` option from the `Tools` menu), which is useful for examining and browse the arguments of a term in detail.
- the delayed goals browser (available by selecting the `Delayed Goals` option from the `Tools` menu), which allows you to inspect the current list of delayed goals.
- the display matrix (available either from calls in user's code, or by interactively selecting terms to be observed from the inspector, tracer or delay goals tools), which allows you to monitor any changes to a term and its arguments.

More information about debugging in ECL^{i}PS^e may be found in chapter 15.

3.4.5 Getting help

More detailed help than is provided here can be obtained online. Simply select the entry from the `Help` menu on TkECL^{i}PS^e's top-level window which corresponds to the topic or tool you are interested in.

3.4.6 Other tools

TkECL^{i}PS^e comes with a number of useful tools. Some have been mentioned above, but here is a more complete list. Note that we only provide brief descriptions here; for more details, please see the online help for the tool in question.

Compile scratch-pad

This tool allows you to enter small amounts of program code and have it compiled. This is useful for quick experimentation, but not for larger examples or programs you wish to keep, since the source code is lost when the session is exited.

Source File Manager

This tool allows you to keep track of and manage which source files have been compiled in the current ECL^{i}PS^e session. You can select files to edit them, or compile them individually, as well as adding new files.

Predicate Browser

This tool allows you to browse through the modules and predicates which have been compiled in the current session. It also lets you alter some properties of compiled predicates.

Source Viewer

This tool attempts to display the source code for predicates selected in other tools.

Delayed Goals

This tool displays the current delayed goals, as well as allowing a spy point to be placed on the predicate and the source code viewed.

Tracer

As discussed in section 3.4.4, the tracer is useful for debugging programs. See also chapter 15.

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 buttons as an alternative way to navigate the tree. Note that while the inspector window is open, interaction with other TkECL^{i}PS^e windows is disallowed. This prevents the term from changing while being inspected. To continue TkECL^{i}PS^e, the inspector window must be closed.

Global Settings

This tool allows the setting of some global flags governing the way ECL^{i}PS^e behaves. See also the documentation for the **set_flag/2** and **get_flag/2** predicates.

Statistics

This tool displays some statistics about memory and CPU usage of the ECL^{i}PS^e system, updated at regular intervals. See also the documentation for the **statistics/0** and **statistics/2** predicates.

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).

Library Help

This tool allows you to browse the online help for the ECL^{i}PS^e libraries. On the left is a tree display of the libraries available and the predicates they provide.

- Double clicking on a node in this tree either expands it or collapses it again.
- Clicking on an entry displays help for that entry to the right.
- Double clicking on a word in the right-hand pane searches for help entries containing that string.

You can also enter a search string or a predicate specification manually in the text entry box at the top right. If there is only one match, detailed help for that predicate is displayed. If there are multiple matches, only very brief help is displayed for each; to get detailed help, try specifying the module and/or the arity of the predicate in the text field.

3.4.7 Preference Editor

This tool allows you to edit and set various user preferences. This include parameters for how TkECL^{i}PS^e will start up, e.g., the amount of memory it will be able to use, and an initial query to execute; and parameters which affects the appearance of TkECL^{i}PS^e, such as the fonts TkECL^{i}PS^e uses.

3.5 How do I use `eclipse`?

3.5.1 Getting started

To start ECL^{i}PS^e, type the command `eclipse` at an operating system command-line prompt. This will display something like this:

```
% eclipse
ECLiPSe Constraint Logic Programming System [kernel]
Kernel and basic libraries copyright Cisco Systems, Inc.
and subject to the Cisco-style Mozilla Public Licence 1.1
(see legal/cmpl.txt or eclipseclp.org/licence)
Source available at www.sourceforge.org/projects/eclipse-clp
GMP library copyright Free Software Foundation, see legal/lgpl.txt
For other libraries see their individual copyright notices
Version X.Y #Z, DAY MONTH DD HH:MM YYYY
[eclipse 1]:
```

The list in square brackets on the first line specifies the configuration of the running system, i.e., the language extensions that are present. The copyright and version information 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 general help and **help/1** gives help about specific built-in predicates.

3.5.2 Interacting with the top level loop

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 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 ECL^{i}PS^e goals, e.g., to pose queries, to enter an ECL^{i}PS^e 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.
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 either the **halt/0** or the **exit/1** predicates.

3.5.3 Compiling a program

The square brackets `[...]` or the **compile/1** predicate are used to compile ECL^{i}PS^e 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` or `myfile.ecl` and compiles it. 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.
If a file has been modified since it was compiled, it may be recompiled by invoking the **make/0** predicate. This recompiles any files which have become out-of-date.
For more information on program compilation and the compiler, please see chapter 6.

3.5.4 Entering a program from the terminal

Programs can be entered directly from the terminal, as well as being read from files. To do this, simply compile the special file `user`. That is, `[user].` or `compile(user).` at a top level prompt. The system then displays the compiler prompt (which is a blank by default) and waits for a sequence of 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 fullstop by accident simply type it in on the following line, and then proceed to type in the program clauses, each followed by a fullstop 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.
For example:

```
[eclipse 1]: [user].
source_processor.eco loaded in 0.01 seconds
...
ecl_compiler.eco loaded in 0.23 seconds
 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
 tty        compiled 420 bytes in 0.01 seconds

Yes (0.24s cpu)
[eclipse 2]:
```

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

3.5.5 Executing a query

Once a set of clauses has been compiled, it may be queried in the usual Prolog manner. 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. If potentially there is another solution, the top level 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 3]: father(X, Y).

X = abraham
Y = isaac
Yes (0.00s cpu, solution 1, maybe more) ? ;            (user types ';')

X = isaac
Y = jacob
Yes (0.00s cpu, solution 2)
[eclipse 4]:
```

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

3.5.6 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 : ? h       (user types 'h')
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, and `e` is an emergency exit of the whole ECL^{i}PS^e session. If the debugger is running, an additional option `d` is displayed: it switches the debugger to creep mode.
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 shell command `fg` will return to ECL^{i}PS^e. Note that this feature may not be available on all systems.

3.5.7 Debugging a program

Please see the chapters on debugging in the tutorial and user manuals for more details. The tutorial chapter covers the TkECL^{i}PS^e debugging in a tutorial style tour, and the user manual chapter covers debugging in general and the command-line debugger in particular.

3.5.8 The history mechanism

The ECL^{i}PS^e toplevel loop provides a simple history mechanism which allows the examination and repetition of previous queries. The history list is printed with the command `h`. A previous query is invoked by typing either 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.

3.5.9 Getting help

Detailed documentation about all the predicates in the ECL^{i}PS^e libraries can be obtained online through 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. For example,

```
:- 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).
```

3.6 How do I make things happen at compile time?

A file being compiled may contain queries. These are goals preceded by the symbol ":-". As soon as a query is encountered in the compilation of a file, the ECL^{i}PS^e system will try to satisfy it. Thus by inserting goals in this fashion, things can be made to happen at compile time.
In particular, a file can contain a directive to the system to compile another file, and so large programs can be split between files, while still only requiring a single simple command to compile them. 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, for example, 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 ECL^{i}PS^e 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. (Note that if your program is large enough to warrant breaking into multiple files (let alone multiple directories), it is probably worth turning the constituent components into modules — see chapter 8.)

3.7 How do I use ECL^{i}PS^e libraries in my programs?

A number of files containing library predicates are supplied with the ECL^{i}PS^e system. These predicates provide utility functions for general use. They are usually installed in an 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 goals:

```
lib(lists)
use_module(library(lists))
```

This will load the library file unless it has been already loaded. In particular, a program can ensure that a given library is loaded when it is compiled, by including an appropriate directive in the source, e.g., `:- lib(lists).`
Library files are 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.

3.8 How do I make my programs run faster?

By default, ECL^{i}PS^e compiles programs as *traceable*, which means that they can be traced using the built-in debugger. To obtain maximum efficiency, the directive **nodbgcomp/0** 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]:
```

Section 6.7 contains more detailed discussion on other techniques which can be used to optimise your programs.

3.9 Other tips

3.9.1 Initialization at start-up

If you wish to have ECL^{i}PS^e do or execute things at startup time, you can achieve this in TkECL^{i}PS^e by setting the initial query call in the Preference editor; and in the command-line `eclipse` by putting via a `.eclipserc` file.
For `eclipse`, before displaying the initial prompt, the system checks whether there is a file called `.eclipserc` in the current directory and if not, in the user's home directory. If such a file is found, ECL^{i}PS^e compiles it first. Thus it is possible to put various initialization 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 initialization file can be specified in the environment variable `ECLIPSEINIT`. If `ECLIPSEINIT` is set to an empty string, no initialization is done. If the system is started with a -e option, then the `.eclipserc` file is ignored.
For TkECL^{i}PS^e, the system will make the initial query call as set in the Preference Editor before giving control to the user. This call can be set to compile an initialization file. This can be the `.eclipserc` file, or some other file if the user wants to initialize the system differently in TkECL^{i}PS^e.

3.9.2 Recommended file names

It is 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 file name, so that it does not have to be specified; if the literal file name can not be found, the system tries appending each of the valid suffixes in turn and tries to find the resulting file name. 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]).
```

Chapter 4

The TkECL^{i}PS^e Development Tools

TkECL^{i}PS^e 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 to assist the user in interacting 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.

TkECL^{i}PS^e is implemented in the Tcl/Tk scripting language/graphical toolkit [12], using the new ECL^{i}PS^e Tcl/Tk interface [11]. The development tools are designed to be independent of the top-level, so the users can develop their own applications with a graphical front end written in Tcl/Tk, replacing the TkECL^{i}PS^e top-level, but still using the development tools.
Chapter 3 gave an introduction to using TkECL^{i}PS^e from a user's point of view. This chapter focuses on how to use the tools from a programmer's point of view (i.e., how to include them in a program). In particular it discusses in detail the *display matrix* tool, which can be invoked in user's ECL^{i}PS^e code; and also how to use the development tools in the user's own applications.

4.1 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 predicate which invokes the display matrix is considered a no-op in the tty-based ECL^{i}PS^e,[2] and so the same code can be run without modification from either `eclipse` or `tkeclipse`, though the matrix display is only presented to the user in the latter.
To invoke this tool use either **make_display_matrix/2** or **make_display_matrix/5**. Adding a call to one of these predicates should be the only change you need to make to your 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 tool is similar to the variable display of *Grace*. The main differences are: it can display all attributes, not just the finite domain attribute; the attributes can only be observed, not changed; and the labelling strategy cannot be changed.

[2]Unless it is attached to the remote development tools, in which case the display matrix is invoked.

Figure 4.1: Display Matrix Tool for 4-Queens (Initial)

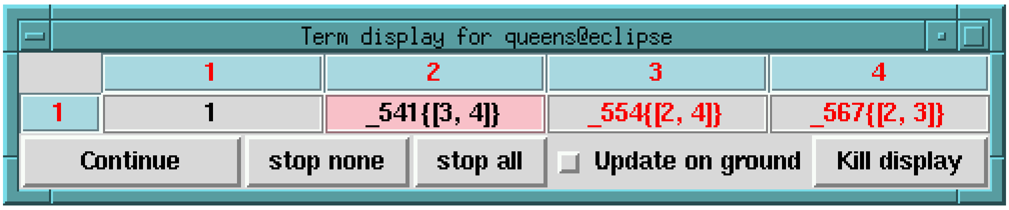


Figure 4.2: Display Matrix Tool for 4-Queens (During execution)

```
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 4.1 and 4.2 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. Spy 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 allows the user to inspect the term using the inspector.
Note that 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 TkECL^{i}PS^e. 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 *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 retracted during backtracking, the old value is sent to the display matrix. The other arguments in this predicate are 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 are 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.

4.1.1 Invoking display matrix tool interactively

Display matrices can be created interactively when a program is executing, if the program is being debugged with the tracer tool. The user can select terms that are to be observed by a display matrix while at a debug port. This can be done from the inspector, the tracer, and the delay goal tools. See the online help files (available from the help menu of TkECL^{i}PS^e) for more details.

4.2 Using the development tools in applications

The user can develop their own ECL^{i}PS^e applications using the development tools independently of the TkECL^{i}PS^e toplevel. There are two ways to do this, depending on if the user is also using the embedding Tcl/Tk interface (see the Embedding and Interfacing Manual) to provide a graphical front end:

- The user is using the embedding Tcl/Tk interface, and is thus developing a graphical front end in Tk. In this case the user can use the development tools via the embedding interface. This is described in section 4.2.1.
- The user is not using the embedding Tcl/Tk interface. In this case the user can use the development tools remotely, by using the remote_tools library. This is described in section 4.2.2.

4.2.1 Using the Development tools in the Tcl/Tk Embedding Interface

The development tool suite was designed to be independent of the TkECL^{i}PS^e top-level so that they can be used in a user's application. In effect, the user can replace the TkECL^{i}PS^e top-level with their own alternative top-level. Two simple examples in which this is done are 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 initialized 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 initialization 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 TkECL^{i}PS^e. 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.

4.2.2 Using the Remote Development Tools

The user can also use the development tools via the remote_tools library. In this case, the development tools are run as a separate program from the ECL^{i}PS^e session, and are attached to it via the Tcl/Tk remote interface (see the Embedding and Interfacing Manual). This allows any ECL^{i}PS^e session to use the development tools, as long as there is the capability for graphical display.

The main purpose for the remote_tools library is to allow the user to use the development tools in situations where (s)he cannot use the Tcl/Tk embedding interface, e.g., if ECL^{i}PS^e is already embedded into another programming language, or if the user has to use the tty interface for ECL^{i}PS^e.

Once attached to an ECL^{i}PS^e session, the remote development tools have their own window as shown in Figure 4.3. The Tools menu is the same as in TkECL^{i}PS^e, providing access to the same suite of development tools. The main body of the window consists of one button and a status indicator. The indicator shows whether the tools can be used or not (the tools cannot be used when the ECL^{i}PS^e is active), and the button is used to pass control explicitly to ECL^{i}PS^e.

The ECL^{i}PS^e session and the development tools are two separate processes (and in fact they can be running on different machines) that are connected to each other via the remote Tcl/Tk interface. The interactions of the two processes are synchronised in that there is a thread-like

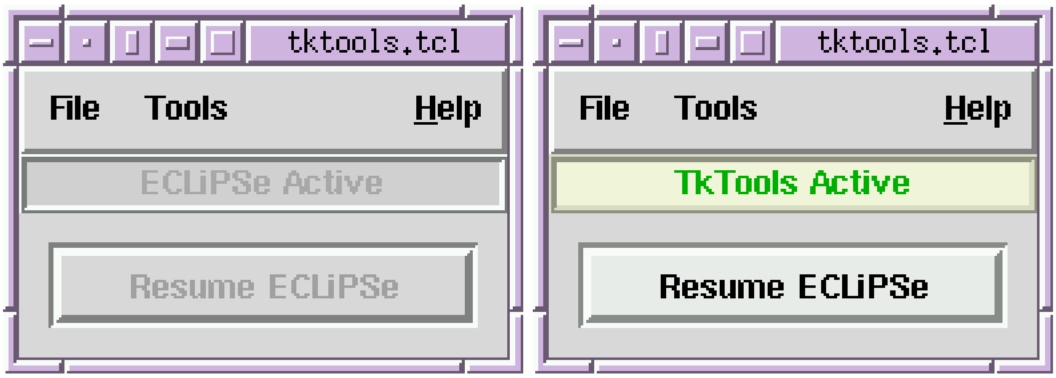


Figure 4.3: Remote Development Tools Toplevel (left: ECL^{i}PS^e active; right: remote tools active)

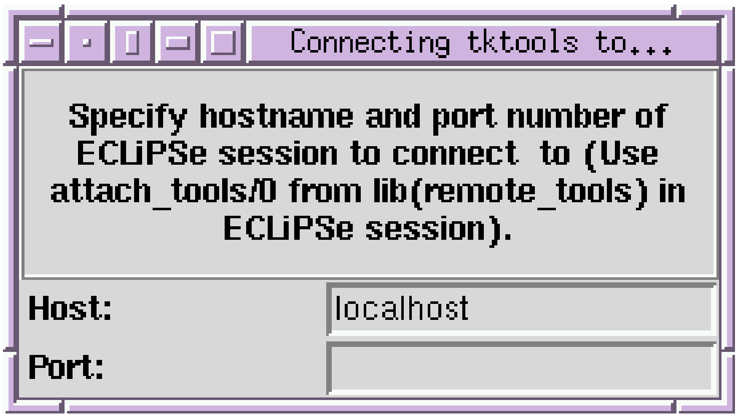
flow of control between them: only one process can be 'active' at any time. The interaction is similar to the standard interaction between a debugger and the program being debugged – debugging commands can only be issued while the execution of the program is suspended. In the same way, the user can only interact with the remote tools window when execution in the ECL^{i}PS^e session is suspended. The toplevel window of the remote tools has an indicator showing which side has control (see Figure 4.3). To allow ECL^{i}PS^e to resume execution, control is transferred back from the remote tools to ECL^{i}PS^e. This can either be done automatically from the tools (e.g., when one of the debug buttons is pressed in the tracer tool), or control can be transferred explicitly back to ECL^{i}PS^e via the "Resume ECLiPSe" button on the remote tools window.

Starting Remote Tools

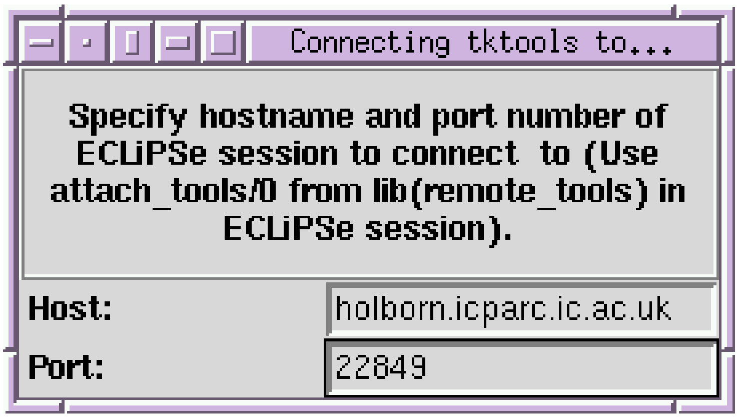
To use the remote tools, the user must first load the right library with `lib(remote_tools)`. After loading the library, the user can start the remote tools by starting the development tools as a separate program and then manually attaching the program to the ECL^{i}PS^e session. This allows the development tools to be run on a different machine from the ECL^{i}PS^e session. In this case, the user initiates the attachment in ECL^{i}PS^e with **attach_tools/0**:

```
[eclipse 2]: attach_tools.
Socket created at address holborn.icparc.ic.ac.uk/22849
```

ECL^{i}PS^e prints the host and port address it expects the remote tools to attach to, and execution is now suspended waiting for the remote tools to attach. This is done by running the *tktools* program, which is located with the other ECL^{i}PS^e executables. As stated, this program can be run on a different machine from the ECL^{i}PS^e session, as long as the two are connected via a network such as the internet. A connection window is then displayed as shown:



The same ‘host’ and ‘port’ fields as printed by the ECL^{i}PS^e session should be entered. The default ‘host’ field is ‘localhost’. This will work if the remote tools are ran on the same machine as the ECL^{i}PS^e session. Otherwise the full name of the ‘host’ as given by **attach_tools/0** must be entered:



Typing return in the ‘port’ field will start the attachment, and with success, the remote tools window (see Figure 4.3) will be displayed. The **attach_tools/0** predicate will also return.
The user is not able to immediately interact directly with the remote tools, as the ECL^{i}PS^e session is initially given control. The user can use the ECL^{i}PS^e session normally, with the additional availability of the development tools. For example, the display matrix predicates can be used as in TkECL^{i}PS^e. Also, the tracer tool replaces the previous tracing facilities of the ECL^{i}PS^e session (this would typically be the command-line debugger).
The tools can be triggered by events in the ECL^{i}PS^e session as described above. In order to use the tools in a more interactive way, control should be handed over to the remote tools. This can be done by calling the **tools/0** predicate. When the remote tools have control, the user can now interactively select development tools from the Tools menu.
The remote_tools library provides several predicates to facilitate the use of the remote development tools:

tools Explicitly hands over control to the remote development tools. The tools window can then be used interactively. Execution on the ECL^{i}PS^e session is suspended until the remote tools allows ECL^{i}PS^e to resume, at which point the predicate succeeds. The predicate will abort if the development tools are disconnected from the ECL^{i}PS^e session.

attached(?*ControlStream*) Checks if the remote development tools have been attached to this ECL^{i}PS^e session or not. If attached, the predicate succeeds and unifies *ControlStream* with the stream name of the control stream. If not attached, the predicate fails.

Once attached, the remote development tools should be connected until the user quits the session. Although it is possible to disconnect the tools from the ECL^{i}PS^e session (from the File menu in the development tools window), this is not recommended, as there would not be any debugging facilities available after the disconnection – the original tracer would not be restored.
It is possible to attach the remote development tools to any ECL^{i}PS^e session, including one that is using the embedding Tcl/Tk interface (and indeed, to TkECL^{i}PS^e itself). However, using the tools via the embedding interface is usually the better option if available, because the tools are more tightly coupled to ECL^{i}PS^e in this case. This means that the communications between ECL^{i}PS^e and the tools are more efficient (and hence something like the display matrix would perform more efficiently).

Chapter 5

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.

5.1 Structure Notation

ECL^{i}PS^e abstract 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 parse-time preprocessing).
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{}
book{title:'tom sawyer'}
book{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 the parser, 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
```

For example, to access (i.e., 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), ...
```

If a program is consistently written using the abstract structure notation (i.e., with `{...}` and `of`), then the struct-declaration can be modified (fields added or rearranged) without having to update the code anywhere else.

5.1.1 Updating Structures

To construct an updated structure, i.e., a structure which is similar to an existing structure except that one or more fields have new values, use the **update_struct/4** built-in, which allows you to do that without having to mention all the other field names in the structure.

5.1.2 Arity and Functor of Structures

The arity of a structure can be symbolically written using **of/2** as follows:

```
property(arity) of StructName
```

For example,

```
?- printf("A book has %d fields%n", [property(arity) of book]).
A book has 4 fields
Yes.
```

Similarly, the whole *StructName/Arity* specification can be written as

```
property(functor) of StructName
```

which is used for the portray-declaration in the example below.

5.1.3 Printing Structures

When structures are printed, they are not translated back into the abstract structure 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 customized portray-transformations for them, for instance

```
:- local portray(property(functor) of book, tr_book_out/2, []).
tr_book_out(book{author:A,title:T},
        no_macro_expansion(book{author:A,title:T})).
```

which will cause **book/4** structures to be printed like

```
book{author:twain, title:tom sawyer}
```

while the other two arguments remain hidden.

5.1.4 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{name:john,salary:2000}.
Emp = employee(person(john, _105, _106), 2000)
yes.
```

Note that, as long as the abstract structure notation 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{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.
```

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

5.1.5 Visibility

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

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

in the module.

5.2 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. More background can be found in [13]. 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 auxiliary predicate:

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

This looks very much like a loop in a procedural language. However, due to the relational nature of logic programming, the same **foreach** construct can be used not only to control iteration over an existing list, but also to build a new list during an iteration. For example

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

will print `[-1, -2, -3]`.
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(...).
```

The *IterationSpecs* determine the number of times the loop is executed (i.e., the termination condition), and the way information is passed into the loop, from one iteration to the next, and out of the loop.
IterationSpecs is one (or a combination) of the following:

fromto(*First*, *In*, *Out*, *Last*)
: iterate *Goals* starting with *In=First* until *Out=Last*. *In* and *Out* are local loop variables. For all but the first iteration, the value of *In* is the same as the value of *Out* in the previous iteration.

foreach(*X*, *List*)
: iterate *Goals* with *X* ranging over all elements of *List*. *X* is a local loop variable. 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 loop variable. Cannot be used for constructing a term.

foreacharg(*X*, *Struct*, *Idx*)
: same as before, but *Idx* is set to the argument position of *X* in *Struct*. (In other words, `arg(Idx, Struct, X)` is true.) *X* and *Idx* are local loop variables.

foreachelem(*X*, *Array*)
: like **foreacharg/2**, but iterates over all elements of an array of arbitrary dimension. The order is the natural order, i.e., if
`Array = []([](a, b, c), [](d, e, f)),`
then for successive iterations *X* is bound in turn to *a*, *b*, *c*, *d*, *e* and *f*. *X* is a local loop variable. Cannot be used for constructing a term.

foreachelem(*X*, *Array*, *Idx*)
: same as before, but *Idx* is set to the index position of *X* in *Array*. (In other words, `subscript(Array, Idx, X)` is true.) *X* and *Idx* are local loop variables.

foreachindex(*Idx*, *Array*)
: like **foreachelem/3**, but returns just the index position and not the element.

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

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

multifor(*List*, *MinList*, *MaxList*)
: like **for/3**, but allows iteration over multiple indices (saves writing nested loops). Each element of *List* takes a value between the corresponding elements in *MinList* and *MaxList*. Successive iterations go through the possible combinations of values for *List* in lexicographic order. *List* is a local loop variable. *MinList* and *MaxList* must be either lists of arithmetic expressions evaluating to integers, or arithmetic expressions evaluating to integers (in the latter case they are treated as lists containing the (evaluated) integer repeated an appropriate number of times). At least one of *List*, *MinList* and *MaxList* must be a list of fixed length at call time so that it is known how many indices are to be iterated.

multifor(*List*, *MinList*, *MaxList*, *IncrementList*)
: same as before, but *IncrementList* can be specified (i.e., how much to increment each element of *List* by). *IncrementList* must be either a list of arithmetic expressions evaluating to non-zero integers, or an arithmetic expression evaluating to a non-zero integer (in which case all elements are incremented by this amount). *IncrementList* defaults to 1.

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

param(*Var1*, *Var2*, ...)
: for declaring variables in *Goals* as global, i.e., as shared with the loop context, and shared among all iterations of the loop.

> CAUTION: By default, variables in *Goals* have local scope. This means that, in every iteration, these variables are new (even if a variable of the same name occurs outside the do-construct).

Note that **fromto/4** is the most general specifier (all the others could be implemented on top of it), while **foreach/2**, **foreacharg/2,3**, **foreachelem/2,3**, **foreachindex/2**, **count/3**, **for/3,4**, **multifor/3,4** and **param/*N*** are convenient shorthands.
There are three ways to combine the above specifiers in a single do loop:

IterSpec1*, *IterSpec2 ("synchronous iteration")
: This is the normal way to combine iteration specifiers: simply provide a comma-separated sequence of them. The specifiers are iterated synchronously; that is,

they all take their first "value" for the first execution of *Goals*, their second "value" for the second execution of *Goals*, etc. The order in which they are written does not matter, and the set of local loop variables is the union of those of *IterSpec1* and *IterSpec2*.

When multiple iteration specifiers are given in this way, typically not all of them will impose a termination condition on the loop (e.g., **foreach** with an uninstantiated list and **count** with an uninstantiated maximum do not impose a termination condition), but at least one of them should do so. If several specifiers impose termination conditions, then these conditions must coincide, i.e., specify the same number of iterations.

IterSpec1 * IterSpec2 ("cross product")
: This iterates over the cross product of *IterSpec1* and *IterSpec2*. The sequence of iteration is to iterate *IterSpec2* completely for a given "value" of *IterSpec1* before doing the same with the next "value" of *IterSpec1*, and so on. The set of local loop variables is the union of those of *IterSpec1* and *IterSpec2*.

IterSpec1 >> ***IterSpec2*** ("nested iteration")
: Like `( IterSpec1 do ( IterSpec2 do Goals ) )`, including with respect to scoping. The local loop variables are those of *IterSpec2*; in particular, those of *IterSpec1* are not available unless *IterSpec2* passes them through, e.g., using **param**. Similarly, the only "external" variables available as inputs to *IterSpec2* are the locals of *IterSpec1*; variables from outside the loop are not available unless passed through by *IterSpec1*, e.g., using **param**.

Syntactically, the do-operator binds like the semicolon, i.e., less than comma. That means that the whole do-construct should always be enclosed in parentheses (see examples).
Unless you use `:-pragma(noexpand)` or the compiler's `expand_goals:off` option, the do-construct is compiled into an efficient auxiliary predicate named *do__nnn*, where *nnn* is a unique integer. This will be visible during debugging. To make debugging easier, it is possible to give the loop a user-defined name by adding `loop_name(Name)` to the iteration specifiers. Name must be an atom, and is used as the name of the auxiliary predicate into which the loop is compiled (instead of *do__nnn*). The name should therefore not clash with other predicate names in the same module.
Finally, do-loops can be used as a control structure in grammar rules as well: A do-loop in a grammar rule context will generate (or parse) the concatenation of the lists of symbols generated (or parsed) by each loop iteration (the grammar rule transformation effectively adds a hidden fromto-iterator to a do-loop). The following rule will generate (or parse) a list of integers from 1 to N

```
intlist(N) --> ( for(I,1,N) do [I] ).
```

5.2.1 Examples

Iterate over a list:

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

Map a list (construct a new list from an existing list):

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

Compute the sum of a list of numbers:

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

Reverse a 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).
```

Iterate over integers from 5 down to 1:

```
(for(I,5,1,-1) do writeln(I)).
```

Make the 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** built-in is (almost)

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

Iterate `[I,J]` over `[1,1]`, `[1,2]`, `[1,3]`, `[2,1]`, ..., `[3,3]`:

```
(multifor([I,J],1,3) do writeln([I,J])).
```

Similar, but have different start/stop values for I and J:

```
(multifor([I,J], [2,1], [4,5]) do writeln([I,J])).
```

Similar, but only do odd values for the second variable:

```
(multifor(List, [2,1], [4,5], [1,2]) do writeln(List)).
```

Filter the elements of a list:

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

Iterate over the arguments of a structure:

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

Collect arguments in a list (in practice you would use **=..** to do this):

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

Collect arguments in reverse order:

```
(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,-1), foreach(A,List), param(S) do arg(I,S,A)).
```

Rotate the arguments of a structure:

```
S0 = s(a,b,c,d,e), functor(S0, F, N), functor(S1, F, N),
(foreacharg(X,S0,I), param(S1, N) do I1 is (I mod N)+1, arg(I1,S1,X)).
```

Flatten an array into a list:

```
(foreachelem(X,[]([](5,1,2),[](3,3,2))), foreach(X,List) do true).
```

Transpose a 2D array:

```
A = []([](5,1,2),[](3,3,2)), dim(A, [R,C]), dim(T, [C,R]),
(foreachelem(X,A,[I,J]), param(T) do X is T[J,I]).
```

Same, using **foreachindex**:

```
A = []([](5,1,2),[](3,3,2)), dim(A, [R,C]), dim(T, [C,R]),
(foreachindex([I,J],A), param(A, T) do
    subscript(A, [I,J], X), subscript(T, [J,I], X)).
```

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).
```

Now for some examples of 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)
    )
).
% or
Xs = [1,2,3,4],
( foreach(X, Xs) * foreach(Y, Xs) 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)
    )
).
```

or

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

Find all pairs of list elements and collect them in a result list:

```
pairs(Xs, Ys, Zs) :-
    (
        foreach(X,Xs),
        fromto(Zs, Zs4, Zs1, []),
        param(Ys)
    do
        (
            foreach(Y,Ys),
            fromto(Zs4, Zs3, Zs2, Zs1),
            param(X)
        do
            Zs3 = [X-Y|Zs2]
        )
    ).
```

or

```
pairs(Xs, Ys, Zs) :-
    (
        foreach(X, Xs) * foreach(Y, Ys),
        foreach(Z, Zs)
    do
        Z = X-Y
    ).
```

Flatten a 2-dimensional matrix into a list:

```
flatten_matrix(Mat, Xs) :-
    dim(Mat, [M,N]),
    (
        for(I,1,M),
        fromto(Xs, Xs4, Xs1, []),
```

```
            param(Mat,N)
        do
            (
                for(J,1,N),
                fromto(Xs4, [X|Xs2], Xs2, Xs1),
                param(Mat,I)
            do
                subscript(Mat, [I,J], X)
            )
        ).
```

Same using * to avoid nesting:

```
    flatten_matrix(Mat, Xs) :-
        dim(Mat, [M,N]),
        (
            for(I, 1, M) * for(J, 1, N),
            foreach(X, Xs),
            param(Mat)
        do
            subscript(Mat, [I,J], X)
        ).
```

Same using multifor to avoid nesting:

```
    flatten_matrix(Mat, Xs) :-
        dim(Mat, [M,N]),
        (
            multifor([I,J], 1, [M,N]),
            foreach(X, Xs),
            param(Mat)
        do
            subscript(Mat, [I,J], X)
        ).
```

Same for an array of arbitrary dimension:

```
    flatten_array(Array, Xs) :-
        dim(Array, Dims),
        (
            multifor(Idx, 1, Dims),
            foreach(X, Xs),
            param(Array)
        do
            subscript(Array, Idx, X)
        ).
```

Same but returns the elements in the reverse order:

```
flatten_array(Array, Xs) :-
    dim(Array, Dims),
    (
        multifor(Idx, Dims, 1, -1),
        foreach(X, Xs),
        param(Array)
    do
        subscript(Array, Idx, X)
    ).
```

Flatten nested lists one level (cf. flatten/2 which flattens completely):

```
List = [[a,b],[[c,d,e],[f]],[g]],
(foreach(Xs,List) >> foreach(X,Xs), foreach(X,Ys) do true).
```

Iterate over all ordered pairs of integers 1..4 (param(I) required to make I available in body of loop):

```
(for(I,1,4) >> (for(J,I+1,4), param(I)) do writeln(I-J)).
```

Same for general 1..N (param(N) required to make N available to second for):

```
N=4,
((for(I,1,N), param(N)) >> (for(J,I+1,N), param(I)) do writeln(I-J)).
```

5.3 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 array 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)
            )
        ).
```

5.3.1 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.

5.4 The String Data Type

In the Prolog community there are 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, ECL^{i}PS^e provides and heavily uses the string data type. It is familiar from other programming languages, and facilitates interfacing. It also offers programmers who are aware of the characteristics of the different data types a choice of most appropriate one. The system provides efficient built-ins for converting from one data type to another.

5.4.1 Choosing The Appropriate Data Type

Strings, atoms and character lists are written in similar ways, just distinguished by the type of quote:

```
"abc"     is a string
'abc'     is an atom
`abc`     is a character code list, equivalent to [97,98,99]
```

They 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. Maybe the main disadvantage of a character code list in an untyped language is that it is indistinguishable from a general list of small integers. This implies, for example, that the system cannot reliably decide whether to pretty-print a code list as a quoted string.

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, i.e., 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 **string_code/3** 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 are 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 to *index* 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: the matching clause is selected directly 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 other languages. 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, and lends itself to the use of Definite Clause Grammars (see 13.3).
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.

5.4.2 Built-in Support for Strings

Most ECL^{i}PS^e built-ins that deliver text objects (like **getcwd/1**, **read_string/3,4,5** and many others) return strings.
By means of the built-ins **atom_string/2**, **string_list/2,3**, **string_chars/2**, **number_string/2**, **term_string/2**, **text_to_string/2**, **atomics_to_string/2,3**, strings can easily be converted to and from other data types.
String manipulation is provided by builtins like **string_concat/3**, **string_code/3**, **string_codes/2**, **string_char/3**, **string_chars/2**, **split_string/4**, and **substring/5**. The regular expression library **library(regex)** also operates on strings.
The string stream feature (cf. section 11.3.1) allows strings to be opened like I/O streams, thus providing another way of creating or analysing strings.

5.5 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 in the clause head:

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

Using the `?-` operator in the neck of the clause (instead of `:-`) is an alternative way of expressing the same, so the following is equivalent to the above:

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

Matching clauses are not supported in dynamic clauses. A runtime error (calling an undefined procedure **-?->/1**) will be raised when executing dynamic code that has a matching clause head.
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* [5]. 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.

5.6 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.
Softcuts are written as:

A *-> **B** ; **C**

If A succeeds, B is executed and on backtracking subsequent solutions of A followed by B are returned, but C is never executed. If A fails straight away, C is executed. The behaviour of ***->/2** 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 6

The Compiler

6.1 Summary

The ECL^{i}PS^e compiler compiles ECL^{i}PS^e source (or Prolog source in various dialects) into the instructions of an abstract machine, which are then executed by an emulator.
Program source can be read in text form from files, console, strings and general input streams. Alternatively, it can be provided in the form of a data structure (list of clause terms).
The smallest program unit the compiler can meaningfully process is a predicate. In practice it is best to compile modules as a whole, since this allows for better consistency checks.
Usually, the generated code is immediately loaded into main memory and ready for execution. This method is the most convenient during program development. In addition, compiled code can be output to a file (ECL^{i}PS^e object format, or *eco*), from which it can later be loaded more quickly.
Compiled code optionally contains debugging information, allowing a source-oriented trace of program execution.

6.2 Compiler Invocation

The compiler is usually invoked by calling one of the following built-in predicates:

compile(*Source*) This is the standard compiler predicate. Source is usually a file name, other forms are detailed below. The contents of the file is compiled with the default compiler options.

compile(*Source*, *Options*) This is the standard compiler predicate. Source is usually a file name, other forms are detailed below. Options is a list of options to control the compilation process, see details below.

[*File1,...,FileN*] This predicate can be used as a shorthand for the **compile/1** predicate. It accepts a list of files, which can be source files or precompiled files.

compile_stream(*Stream*) This predicate compiles a given, open 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.

compile_stream(*Stream*, *Options*) Like compile_stream/1 but with options list.

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.

compile_term(*Clauses*, *Options*) Like **compile_term/2** but with options list.

When using a development environment like TkEclipse or Saros, the compiler is usually invoked implicitly via menu options or buttons.

6.2.1 Source Files

Program source is usually contained in files. The recommended file name suffixes (extensions) are

- `.ecl` for ECL^{i}PS^e specific source
- `.pl` for Prolog source

To compile a source files solver.ecl, any of the following forms is acceptable:

```
?- compile('solver.ecl').
?- compile("solver.ecl").
?- compile("/home/joe/solver.ecl").
?- compile("/home/joe/solver").
?- compile(solver).
```

File names must be single quoted (atom) or double quoted (string) if they contain punctuation, blank space, or start with an upper case letter. The `.ecl` extension can be omitted as long as no file without extension is present. A `.pl` extension can be omitted as long as no file without extension and no file with `.ecl` extension is present. The list of accepted suffixes and their precedence is given by the global flag `prolog_suffix`, see **get_flag/3**.
The following shorthands can be used, but note that the last two forms will load precompiled .eco files by preference, should they be present:

```
?- ['solver.ecl'].
?- ["solver.ecl"].
?- ["/home/joe/solver.ecl"].
?- ["/home/joe/solver"].
?- [solver].
```

If the source is given as `library(`*`Name`*`)`, the predicates looks for the file in the directories from the global flag `library_path`.
If File is the special atom 'user', the source will be taken from the current 'input' stream, i.e., will usually generate a prompt at which clauses can be typed in. In this case, input must be terminated either by typing `CTRL-D` (on Unix), `CTRL-Z` + `RETURN` (on Windows), or with the single atom `end_of_file`, followed by a fullstop (period).

```
?- [user].
 main :- writeln(hello).
^D
tty        compiled 72 bytes in 0.01 seconds
Yes (0.01 cpu)
```

```
?- main.
hello
Yes (0.00s cpu)
```

If File is the special form stream(Stream), then the source is taken from the given stream (which must be already opened). The stream content is compiled until the end of stream (or the `end_of_file` marker). Using this feature, any ECL^{i}PS^e stream (file, socket, tty, string, queue, pipe) can be used as the source for program text.

6.2.2 Main Compiler Options

The following compiler options affect the generated code:

debug: This option (`off/on`) determines whether the resulting code contains debugging information. If `off`, subgoals of the compiled predicates will not be visible to the debugger, the code will be significantly smaller, and slightly faster. The default value is taken from the global flag `debug_compile`. The setting can be changed via a pragma (`debug/nodebug`) in the code.

opt_level: Currently the integer 0 or 1, with 1 the default. Setting this to 0 will disable certain compiler optimizations and usually reduce performance. The setting can be changed via an `opt_level(Level)` pragma in the code.

The following options determine what is being done with the compilation result:

load: Determines whether the generated code is immediately loaded into memory, ready for execution. Values for the `load` option are:

- **all** (This is the default.) Load and replace code in memory, create/re-create all modules, interpret pragmas, and execute all directives and queries.
- **none** Do not load any code into memory, do not execute queries, but interpret pragmas and execute directives. Do not re-create modules, but create new ones and erase them again after compilation.
- **new** Do not overwrite any code in memory, but load new predicates. Do not execute queries, but interpret pragmas and execute directives. Do not re-create modules, but create new ones and erase them again after compilation. For existing modules, erase pragmas.

output: The abstract machine code which is the result of the compilation can be output in various forms. Possible values are:

- **none** (This is the default). No output (but code may be loaded, see `load` option).
- **eco** output compiled code in *eco* format to a file whose suffix is `.eco`. This format can be loaded using **ensure_loaded/1** or the compiler itself.
- **eco(*File*)** output compiled code in *eco* format to *File*.
- **asm** output compiled code in *asm* format to a file whose suffix is `.asm`. This format represents the code as WAM code that can be loaded back into ECLiPSe using the assembler (`lib(asm)`).
- **asm(*File*)** output compiled code in `asm` format to *File*.

outdir: Value is the destination directory for all output files. The default is the empty string `""`, meaning that all output files go into the same directory as the corresponding input file.

For other options see **compile/2**.
For example, to compile a program without debugging support directly into memory, use

```
?- compile(myprogram, [debug:off]).
```

The following command will create a precompiled file myprogram.eco from a source file called myprogram.ecl (or myprogram.pl):

```
?- compile(myprogram, [output:eco]).
```

6.3 Source Structure

The compiler normally reads files from beginning to end, but the file end can also be simulated with a clause

```
end_of_file.
```

When reading from a terminal/console, the end of the input can be marked by `CTRL-D` (in Unix-like systems) or `CTRL-Z`+`RETURN` on Windows.
When reading program source, the compiler distinguishes *clauses*, *directives* and *file queries*. Directives are terms with main functor **:-/1** while file queries have the main functor **?-/1**. Everything else is a program clause (see Appendix A).
The differences between a directive and a file query are as follows:

- File queries are general goals, and are executed when the program is loaded, i.e., when compiling with the load-option set to `all`, or when loading a compiled file. When compiling without loading, they are ignored.
- Directives can be general goals, in which case they are executed while the program is being compiled, and also when a compiled program is loaded.
- Some directives are not goals, but are interpreted by the compiler (or other source processing tool), e.g., module-directives or pragmas. These should not be combined with general goals in the same directive.

Directives and file queries should succeed and should only have a single solution. No results are printed by the system, failure leads to a warning, and an error condition will cause compilation to abort.

6.3.1 Clauses and Predicates

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 predicate. Normally, all clauses for one predicate should be consecutive in the source. If this is not the case, the compiler issues a warning and ignores the new clauses.
To change this behaviour, a **discontiguous/1** declaration must be used. The clauses are then collected and compiled as a whole once the end of the source unit (file or module) has been reached.
To add clauses for a predicate incrementally though several independent compiler invocations is only possible by declaring the corresponding predicate as **dynamic/1**, see Chapter 12.

6.3.2 Compilation and Modules

In the absence of module-directives (**module/1**, **module/3**) within the file, the file content is compiled into the module from which **compile/1,2** itself was called. This context module may be modified using the `@/2` notation, i.e., `compile(File, Options)@Module`. Existing static predicates will be redefined, and clauses for dynamic predicates appended to the existing ones (unless the 'load' option requests otherwise).
If the compiled file contains module directives (**module/1,3**), these specify to which module(s) the subsequent code belongs. Module directives are effective from the point where they occur until the next module directive, or until the end of file. If a module directive refers to a module that already exists, this module is erased and redefined (unless the 'load' option requests otherwise).
It is generally recommended to follow the *one file – one module* convention, and to make the base name of the file identical to the module name. In rare cases, it may make sense to have an auxiliary module in the same file as the main module. This is allowed, and every new module directive terminates the previous module.
To spread the code for one module over several files, use a top-level file containing the module directive plus one or more include-directives (section 6.4.3) for the component files.

6.3.3 Incrementality

When it encounters a **module/1** or **module/3** directive the compiler first erases previous contents of this module, if there was any, before starting to compile predicates into it. This means that in order to incrementally add predicates to a module, the module directive cannot be used because the previous contents of the module would be destroyed. Instead, the construct `compile(File)@Module` must be used.

6.4 Directives

6.4.1 Modules and Declarations

The following is a list of the directives most commonly used in source files:

:- module(*Name*). Beginning of a module.

:- module(*Name*, *Exports*, *Dialect*). Beginning of a module in a given dialect.

:- local *Specs*. Declaration of local items, e.g., syntax settings, operators, global storage, etc.

:- export *Specs*. Declaration of exported items, e.g., predicates, syntax settings, operators, etc.

:- reexport *Specs*. Declaration of reexported items.

:- import *Specs*. Declaration of imported modules or predicates.

:- use_module(*Mods*). Loading and importing of modules or libraries.

:- lib(*Libs*). Loading and importing of libraries.

:- meta_attribute(*Name*, *Handlers*) Declare a variable attribute.

:- comment(*Type*, *Info*) Structured program documentation.

6.4.2 Conditional Compilation

The compiler and other source-processing tools recognise the conditional compilation directives **if/1**, **elif/1**, **else/0** and **endif/0**. The first two take a goal as their argument, and parts of the program source can be included or excluded depending of the satisfiability of that goal. For example,

```
:- if(get_flag(hostarch, "i386_nt")).
    ...Windows-specific code...
:- elif( (get_flag(version_as_list,Version), Version @>= [6,0]) ).
    ...code for version 6.0 and later...
:- else.
    ...alternative code...
:- endif.
```

Note however, that only complete clauses or directives can be conditionally included.

6.4.3 Include Directives

Generally, it is best to use the module system to structure ECL^{i}PS^e applications, and to use one module per file. The modules then refer to each other via **use_module/1**, **lib/1**, or **import/1** directives. In rare cases it can make sense to split a single module into several files, which can then be pulled together using the following include directives:

:- include(*Files*). The contents of the given Files are treated as if they occurred in place of the include directive. Files is a single file name or a list of them.

:- [*Files*]. A synonym for the include/1 directive. Note that the semantics of this construct when used as a directive (include semantics) differs slightly from its use as a goal or query (compiler/loader invocation).

Included files can contain clauses, directives and queries, but should not contain **module/1,3** directives, since they would be interpreted as occurring within the including file, and the included module would not end at the end of the included file.

6.4.4 Compiler Pragmas

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

:- pragma(*Option*).

It is not possible to have several pragmas grouped together and separated by commas, 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, and any debug-option given to the compiler.

nodebug - generate optimized code with no debugger support. This overrides the global setting of the `debug_compile` flag, and any debug-option given to the compiler.

expand - do in-line expansion of built-ins like **is/2** and user-defined inline predicates. 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, and any expand-option given to the compiler.

noexpand - inhibit the in-line goal expansion. This pragma overrides the global setting of the `goal_expansion` flag, and any expand-option given to the compiler.

opt_level(*Level*) - override the `opt_level` option given to the compiler. *Level* is an integer greater or equal to 0. A zero setting disables all optional optimization.

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.

warnings - enable compiler warnings, overriding any warnings-option given to the compiler.

nowarnings - disable compiler warnings, overriding any warnings-option given to the compiler.

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

6.5 Precompiled (ECO) Files

6.5.1 Making Precompiled Files

ECL^{i}PS^e source files can be compiled into ECL^{i}PS^e object files, for subsequent loading. These files have the `.eco` suffix by default. This facility is mainly intended for module files. To create such a file, call the compiler with the appropriate output-option, e.g.,

```
?- compile(myprogram, [output:eco]).
```

This creates a precompiled file myprogram.eco from a source file called `myprogram.ecl` (or `myprogram.pl`). If the source file contained include directives, the result will be a single object file containing the compiled code of all included files. In earlier releases of ECL^{i}PS^e this was done using the **fcompile/1** predicate from the fcompile library, which is still supported for compatibility.
Loading of ECL^{i}PS^e object files is significantly faster than compilation from source. In ECL^{i}PS^e 6.0, ECL^{i}PS^e object files are text files containing a representation of the compiled abstract machine code, and can be used to deploy application code without revealing the source. The precompiled code is hardware and operating system independent. It may however not be portable between different versions of ECL^{i}PS^e if details of the abstract machine were modified between releases.
The global flag `eclipse_object_suffix` determines the file suffix used for ECL^{i}PS^e object files.

6.5.2 Restrictions

Currently, the compiler generates the auxiliary predicates for the do iterator using a module-wide counter to name the predicates. Unfortunately this means that if an object file with auxiliary predicates is loaded into a module that already has existing code that contains auxiliary predicates, naming conflict can occur and the old auxiliaries may be replaced. It is thus strongly recommended that object files should not be loaded into an existing module. This will only be a problem if the file does not contain any module declarations that redefine the module (i.e., **module/1**), as these redefinition will erase the old copy of the module.
One restriction does apply between platforms of different word sizes: integers which fit in the word size of one platform but not the other are represented differently internally in ECL^{i}PS^e. Specifically, integers which takes between 32 and 64 bits to represent are treated as normal integers on a 64 bit machine, but as bignums (see section 9.2.1) on 32 bit machines. This difference is normally invisible, but if such numbers occur as constants in the program code (i.e., their values appear textually), they can lead to different low-level compiled abstract code on the different platforms. Avoid using such constants if you want the object code to be portable across different word sizes (they can always be computed at run-time, e.g., writing `2^34` instead of `17179869184`).

6.5.3 Loading Precompiled Files

The following predicates either invoke the compiler or load precompiled `.eco` files. If the source specification does not specify the file type, precompiled files are preferred if they can be found in the search path:

[*File1,...,FileN*] This predicate can be used as a shorthand for the **compile** predicate, usually in the interactive toplevel. It accepts a list of files, which can be source files or precompiled files.

ensure_loaded(*Files*) This predicate compiles the specified file if it has not been compiled yet or if it has been modified since the last compilation. It can be used to load application code or system libraries.

use_module(*Files*) A combination of **ensure_loaded/1** and **import/1**.

lib(*Lib*) This predicate is used to ensure that a specified library file is loaded. It is equivalent to `ensure_loaded(library(Lib))`. If this library is not yet compiled or loaded, the system will look in all directories in the `library_path` flag for a file with this name, which is either a source file or a precompiled file, and compile or load it.

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

To implement reloading/recompiling when needed, the system keeps track of when a particular source files was compiled or precompiled file was loaded into memory. This information can be accessed explicitly through **current_compiled_file/3**.

6.5.4 Using the Compiler with a Makefile

To generate .eco file from source files, the compiler can be run from the command line using the -e option. To invoke it from a makefile, use the following suffix rule

```
.SUFFIXES:  $(SUFFIXES) .ecl .eco
.ecl.eco:
        eclipse -e "compile(\"$<\",[output:eco])"
```

or a pattern rule for Gnu make:

```
%.eco:  %.ecl
        eclipse -e "compile(\"$<\",[output:eco])"
```

6.6 Special Compiler Features

6.6.1 Compiling Non-Textual Source

A characteristic feature of Prolog and ECL^{i}PS^e is, that programs can be represented as data structures in a straightforward way. The compiler therefore provides the **compile_term/1** and **compile_term/2** interface predicates, which allow one to compile a list of terms. The compiler interprets these as clauses, directives and queries, similarly to what happens when the program source is being read from a file. For program generators, it is therefore not necessary to create a textual representation of generated code - the data structures can be compiled directly.
There are the following minor differences between compilation from textual sources and term compilation:

- Module directives are not supported - to compile code into a certain module, use the construct compile_term(Clauses,Options)@Module, and use **create_module/1** to create modules beforehand if necessary.
- Include directives do not make sense and are not supported.
- No end-of-compilation events are raised—**compile_term/1** behaves more like the compilation of an included file in this respect. This implies that discontiguous predicates are not supported.

A variant of **compile_term/2** is **compile_term_annotated/3** which takes source terms with source position annotations. This can be used when compiling auxiliary code within inlining/goal expansions transformations, without losing the source position information which is needed by the debugger.

6.6.2 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.

6.6.3 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 semantically 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**. For example, if we now compile the following predicate involving **double/2**:

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

then 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 global flag

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

6.6.4 Clause Expansion

Before compilation, the compiler also performs clause macro expansion (**macro/3**). This includes the DCG grammar rule expansion (section 13.3).

6.7 Writing Efficient Code

Even with a declarative language, there are certain constructs which can be compiled more efficiently than others. It is however not recommended to write unreadable code with the aim of achieving faster execution - intuition is often wrong about which particular construct will execute more efficiently in the end. The advice is therefore

Try the simple and straightforward solution first!

This will keep code maintainable, and will often be as fast or marginally slower than elaborate tricks. 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. ECL^{i}PS^e provides some support for finding those program parts that are worth optimizing.
To achieve the maximum speed of your programs, choose the following compiler options:

- `debug:off` ;
- `opt_level:1` (the default);
- `expand:on` (the default).

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 sped 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 port profiling 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.[1] It is therefore not necessary to reorder the predicate arguments so that the first one is the crucial argument for indexing. For example, in a predicate like

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

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

calls where the first argument is instantiated, like `p(d,Y)`, will be indexed on the first argument, while calls where the second argument is instantiated, like `p(X,b)`, will be indexed on the second. However, the decision is still based on only one argument at a time: a call like `p(d,b)` will be indexed on the first argument only (not because it is the first, but because it is more discriminating than the second). If it is crucial that such a procedure is executed as fast as possible with such a calling pattern, it can help to define an auxiliary procedure which will 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 (or beginning of a disjunction):

- Type testing predicates, i.e., **free/1**, **var/1**, **meta/1**, **atom/1**, **integer/1**, **rational/1**, **float/1**, **breal/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**.
- 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 = [], ...
```

Another example:

```
% A procedure with cuts
p(X{_}) ?- !, ...
p(X) :- var(X), !, ...
p(X) :- integer(X), ...
```

```
p(X) :- real(X), ...
p([H|T]) :- ...
p([]) :- ...
```

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 just as cheap as

  ```
  p :- q.
  ```

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

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

  is somewhat more expensive.

- When accessing an argument of a structure whose functor is known, unification and **arg/3** are both similarly efficient, so the question of whether to write `Struct = emp(_, X, _)` or `arg(2, Struct, X)` is just a matter of taste and style.

  We recommend that the structure notation (see section 5.1) be used, as it improves readability without adding any overhead. So, for example, use `Struct = emp{salary:X}` or `arg(salary of emp, Struct, X)`.

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

- Waking is more expensive (due to the priority mechanism) than metacalling which is more expensive than compiled calls. Metacalls however do not carry as heavy a penalty as 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.

- **=/2** and **==/2** are faster than **=:=/2**.

- **:/2** is optimised away by the compiler if both arguments are known.

- Starting from ECL^{i}PS^e 6.0, there is no performance difference between using multiple clauses or using disjunction or if-then-else cascades. In fact, the compiler normalises multiple clause predicates into a single-clause representation with inline disjunctions. Disjunctions are indexed.

- Conditionals (i.e., ...`->`...;...) are compiled more efficiently if the condition is an indexable built-in test.

6.8 Implementation Notes

The ECL^{i}PS^e compiler is actually contained in the eclipse library `lib(ecl_compiler)` which relies on a number of auxiliary modules. It uses **lib(source_processor)** to read programs, and produces abstract machine code that is assembled using **lib(asm)**.
The built-in predicate **als/1** or **asm:wam/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 7

Engines and Threads

Starting with release 7.0, ECL^iPS^e supports multiple engines and multi-threaded execution.
An **engine** is an entity with its own data areas and potentially independent control flow. This implies:

- engines can execute queries independently of each other
- each engine has its own search tree, backtracking in one engine does not affect others
- engines can operate in a concurrent or interleaved fashion
- communication between engines is explicit
- data transferred between engines is copied, variables cannot be shared

On the other hand, engines share or can share the following:

- loaded modules and predicates
- non-logical storage such as global variables, records, stores, shelves
- global settings (set_flag/2,3)
- streams

An engine can optionally be associated with a **thread**, allowing it to execute concurrently with other engines.
For more details, see the **Reference Manual Section on Engines**.

Chapter 8

The Module System

8.1 Basics

8.1.1 Purpose of Modules

The purpose of the module system is to provide a way to package a piece of code in such a way that

- internals are hidden;
- it has a clearly defined interface;
- naming conflicts are avoided.

In particular, this helps with

- Structuring of large applications: Modules should be used to break application programs into natural components and to define the interfaces between them.
- Provision of libraries: All ECL^{i}PS^e libraries are modules. Their interfaces are defined in terms of what the module makes visible to the world.
- Different implementations of the same predicate: In constraint programming it is quite common to have different implementations of a constraint, which all have the same declarative meaning but different operational behaviour (e.g., different amount of propagation, using different algorithms, exhibiting different performance characteristics). The module system supports that by allowing users to specify easily which version(s) of a predicate should be used in a particular context.

8.1.2 What is under Visibility Control?

The ECL^{i}PS^e module system governs the visibility of the following entities:

Predicate names Predicates can always be used in the module where they are defined and optionally in other modules when they are made available.

Structure names Structure declarations can be valid only locally within a module, or shared between several modules.

Syntax settings These include operator declarations (see **op/3**), syntax options and character classes. This means in particular that different modules can use different language dialects (e.g., ECL^{i}PS^e vs. ISO-Prolog).

Container names These include the names of record keys, non-logical variables and references. They are always local to the module where they are declared.

Initialization and finalization goals Modules can have initialization and finalization goals attached, see section 8.4.3.

Note that every definition (predicate, structure etc.) is in some module, there is no space outside the modules. When you don't explicitly specify a module, you inherit the module from the context in which you do an operation. When you are using an interactive ECL^{i}PS^e toplevel, a prompt will tell you in which module your input is read and interpreted.

8.1.3 What Modules are There?

The module system is flat, i.e., no module is part of another module, and module names must be unique. There are

- a few basic modules that are part of the ECL^{i}PS^e runtime system and are always there. The most important one is called `eclipse_language` and is by default imported into all other modules.
- the library modules: every library consists of at least one module. By convention, that module name is the library name and same as the base part of the library file name.
- the application-defined modules: these are created by the application programmer.
- in an interactive ECL^{i}PS^e toplevel there is one module in which queries entered by the user are read and executed. That module name is displayed in a prompt.

8.2 Getting Started

8.2.1 Creating a Module

You create a module simply by starting your program code with a **module/1** directive. This should usually be placed at the beginning of the source file and looks like

```
:- module(mymodule).
```

As a rule, the module name should be chosen to be the same as the file's base name (the file name without directory/folder and suffix/extension part). For example, the module `mymodule` might be contained in a file `mymodule.ecl`.
Anything you define in your module is by default **local** to that module.

8.2.2 Exporting

A definition is made available to the outside world by **exporting** it. All the exports of a module together form the module's **interface**. Exporting is done with the **export/1** directive, which can take different forms depending on the kind of the exported item.
Predicates are exported as follows:

```
:- export p/2.

p(X,Y) :-
        ...
```

Structures are exported by defining them with an **export/1** instead of a **local/1** directive, e.g.,

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

And the same holds for operators and other syntax settings:

```
:- export op(500, xfx, before).
:- export chtab(0'$, lower_case).
:- export syntax_option(no_array_subscripts).
:- export macro(pretty/1, tr_pretty/2, []).
```

All these declarations are valid locally in the module where they appear *and* in every module that imports them.
Initialization goals are exported as follows:

```
:- export initialization(writeln("I have been imported")).
```

Unlike the other declarations above, an exported **initialization/1** directive is *not* executed locally in they module where it appears, but *only* in the context of the module where it gets imported.[1]

8.2.3 Importing

In order to use a definition that has been exported elsewhere, it has to be **imported**. Often it is desirable to import another module's interface as a whole, i.e., everything it exports. This is achieved by an **import/1** directive of the form

```
:- import amodule.
```

If the module is in a file and has to be compiled first, then **use_module/1** can be used, which is a combination of **ensure_loaded/1** (see chapter 6) and **import/1**:

```
:- use_module("/home/util/amodule").
```

If the module is a library in one of ECL^{i}PS^e's library directories, then it can be loaded and imported by

```
:- use_module(library(hash)).
```

or simply using **lib/1** as in

```
:- lib(hash).
```

It is also possible to import only a part of another module's interface, using an **import-from directive**

```
:- import p/2 from amodule.
```

Note that this is the only form of import that can refer to a module that has not yet been loaded, and therefore allows a restricted form of circularity in the import structure.

[1]For local initialization use `:- local initialization(...).`

8.2.4 Definitions, Visibility and Accessibility

For a given predicate name and arity the following rules hold:

- Every module can contain at most one **definition**:
  - this definition may be local or exported.
- In every module, at most one definition is **visible**:
  - if there is a definition in the module itself, this is also the visible one in the module;
  - otherwise, if there is an (unambiguous) import or reexport, this is the visible one;
  - otherwise no definition is visible.
- All exported definitions are **accessible** everywhere:
  - this might require explicit module qualification (see 8.3.2).

8.3 Advanced Topics

8.3.1 Solving Name Conflicts

Name conflicts occur in two flavours:

Import/Import conflict: this is the case when two or more imported modules provide a predicate of the same name.

Import/Local conflict: this is the case when a local (or exported) predicate has the same name as a predicate provided from an imported module.

Conflicts of the first type are accepted silently by the system as long as there is no reference to the conflict predicate. Only when an attempt is made to access the conflict predicate is an error raised. The conflict can be resolved by explicitly importing one of the versions, e.g.,

```
:- lib(ria).                 % exports #>= / 2
:- lib(eplex).               % exports #>= / 2
:- import (#>=)/2 from ria.  % resolves the conflict
```

Alternatively, the conflict can remain unresolved and qualified access can be used whenever the predicates are referred to (see 8.3.2).
Conflicts of the second type give rise to an error or warning message when the compiler encounters the local (re)definition. To avoid that, an explicit **local/1** declaration has to be used:

```
:- local write/1.
write(X) :-   % my own version of write/1
    ...
```

Note that the **local/1**-declaration must occur textually before any use of the predicate inside the module.

8.3.2 Qualified Access via :/2

Normally, it is convenient to import predicates which are needed. By importing, they become visible and can be used within the module in the same way as local definitions. However, sometimes it is preferable to explicitly specify from which module a definition is meant to be taken. This is the case for example when multiple versions of the predicate are needed, or when the presence of a local definition makes it impossible to import a predicate of the same name from elsewhere. A call with explicit module qualification is done using **:/2** and looks like this:

```
lists:print_list([1,2,3])
```

Here, the module where the definition of **print_list/1** is looked up (the **lookup module**) is explicitly specified. To call **print_list/1** like this, it is not necessary to make **print_list/1** visible. The only requirement is that it is exported (or reexported) from the module `lists`.
Note that, if the called predicate is in operator notation, it will often be necessary to use brackets, e.g., in

```
..., ria:(X #>= Y), ...
```

The **:/2** primitive can be used to resolve import conflicts, i.e., the case where the same name is exported from more than one module and both are needed. In this case, none of the conflicting predicates is imported - an attempt to call the unqualified predicate raises an error. The solution is to qualify every reference with the module name:

```
:- lib(ria).    % exports #>= / 2
:- lib(eplex).  % exports #>= / 2

..., ria:(X #>= Y), ...
..., eplex:(X #>= Y), ...
```

Another case is the situation that a module wants to define a predicate of a given name but at the same time use a predicate of the same name from another module. It is not possible to import the predicate because of the name conflict with the local definition. Explicit qualification must be used instead:

```
:- lib(lists).

print_list(List) :-
        writeln("This is the list"),
        lists:print_list(List).
```

A more unusual feature, which is however very appropriate for constraint programming, is the possibility to call several versions of the same predicate by specifying several lookup modules:

```
..., [ria,eplex]:(X #>= Y), ...
```

which has exactly the same meaning as

```
..., ria:(X #>= Y), eplex:(X #>= Y), ...
```

Note that the modules do not have to be known at compile time, i.e., it is allowed to write code like

```
after(X, Y, Solver) :-
        Solver:(X #>= Y).
```

However, this is likely to be less efficient because it prevents compile-time optimizations.

8.3.3 Reexport - Making Modules from Modules

To allow more flexibility in the design of module interfaces, and to avoid duplication of definitions, it is possible to re-export definitions. A reexport is an import combined with an export. That means that a reexported definition becomes visible inside the reexporting module and is at the same time exported again. The only difference between exported and reexported definitions is that reexported predicates retain their original definition module.
There are 3 forms of the **reexport/1** directive. To reexport the complete module interface of another module, use

```
:- reexport amodule.
```

To reexport only an explicitly enumerated selection, use

```
:- reexport p/1,q/2 from amodule.
```

To reexport everything except some explicitly enumerated items, use

```
:- reexport amodule except p/2,q/3.
```

These facilities make it possible to extend, modify, restrict or combine modules into new modules, as illustrated in figure 8.1.

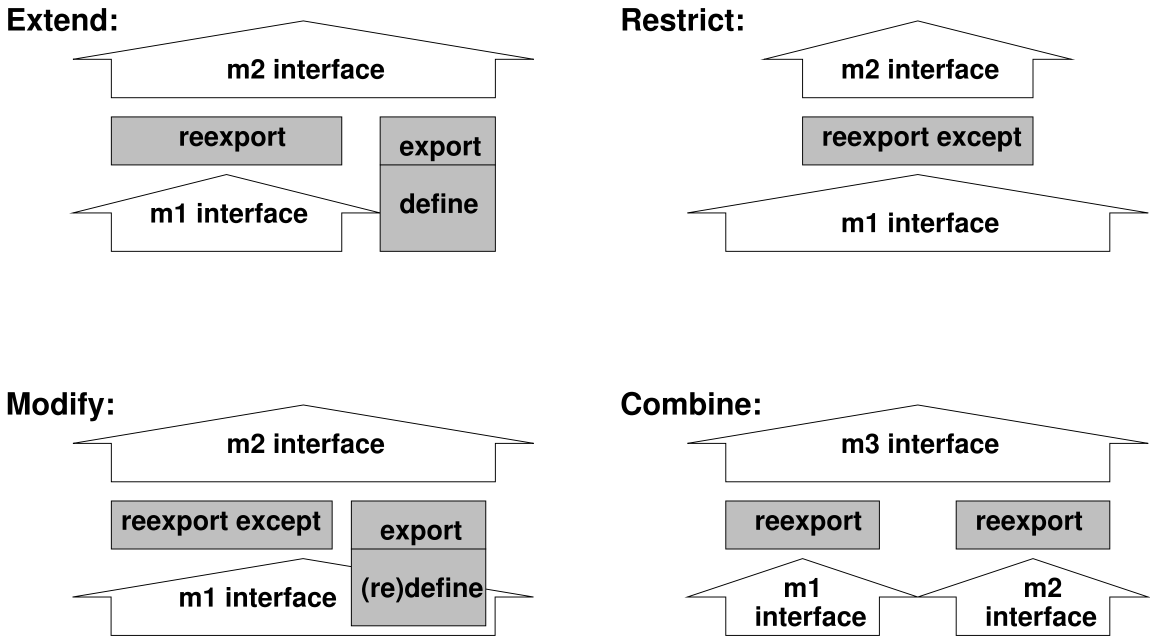


Figure 8.1: Making modules from modules with reexport

8.3.4 Modules and Source Files

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

Often it is the most appropriate to have one file per module and to have the file name match the module name.
It is however possible to have several modules in one file, e.g., a main module and one or more auxiliary modules - in that case the name of the main module should match the file name. Every module-directive in the file marks the end of the previous module and the start of the next one.
It is also possible to spread the contents of a module over several files. In this case, there should be a main file whose file name matches the module name, and the other files should be referenced from the main file using the **include/1** directive, e.g.,

```
:- module(bigmodule).
:- include(part1).
:- include(part2).
```

8.3.5 Tools and Context Modules

Tools

There are predicates in a modular system that must be able to determine from which module they were called (since this may be different from the module in which they were defined). The most common case is where a predicate is a meta-predicate, i.e., a predicate that has another goal or predicate name as an argument. Other cases are I/O predicates—they must be executed in a certain module context in order to obey the correct syntax of this module. In ECL^{i}PS^e, predicates that must be able to determine their context module are called **tool** predicates.[2]
Tool predicates must be declared. As a consequence, the system will automatically add a **context module** argument whenever such a tool predicate is called.
Consider for example a predicate that calls another predicate twice. The naive version of this predicate looks like

```
twice(Goal) :-
        call(Goal),
        call(Goal).
```

As long as no modules are involved, this works fine. Now consider the situation where the definition of **twice/1** and a call of **twice/1** are in two different modules:

```
:- module(stuff).
:- export twice/1.
twice(Goal) :-
        call(Goal),
        call(Goal).

:- module(main).
:- import stuff.

top :- twice(hello).

hello :- writeln(hi).
```

[2]Many Prolog systems call them *meta-predicates*.

This will not work because **hello/0** is only visible in module main and an attempt to call it from within **twice/1** in module stuff will raise an error. The solution is to declare **twice/1** as a tool and change the code as follows:

```
:- module(stuff).
:- export twice/1.
:- tool(twice/1, twice/2).
twice(Goal, Module) :-
        call(Goal)@Module,
        call(Goal)@Module.
```

What happens now is that the call to **twice/1** in module main

```
..., twice(hello), ...
```

is effectively replaced by the system with a call to **twice/2** where the additional argument is the module in which the call occurs:

```
..., twice(hello, main), ...
```

This context module is then used by **twice/2** to execute

```
..., call(hello)@main, ...
```

The **call(Goal)@Module** construct means that the call is supposed to happen *in the context* of module main.
The debugger trace shows what happens:

```
[main 5]: top.
  (1) 1 CALL  top
  (2) 2 CALL  twice(hello)
  (3) 3 CALL  twice(hello, main)
  (4) 4 CALL  call(hello) @ main
  (5) 5 CALL  call(hello)
  (6) 6 CALL  hello
S (7) 7 CALL  writeln(hi)
hi
S (7) 7 EXIT  writeln(hi)
  (6) 6 EXIT  hello
  ...
```

One complication that can arise when you use tools is that the compiler must know that a predicate is a tool in order to properly compile a call to the tool. If the call occurs textually before the tool declaration, this will therefore give rise to an *inconsistent tool redefinition* error. The **tool/2** declaration must therefore occur before any call to the tool.

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 inside user tool predicates), the **@/2** construct must be used, e.g.,

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

8.3.6 Lookup Module vs Context Module

The following table summarises the different call patterns with and without module specifications. There are only two basic rules to remember:

- **:/2** specifies the lookup module (to find the definition)
- **@/2** specifies the context module (where to execute)

Call inside module (m)	Module where definition of **twice/1** is looked up	Context module argument added to **twice/1**
..., twice(X), ...	m	m
..., lm : twice(X), ...	lm	m
..., twice(X) @ cm, ...	m	cm
..., lm : twice(X) @ cm, ...	lm	cm
..., call(twice(X)) @ cm, ...	cm	cm

8.3.7 The Module Interface

The primitive **current_module/1** can be used to check for the existence of a module, or to enumerate all currently defined modules.
Further details about existing modules can be retrieved using **get_module_info/3**, in particular information about the module's interface, what other modules it uses and whether it is locked (see 8.4.4).

8.3.8 Module-related Predicate Properties

Information about a predicate's properties can be retrieved using the **get_flag/3** primitive or printed using **pred/1**. The module-related predicate properties are:

defined (`on`/`off`) indicates whether code for the predicate has already been compiled. If not, only a declaration was encountered.

definition_module (an atom) the module where the predicate is defined.

visibility (`local`/`exported`/`reexported`/`imported`) indicates the visibility of the predicate in the context module.

tool (`on`/`off`) indicates whether the predicate has been declared a tool.

For tool predicates, **tool_body/3** can be used to retrieve the predicate it maps to when the module argument is added.
To get information about a predicate visible in a different module, use for instance

```
get_flag(p/3, visibility, V) @ othermodule
```

8.4 Less Common Topics

8.4.1 Modules That Use Other Languages

Modules created with **module/1** automatically import the module `eclipse_language`, which provides the standard set of ECL^{i}PS^e built-in predicates. To create a module that uses a different language dialect, use **module/3**. For instance

```
:- module(mystdcode, [], iso).
```

creates a module in which you can use ISO Standard Prolog,[3] but not all of ECL^{i}PS^e's usual language features. Note that the third argument (here `iso`) simply specifies a library which implements the desired language, so new languages can be added easily.

8.4.2 Creating and Erasing Modules at Runtime

A module can also be created explicitly by a running program with **create_module/1** or **create_module/3** 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. The same holds for trying to erase essential system modules.

8.4.3 Initialization and Finalization

Sometimes modules have global state which must be initialised or finalised. For this purpose, modules can have

Local Initialization Goals: these are specified as

```
:- local initialization(Goal).
```

and are executed just after the module containing them has been loaded.

Exported Initialization Goals: these are specified as

```
:- export initialization(Goal).
```

and are executed whenever the module containing the declaration gets imported into another module. The call will happen in the context of the importing module.

Finalization Goals: these are specified as

```
:- local finalization(Goal).
```

and are executed just before the module containing them gets erased. Modules can get erased either explicitly through **erase_module/1** or implicitly when the module is recompiled, or when the ECL^{i}PS^e session is exited. Finalization goals should not do any I/O because in the case of an embedded ECLiPSe, I/O may no longer be available at finalization time.

8.4.4 Locking Modules

By default, ECL^{i}PS^e does not strictly enforce the hiding of module internals. This facilitates program development, as it allows the user to inspect and trace without being too concerned about module boundaries. For example, you can set a spy point on a local predicate **p/3** in module *othermod* by calling:

```
:- spy(p/3)@othermod.
```

[3] To the extent implemented by ECL^{i}PS^e's compatibility library.

Once a module implementation is stable and there is a need for privacy, it is possible to *lock* a module. Locking makes it impossible to access internal, local items from outside the module. Of course, the module can still be used though its interface. The built-in predicates related to locking are **lock/0** which provides a definitive lock, **lock_pass/1** which allows subsequent unlocking using a password (**unlock/2**), and **get_module_info/3** which allows to check whether a module is locked. **lock/0** and **lock_pass/1** are usually used as a directive in the source file of the module to be locked.

Chapter 9

Arithmetic Evaluation

9.1 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*.

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

9.1.1 Arithmetic Evaluation vs Arithmetic Constraint Solving

This chapter deals purely with the evaluation of arithmetic expressions containing numbers. No uninstantiated variables must occur within the expressions at the time they are evaluated. This is exactly like arithmetic evaluation in procedural languages.

As opposed to that, in arithmetic constraint solving one can state equalities and inequalities involving variables, and a constraint solver tries to find values for these variables which satisfy these constraints. Note that ECL^{i}PS^e uses the same syntax in both cases, but different implementations providing different solving capabilities. See the chapter *Common Solver Interface* in the Constraint Library Manual for an overview.

9.2 Numeric Types and Type Conversions

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

9.2.1 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, for example:

```
0  3  -5  1024  16'f3ae  0'a  15511210043330985984000000
```

Note that integer range is unlimited if ECL^{i}PS^e was compiled with bignum support. Otherwise, integers are restricted to that representable in a machine word, and `max_integer flag` of **get_flag/2** returns the maximum integer value.

9.2.2 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, 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.
Rationals are supported if ECL^{i}PS^e is compiled with bignum support. If rationals are not supported, a type error will be raised when a rational is required.

9.2.3 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 uses IEEE double precision floats with the following conventions:

- overflows always produce infinity results, never overflow exceptions.
- invalid operations always produce arithemtic exceptions, never NaNs.
- positive (0.0) and negative zero (-0.0) are distinct and do not unify.

9.2.4 Bounded Real Numbers

It is a well known problem that floating point arithmetic suffers from rounding errors. To provide safe arithmetic over the real numbers, ECL^{i}PS^e also implements bounded reals[1]. A bounded real consists of a pair of floating point numbers which constitute a safe lower and upper bound for the real number that is being represented. Bounded reals are written as two floating point numbers separated by two underscores, e.g.,

```
-0.001__0.001  3.141592653__3.141592654  1e308__1.0Inf
```

A bounded real is a representation for a real number that definitely lies somewhere between the two bounds, but the exact value cannot be determined [2]. Bounded reals are usually not typed in by the user, they are normally the result of a computation or type coercion.
All computations with bounded reals give safe results, taking rounding errors into account. This is achieved by doing interval arithmetic on the bounds and rounding the results outwards. The resulting bounded real is then guaranteed to enclose the true real result.
Computations with floating point values result in uncertainties about the correct result. Bounded reals make this uncertainty explicit. A consequence of this is that sometimes it is conceptually not possible to decide whether two bounded reals are identical or not. This occurs when the bounds of the compared intervals overlap. In this case, the arithmetic comparisons leave a (ground) delayed goal behind which can then be inspected by the user to decide whether the match is considered close enough. The syntactical comparisons like **=/2** and **==/2** treat bounded reals simply as a pair of bounds, and consider them equal when the bounds are equal.

9.2.5 Type Conversions

Note that numbers of different types never unify, e.g., 3, 3_1, 3.0 and 3.0__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 → float → bounded real

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 **integer**, **rational**, **float** and **breal** functions.

9.3 Arithmetic Functions

9.3.1 Predefined Arithmetic Functions

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

[1] We have chosen to use the term *bounded real* rather than *interval* in order to avoid confusion with interval variables as used in the interval arithmetic constraint solvers

[2] This is in contrast to a floating point number, which represents a real number which lies somewhere in the vicinity of the float

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
ceiling(E)	round up to integral value	number	number
round(E)	round to nearest integral value	number	number
truncate(E)	truncate to 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 (truncate)	integer × integer	integer
E1 rem E2	integer remainder	integer × integer	integer
E1 div E2	integer division (floor)	integer × integer	integer
E1 mod E2	integer modulus	integer × integer	integer
gcd(E1,E2)	greatest common divisor	integer × integer	integer
lcm(E1,E2)	least common multiple	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
copysign(E1,E2)	combine value and sign	number × number	number
nexttoward(E1,E2)	next representable number	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	real
cos(E)	trigonometric function	number	real
tan(E)	trigonometric function	number	real
asin(E)	trigonometric function	number	real
acos(E)	trigonometric function	number	real
atan(E)	trigonometric function	number	real
atan(E1,E1)	trigonometric function	number × number	real
exp(E)	exponential function e^x	number	real
ln(E)	natural logarithm	number	real
sqrt(E)	square root	number	real
pi	the constant pi = 3.1415926...	—	float
e	the constant e = 2.7182818...	—	float
fix(E)	convert to integer (truncate)	number	integer
integer(E)	convert to integer (exact)	number	integer
float(E)	convert to float	number	float
breal(E)	convert to bounded real	number	breal
rational(E)	convert to rational	number	rational
rationalize(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(Es)	sum of elements	vector	number
sum(Es*Es)	scalar product	vector*vector	number
min(Es)	minimum of list elements	vector	number
max(Es)	maximum of list elements	vector	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*, *float* or *breal* with the type conversions as specified above. As a result type, *number* stands for *the more general of the argument types*, and *real* stands for *float* or *breal*. 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 integer division `//` rounds the result towards zero (truncates), while the `div` division rounds towards negative infinity (floor). Each division function is paired with a corresponding remainder function: (`rem` computes the remainder corresponding to `//`, and `mod` computes the remainder corresponding to `div` [3]. The remainder results differ only in the case where the two arguments have opposite signs. The relationship between them is as follows:

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

This table gives an overview:

	10 x 3	-10 x 3	10 x -3	-10 x -3
//	3	-3	-3	3
rem	1	-1	1	-1
div	3	-4	-4	3
mod	1	2	-2	-1

9.3.2 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 *T1*, *T2* etc. 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.

9.3.3 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:

[3] Caution: In ECL^{i}PS^e versions up to 5.8, `mod` was the remainder corresponding to `//`, i.e., behaved like `rem`

```
[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, e.g.,

```
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. For example, 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**.

9.3.4 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. For example, 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.

9.4 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.

9.5 The Multi-Directional Arithmetic Predicates

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:

succ(X, Y) True if *X*and *Y* are natural numbers, and *Y* is one greater than *X*. At most one of *X*, *Y* can be a variable.

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.

These predicates 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 :

```
[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.

9.6 Arithmetic and Coroutining

Arithmetic comparisons can be delayed until their arguments are instantiated instead of generating an instantiation fault by passing the comparison to the suspend solver (see section 18.3). This gives a form of coroutining.

Chapter 10

Non-logical Storage and References

10.1 Introduction

This chapter describes primitives that allow one to break the normal logic programming rules in two ways:

- information can be *saved across logical failures* and backtracking;
- information can be accessed by *naming* rather than by argument passing.

Obviously, these facilities must be used with care and should always be encapsulated in an interface that provides logical semantics.
ECLiPSe provides several facilities to store information across backtracking. The following table gives an overview. If at all possible, the handle-based facilities (bags, record, shelves and stores) should be preferred because they lead to cleaner, reentrant code (without global state) and reduce the risk of memory leaks.

Facility	Type	Access	See
bags	unordered bag	by handle	bag_create/1
records	ordered list	by handle	record_create/1
shelves	array	by handle	shelf_create/2,3
stores	hash table	by handle	store_create/1
named records	ordered list	by name	record/1,2
named shelves	array	by name	shelf/2
named stores	hash table	by name	store/1
non-logical variables	single cell	by name	variable/1
non-logical arrays	array	by name	array/1,2
dynamic predicates	ordered list	by name	dynamic/1, assert/1

The other facility described here, *Global References*, does not store information across failure, but provides a means to give a name to an otherwise logical data structure. See section 10.8.

10.2 Bags

A **bag** is an anonymous object which can be used to store information across failures. A bag is unordered and untyped. Any ECL^{i}PS^e term can be stored in a bag. Bags are referred to by a handle. An empty bag is created using **bag_create/1**, data is stored in the bag by invoking **bag_enter/2**, and the stored data can be retrieved as a list with **bag_retrieve/2** or **bag_dissolve/2**. A typical application is the implementation of the **findall/3** predicate or similar functionality. As opposed to the use of **record/2** or **assert/1**, the solution using bags is more efficient, more robust, and trivially reentrant.

```
simple_findall(Goal, Solutions) :-
    bag_create(Bag),
    (
        call(Goal),
        bag_enter(Bag, Goal),
        fail
    ;
        true
    ),
    bag_dissolve(Bag, Solutions).
```

10.3 Records

A **record** is an anonymous or named object which can be used to store information across failures. A typical application is the collection of multiples solutions during backtracking. Another application is communication between engines/threads.

A record is a list of (copies of) arbitrary terms. New terms can be added at either end of the list (**recorda/2**, **recordz/2**), and list elements can be removed from any position in the list (**erase/2**). Elements can be retrieved individually (**recorded/2**) or collectively (**recorded_list/2**).

Records come in two flavours: anonymous records are created with **record_create/1** and referred to by handle, while named records are created with a **record/1** declaration and referred to by their name within a module. If possible, anonymous records should be preferred because they make it easier to write robust, reentrant code. For example, an anonymous record automatically disappears when the system backtracks over its creation, or when the store handle gets garbage collected. Named records, on the other hand, must be explicitly destroyed in order to free the associated memory.

A typical application is the implementation of the **findall/3** predicate or similar functionality:

```
simple_findall(Goal, Solutions) :-
    record_create(Record),
    (
        call(Goal),
        recordz(Record, Goal),
        fail
    ;
        true
    ),
    recorded_list(Record, Solutions).
```

For an example involving thread communication, see **record_wait_append/4**.

10.4 Shelves

A **shelf** is an anonymous object which can be used to store information across failures. A typical application is counting of solutions, keeping track of the best solution, aggregating information across multiple solutions, etc.

A shelf is an object with multiple slots whose contents survive backtracking. The content of each slot can be set and retrieved individually, or the whole shelf can be retrieved as a term.

Shelves come in two flavours: anonymous shelves are created with **shelf_create/2** or **shelf_create/3** and referred to by handle, while named shelves are created with a **shelf/ 2** declaration and referred to by their name within a module. If possible, anonymous shelves should be preferred because they make it easier to write robust, reentrant code. For example, an anonymous shelf automatically disappears when the system backtracks over its creation, or when the store handle gets garbage collected. Named shelves, on the other hand, must be explicitly destroyed in order to free the associated memory.

All shelf slots are initialized when the shelf is created. Data is stored in the slots (or the shelf as a whole) with **shelf_set/3** and retrieved with **shelf_get/3**.

Example: Counting how many solutions a goal has:

```
count_solutions(Goal, Total) :-
    shelf_create(count(0), Shelf),
    (
        call(Goal),
        shelf_get(Shelf, 1, Old),
        New is Old + 1,
        shelf_set(Shelf, 1, New),
        fail
    ;
        shelf_get(Shelf, 1, Total)
    ).
```

In this particular example, we could have used **shelf_inc/2** to increment the counter.

10.5 Stores

A **store** is an anonymous object which can be used to store information across failures. A typical application is aggregating information across multiple solutions. Note that, if it is not necessary to save information across backtracking, the use of the **library(hash)** is more appropriate and efficient than the facilities described here.

A store is a hash table which can store arbitrary terms under arbitrary ground keys. Modifications of a store, as well as the entries within it, survive backtracking. The basic operations on stores are entering and looking up information under a key, or retrieving the store contents as a whole.

Stores come in two flavours: anonymous stores are created with **store_create/1** and referred to by handle, while named stores are created with a **store/ 1** declaration and referred to by their name within a module. If possible, anonymous stores should be preferred because they make it easier to write robust, reentrant code. For example, an anonymous store automatically disappears when the system backtracks over its creation, or when the store handle gets garbage

collected. Named stores, on the other hand, must be explicitly destroyed in order to free the associated memory.
Data is entered into a store using **store_set/3** and retrieved using **store_get/3**. It is also possible to retrieve information: either all keys with **stored_keys/2**, or the full contents of the table with **stored_keys_and_values/2**. Entries can be deleted via **store_delete/2** or **store_erase/1**.
A typical use of stores is for the implementation of memoization. The following is an implementation of the Fibonacci function, which uses a store to remember previously computed results. It consists of the declaration of a named store, a wrapper predicate **fib/2** that handles storage and lookup of results, and the standard recursive definition **fib_naive/2**:

```
:- local store(fib).

fib(N, F) :-
    ( store_get(fib, N, FN) ->
        F = FN                     % use a stored result
    ;
        fib_naive(N, F),
        store_set(fib, N, F)       % store computed result
    ).

fib_naive(0, 0).
fib_naive(1, 1).
fib_naive(N, F) :-
    N1 is N-1, N2 is N-2,
    fib(N1, F1), fib(N2, F2),
    F is F1 + F2.
```

Using this definition, large function values can be efficiently computed:

```
?- fib(300, F).
F = 222232244629420445529739893461909967206666939096499764990979600
Yes (0.00s cpu)
```

The next example shows the use of an anonymous store, for computing how many solutions of each kind a goal has. The store is used to maintain counter values, using the solution term as the key to distinguish the different counters:

```
solutions_profile(Sol, Goal, Profile) :-
    store_create(Store),
    (
        call(Goal),
        store_inc(Store, Sol),
        fail
    ;
        stored_keys_and_values(Store, Profile)
    ).
```

Running this code produces for example:

```
?- solutions_profile(X, member(X, [a, b, c, b, a, b]), R).
X = X
R = [a - 2, b - 3, c - 1]
Yes (0.00s cpu)
```

10.6 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 toincrement 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. That is, if **count_solutions/2** were called recursively from inside *Goal*, this would smash the counter and yield incorrect results.[2]
The visibility of a non-logical variable is local to the module where it is first set. It is good style to declare them using **local/1 variable/1** declarations. For example, in the above example one should use

[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

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

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

If it is necessary to access the value of a variable in other modules, exported access predicates should be provided.

10.7 Non-logical Arrays

Non-logical arrays are a generalisation of the non-logical variable, capable of storing multiple values. An array has to be declared in advance. It has 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 **local/1 array/1** declaration, e.g.,

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

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)
```

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 `float` (for floating point 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 initialized to zero in the case of `byte`, `integer` and `float`, 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 initialized 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: *Array* is then an atom, and the type is `prolog`.

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

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

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

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

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

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

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

10.8 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 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 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).
```

or

```
:- local reference(p, 0).
```

creates a named reference *p* (with an initial value of 0) which can be used to store references to terms. The reference is accessed and modified via **setref/2** and **getref/2**[3].
The following points are different for references:

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

```
[eclipse 1]: local reference(r).
yes.
[eclipse 2]: Term = p(X), setref(r, Term), getref(r, Y), Y == Term.
X = X
Y = p(X)
Term = p(X)
yes.

[eclipse 3]: local variable(v).
yes.
[eclipse 4]: Term = p(X), setval(v, Term), getval(v, Y), Y == Term.
no (more) solution.
```

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

```
[eclipse 5]: setref(r, 1), (setref(r, 2), getref(r, X) ; getref(r, Y)).
X = 2
Y = Y     More? (;)

X = X
Y = 1
```

- references (and the associated terms) are private to each **engine**. This implies that the referenced terms are not shared between engines, and each engine may have a different term associated to a particular reference name.
- 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.

The use of references should be considered carefully. Their overuse can lead to programs which are difficult to understand and difficult to optimize. Typical applications use at most a single reference per module, for example to hold state that would otherwise have to be passed via additional arguments through many predicate invocations.

[3] For backward compatibility, setval/2 and getval/2 also work on references, but this use is deprecated: use setref/2 and getref/2 instead.

Chapter 11

Input and Output

11.1 Streams

Input and output in ECL^{i}PS^e is done via communication channels called **streams**. They are usually associated with I/O devices (a file, a terminal, a socket, a pipe), or in-memory queues or buffers.
The streams may be opened for input only (**read mode**), output only (**write mode**), or for both input and output (**update mode**).

11.1.1 Predefined Streams

Every ECL^{i}PS^e session defines the following symbolic stream names, which indicate the *current streams* for certain categories of input/output:

input Used by the input predicates that do not have an explicit stream argument, e.g., **read/1**. This is by default the same as user_input and stdin, but can be redirected.

output Used by the output predicates that do not have an explicit stream argument, e.g., **write/1**. This is by default the same as user_output and stdout, but can be redirected.

error Output for error messages and all messages about exceptional states. This is by default the same as user_error and stderr, but can be redirected.

warning_output Used by the system or user programs to output warning messages. This is by default the same as user_output and stdout, but can be redirected.

log_output Used by the system for information messages (e.g. garbage collection), or by user programs for e.g. messages about computational progress. This is by default the same as user_output and stdout, but can be redirected.

The above streams can be freely redirected, but are initially set to one of the following three *default streams*, to which they will also be reset whenever a redirection ends:

user_input The default input stream. This is initially the same as stdin, but can be redirected.

user_output The default output stream. This is initially the same as stdout, but can be redirected.

user_error The default error stream. This is initially the same as stderr, but can be redirected.

Finally, there are the four predefined *standard streams*, which cannot be closed or redirected. Apart from the `null` stream, there is usually no need to refer to them explicitly:

stdin The standard input stream.

stdout The standard output stream.

stderr The standard error stream.

null A dummy stream, output to it is discarded, on input it always gives end of file.

In a stand-alone ECL^{i}PS^e stdin, stdout and stderr are connected to the corresponding standard I/O descriptors of the process. In an embedded ECL^{i}PS^e, the meaning of stdin, stdout and stderr is determined by the ECL^{i}PS^e initialization options.

Current Stream	Default Stream	Standard Stream
input	user_input	stdin
output	user_output	stdout
warning_output	user_output	stdout
log_output	user_output	stdout
error	user_error	stderr
		null

Initial assignment of symbolic stream names

For compatibility with Prolog systems, the system accepts the stream name `user` in certain places. Its meaning is identical to `stdin` and `stdout`, depending on the context where it is used.

11.1.2 Stream Handles and Aliases

Streams can be identified by anonymous *stream handles* or by symbolic names.[1] Most of the built-in predicates that require a stream to be specified have a stream argument at the first position, e.g., `write(Stream, Term)`. This argument can be either a stream handle or a symbolic stream name.
Streams that are opened by programs should preferably use stream handles, as this allows the system to better keep track of their lifetime. Nevertheless, alias names can be given, either immediately when a stream is newly opened (e.g. with **open/4**), or later via redirection (**set_stream/2**). A stream can have more than one symbolic alias.
To obtain a handle when only an alias is known, use **get_stream/2**:

```
get_stream(StreamOrAlias, Handle)
```

get_stream/2 can also be used to check whether two stream names are aliases of each other. Note that stream handles are not normal Prolog terms! They can not be assembled, decomposed, or occur literally in Prolog text.

[1]Earlier ECL^{i}PS^e versions identified streams by small integers, which is now deprecated, except for some foreign language interfaces. If needed, the number is available as the physical_stream stream property.

11.1.3 Opening New Streams

Streams provide a uniform interface to a variety of I/O devices and pseudo-devices. The following table gives an overview of how streams on the different devices are opened.

I/O device	How to open
tty	implicit (`stdin`, `stdout`, `stderr`) or **open/3** of a device file
file	**open(*FileName*, *Mode*, *Stream*)**
string	**open(string(*String*), *Mode*, *Stream*)**
queue	**open(queue(*String*), *Mode*, *Stream*)**
pipe	**exec/2**, **exec/3** and **exec_group/3**
socket	**socket/3** and **accept/3**
null	implicit (null stream)

How to open streams onto the different I/O devices

Most streams are 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:

- absolute 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 11.3.1.
If *SourceSink* is of the form `queue(`*`InitString`*`)` a pseudo-pipe in memory is opened, see section 11.3.2.
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 symbolic stream identifier or an uninstantiated variable. If it is uninstantiated, the system will bind it to an anonymous stream handle:

```
[eclipse 1]: open(new_file, write, Stream).
Stream = $&(stream,7)
Yes (0.00s cpu)
```

If the stream argument is an atomic name, this name becomes an alias for the (hidden) stream number:

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

This is equivalent to

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

The stream identifier (symbolic or handle) 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), close(Stream).
```

will write the atom *subject* to the file 'foo' and close the stream subsequently.
It is recommended style *not* to use symbolic stream names in code that is meant to be reused. This is because these stream names are global, there is the possibility of name clashes, and the code will not be reentrant. It is cleaner to open streams with a variable for the stream identifier and pass the resulting handle as an argument wherever it is needed.
Socket streams are not opened with **open/3**, but with the special primitives **socket/3** and **accept/3**. More details are in chapter 22.
A further group of primitives which open streams implicitly consists of **exec/2**, **exec/3** and **exec_group/3**. They open **pipe streams** which connect directly to the I/O channels of the executed process. See chapter 21 for details.

11.1.4 Closing Streams

A stream lives until it is closed. Streams that are only referenced by handle are closed automatically, either on failure across the open/3,4 predicate, or after all copies of their handle become unused and garbage collected. This means that no extra precautions have to be taken to ensure that streams are closed on failure or when aborting. Handle-streams can optionally be closed explicitly if their lifetime is statically known in the program. Streams that have aliases cannot be closed automatically: all aliases must be closed explicitly.
The predicates **close/1**, **2**

close(*Stream*)
close(*Stream*,*Options*)

are used to explicitly close an open stream. If a stream has several alias names, closing any of them will close the actual stream. All the other aliases should be closed as well (or redirected to streams that are still open), because otherwise they will continue to refer to an identifier of the already closed stream.
When an attempt is made to close a redirected system stream (e.g., `output`), the stream is closed, but the system stream is reset to its default setting.

11.1.5 Redirecting Streams

The **set_stream/2** primitive can be used to redirect an already existing symbolic stream to a new actual stream. This is particularly useful to redirect e.g., the default `output` stream:

```
set_stream(output, MyStream)
```

so that all standard output is redirected to some other destination (e.g., an opened file instead of the terminal). Note that the stream modes (read/write) must be compatible. The redirection is terminated by calling

```
close(output)
```

which will reestablish the original meaning of the output stream by resetting it to the user_output default stream.

11.1.6 Finding Streams

The predicate

current_stream(?*Stream*)

can be used to backtrack over all the currently opened streams, and obtain handles for them (but not their aliases).

11.1.7 Stream Properties

A stream's properties can be accessed using **get_stream_info/3**:

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

e.g., its mode, line number, file name etc. Some stream properties can be modified using **set_stream_property/3**:

set_stream_property(+*Stream*, +*Property*, +*Value*)

e.g., the end-of-line sequence used, the flushing behaviour, the event-raising behaviour, the prompt etc.

11.2 Communication via Streams

The contents of a stream may be interpreted in one of 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.

11.2.1 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 ISO-8859-1 (iso_latin_1) code of the character. If the end of file has been reached then an exception is raised and -1 is returned. 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.

11.2.2 Token I/O

The predicates **read_token/2** and **read_token/3**:

read_token(*Token*, *Class*)
read_token(*Stream*, *Token*, *Class*)

represent the second way of interpreting stream contents. They read the next token from the current input stream, unify 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. The exact definition of character classes and tokens can be found in appendices A.2.1 and A.2.3, respectively.
A further, very flexible possibility to read a sequence of characters is provided by the built-in **read_string/5**

read_string(*Stream*, *SepChars*, *PadChars*, *Separator*, *String*)

Here, the input is read up to a specified delimiter, and returned as an ECL^{i}PS^e string.
In particular, one line of input can be read as follows:

```
read_line(Stream, String) :-
    read_string(Stream, end_of_line, "", _Separator, String).
```

The *SepChar* argument allows the specification of padding characters, which will be ignored before and after separators. Once a string has been read, string manipulation predicates like **split_string/4** can be used to break it up into even smaller components.

11.2.3 Term I/O

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

```
read(Term)
```

reads the next ECL^{i}PS^e 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. The exact definition of the term syntax can be found in appendix A.
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.
For additional information about other options for reading terms, in particular for how to get variable names, refer to **readvar/3**, **read_term/2** and **read_term/3**. For reading and processing complete ECL^{i}PS^e source code files, use the **library(source_processor)**.
For output, 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. For more details about how to output terms in different formats, see section 11.4.
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

```
_N
```

where *N* is a number which identifies the variable (but note that these numbers may change on garbage collection and can therefore not be used to identify the variable in a more permanent way). Occasionally the number will be prefixed with the lower-case letter `l`, indicating that the variable is in a short-lived memory area called the local stack (see 20).

11.2.4 Newlines

Newlines should be output using either **nl/0**, **nl/1**, **writeln/1**, **writeln/2**, or using the `%n` format with **printf/2**, **printf/3**. All those will produce a LF or CRLF sequence, depending on the stream property settings (see **set_stream_property/3**).

11.2.5 General Parsing and Text Generation

Reading and writing of I/O formats that cannot be handled by the methods discussed above are probably best done using Definite Clause Grammar (DCG) rules. See chapter 13.3 for details.

11.2.6 Flushing

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.
It is possible to configure a stream such that it is automatically flushed at every line end (see **set_stream_property/3**).

11.2.7 Prompting

Input streams on terminals can be configured to print a prompt whenever input is required, see **set_stream_property/3**.

11.2.8 Positioning

Streams that are opened on files or strings can be positioned, i.e., the read/write position can be moved forward or backwards. This is not possible on pipes, sockets, queues and terminals.
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.

11.3 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.

11.3.1 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 a call to the **open/3** predicate:

```
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.
```

11.3.2 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).
X = "hello"
yes.
[eclipse 13]: write(q, "rld"), read(q, X).
X = 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.
```

11.4 Term Output Formats

11.4.1 Write_term and Printf

The way ECL^{i}PS^e terms are printed can be customised in a number of ways. The most flexible predicates to print terms are **write_term/3** and **printf/3**. They both allow all variants of term output, but the format is specified in a different way. The following figure gives an overview.

write_term/2,3 output option	**printf/2,3** %..w	Meaning
`as(term)`		do not assume any particular meaning of the printed term
`as(clause)`	`C`	print the term as a clause (apply clause transformations)
`as(goal)`	`G`	print the term as a goal (apply goal transformations)
`attributes(none)`		do not print any variable attributes
`attributes(pretty)`	`m`	print attributes using the corresponding print handlers
`attributes(full)`	`M`	print the full contents of all variable attributes
`compact(false)`		print blank space around operators, after commas, etc.
`compact(true)`	`K`	don't print blank space unless necessary
`depth(`*Max*`)`	<*Max*>	print the term only up to a maximum nesting depth of *Max* (a positive integer)
`depth(0)`		observe the stream-specific or global flag `print_depth`
`depth(full)`	`D`	print the whole term (may loop when the term is cyclic!)
`dotlists(false)`		write lists in square bracket notation, e.g., `[a,b]`
`dotlists(true)`	`.`	write lists as terms with functor **./2**
`newlines(false)`		print newlines inside quotes as the escape sequence `\n`
`newlines(true)`	`N`	print newlines as line breaks even inside quotes
`nl(false)`		do not add a newline
`nl(true)`	`L`	print a newline sequence (as with nl/1) after the term.
`fullstop(false)`		do not add a fullstop
`fullstop(true)`	`F`	terminate the term with a fullstop, so it can be read back.
`flush(false)`		do not force a flush after printing the term
`flush(true)`	`b`	flush after printing the term
`numbervars(false)`		do not treat **'$VAR'/1** terms specially
`numbervars(true)`	`I`	print terms of the form `'$VAR'(`*N*`)` as named variables: `'$VAR'(0)` is printed as `A`, `'$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.
`operators(true)`		obey operator declarations and print prefix/infix/postfix
`operators(false)`	`O`	ignore operator declarations and print functor notation
`portrayed(false)`		do not use **portray/1,2**
`portrayed(true)`	`P`	call the user-defined predicate **portray/1,2** for printing
`precedence(Prec)`		print assuming given context precedence
`quoted(false)`		do not print quotes around strings or atoms
`quoted(true)`	`Q`	quote strings and atoms if necessary
`transform(true)`		apply portray transformations (write macros)
`transform(false)`	`T`	do not apply portray transformations (write macros).
`variable_names(VarNames)`		use the given variable names when printing the corresponding variables. VarsNames is a list of terms of the form Name=Var.
`variables(default)`		print variables using their source name (if available)
`variables(raw)`	`v`	print variables using a system-generated name, e.g., `_123`
`variables(full)`	`V`	print variables as source name plus number, e.g., `Alpha_132`
`variables(anonymous)`	`_`	print every variable as a simple underscore

Overview of term output options (see **write_term/3** for more details)

The **write_term/2** and **write_term/3** predicates print a single ECL^{i}PS^e term and accept a list of output options (first column in the table).
The **printf/2** and **printf/3** predicates are similar to C's printf(3) function, but provide additional format characters for printing ECL^{i}PS^e terms. The basic format string for printing arbitrary terms is `%w`. Additional format characters can go between `%` and `w`, according to the second column in the table.
For example, the following pairs of printing goals are equivalent:

```
printf("%mw",  [X])  <->   write_term(X, [attributes(pretty)])
printf("%O.w", [X])  <->   write_term(X, [operators(false),dotlist(true)])
printf("%5_w", [X])  <->   write_term(X, [depth(5),variables(anonymous)])
```

11.4.2 Other Term Output Predicates

The other term output predicates **write/1**, **writeln/1**, **writeq/1**, **write_canonical/1**, **display/1**, **print/1** can all be defined in terms of **write_term/2** (or, similarly in terms of **printf/2**) as follows:

```
write(X)   :- write_term(X, [numbervars(true)]).
writeln(X) :- write_term(X, [numbervars(true)]), nl.
writeq(X)  :- write_term(X, [variables(raw), attributes(full),
              transform(false), quoted(true), depth(full),
              numbervars(true)]).
write_canonical(X) :-
              write_term(X, [variables(raw), attributes(full),
              transform(false), quoted(true), depth(full),
              dotlist(true), operators(false)]).
display(X) :- write_term(X, [dotlist(true), operators(false)]).
print(X)   :- write_term(X, [portrayed(true), numbervars(true)]).
```

11.4.3 Default Output Options

It is possible to set default output options for an output stream in order to globally affect all output to this particular stream. The **set_stream_property/3** predicate can be used to assign default options (in the same form as accepted by **write_term/3**) to a stream. These options will then be observed by all output predicates which do not override the particular option.

Chapter 12

Dynamic Code

Support for dynamic code is provided partly for compatibility with Prolog. It is worth noting that ECL^{i}PS^{e}provides much better primitives (see chapter 10) to support the non-logical storage of information—a major use for dynamic predicates in Prolog.
An ECL^{i}PS^e predicate can be made *dynamic*. That is, it can have clauses added and removed from its definition at run time. This chapter discusses how to do this, and what the implications are.

12.1 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, each of 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 will be just silently accepted without erasing any clauses and without printing an error message.

12.2 Altering programs at run time

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.
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 built-ins 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.

12.3 Differences between static and dynamic code

- Only dynamic procedures can have clauses added or removed at run time.
- Matching clauses (section 5.5) are not supported by dynamic code. A runtime error (about calling an undefined procedure `-?->/1`) will be raised when executing dynamic code that has a matching clause head.
- Clauses for a dynamic procedure need not be consecutive.
- Source tracing is not supported for dynamic procedures.
- **assert/1**, **retract/1** and **clause/1** do not perform clause transformation on the clause. If clause transformation is required, this can be done explicitly with **expand_clause/2** before.
- Internally, dynamic procedures are represented differently from static procedures. The execution of dynamic procedures will generally be slower than for static procedures.

[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))`.

Chapter 13

ECL^{i}PS^e Macros

13.1 Introduction

ECL^{i}PS^e provides a general mechanism to perform macro expansion of Prolog terms. Macro expansion can be performed in 3 situations:

read macros are expanded just after a Prolog term has been read by the ECL^{i}PS^e parser. Note that the parser is not only used during compilation but by all **term-reading** predicates.

compiler macros are expanded only during compilation and only when a term occurs in a certain context (clause or goal).

write macros are expanded just before a Prolog term is printed by one of the output predicates

In addition to transforming a term, macros can also be *source annotation aware*, and provide source annotation information for the transformed term if supplied with source annotation information for the orginal term. Source annotation information is about the source and position of a term, and is provided by the predicate **read_annotated/3**.
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 **local** or **exported**. The macro expansion is performed by a user-defined Prolog predicate.

13.2 Using the macros

The following declarations and built-ins control macro expansion:

local macro(+*TermClass*, +*TransPred*, +*Options*) defines a macro for the given *TermClass*. The transformation itself will be performed by the predicate *TransPred*.

export macro(+*TermClass*, +*TransPred*, +*Options*) as above, except that the macro is available to other modules.

erase_macro(+*TermClass*, +*Options*) erases the macro that is currently defined for *TermClass*. Note that this can only be done in the module where the definition was made.

current_macro(?*TermClass*, ?*TransPred*, ?*Options*, ?*Module*) can be used to get 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 the following: `compound`, `string`, `integer`, `rational`, `float`, `breal`, `atom`, `goal`.[1]

The *+TransPred* argument specifies the predicate that will perform the transformation. It has to be either of arity 2 or 3 and should have the form:

```
trans_function(OldTerm, NewTerm [, Module]) :- ... .
```

or it can be source annotation aware, and be of arity 4 or 5, as follows:

```
trans_function(OldTerm, NewTerm, OldAnn, NewAnn [, Module]) :- ... .
```

At transformation time, the system will call *TransPred* in the module where **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. In the case of the source annotation aware version of *TransPred*, if the term was read in by **read_annotated/2,3**, the annotated version of the term to transformed is passed in the third argument, and the transformation should bind the fourth argument to the annotated transformed term; otherwise, if no source annotation information is available, the third argument is passed in as a free variable, and the transformation should not bind the fourth argument. In both *TransPred* cases, the optional last argument is the module where the term was being read in. See section 13.2.1 for more details on annotated terms.

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:

- 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**, etc.[2] Write macros are applied using the `C` option in the **printf/2** predicate.

  goal: Goal-read-macros are transformed only if the term is a subgoal in the body of a program clause. Goal-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).

[1] `type(goal)` stands for suspensions.

[2] Note that clause transformation is *not* performed with **assert/1**, **retract/1** and **clause/1**. This is a change from previous versions of ECL^{i}PS^e.

The following shorthands exist:

local/export portray(+*TermClass*, +*TransPred*, +*Options*): portray/3 is like **macro/3**, but the write-option is implied.

inline(+*PredSpec*, +*TransPred*): inline/2 is the same as a goal-read-macro. The visibility is inherited from the transformed predicate.

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

:- local 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).
:- export macro(no_macro_expansion/1, trprotect/2, [protect_arg]).
```

A local macro is only visible in the module where it has been defined. When it is defined as exported, then it is copied to all other modules that contain a **use_module/1** or **import/1** for this module. The transformation function should also be exported in this case. There are a few global macros predefined by the system, e.g., for `-->/2` (grammar rules, see below) or **with/2** and **of/2** (structure syntax, see section 5.1). These predefined macros can be hidden by local macro definitions.

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).
 :- local 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.
 :- local macro(s/1, tr_s/2, [write]).

yes.
[eclipse 2]: write(s(s(s(0)))).
3
yes.
```

13.2.1 Source annotation aware macro transformations

When the macro transformation predicate has 4 or 5 arguments, it is termed source annotation aware, and the extra arguments are used to specify how source information from the original term should be mapped to the transformed term.
An annotated term provides the source information about a term. It is structurally similar to the original term and contains all information about the term, plus additional type information, variable names, and source position annotations for all subterms.
The structure of the descriptive terms is as follows:

```
:- export struct(annotated_term(
                                term,     % var, atomic or compound
                                type,     % term type (see below)
                                file,     % source file name (atom)
                                line,     % source line (integer)
                                from, to  % source position (integers)
                  ...
                 )).
```

The type-field describes the type of the original term and provide type information similar to those used in **type_of/2**, except that they convey additional information about variables and `end_of_file`.
In the case of atomic terms and variables, the term-field simply contains the plain original term. For compound terms, the term-field contains a structure whose functor is the functor of the plain term, but whose arguments are annotated versions of the plain term arguments.
For example, the annotated term representing the source term `foo(bar, X, _, 3)` is:

```
annotated_term(foo(
                  annotated_term(bar, atom, ...),
                  annotated_term(X, var('X'), ...),
                  annotated_term(_, anonymous, ...),
                  annotated_term(3, integer, ...)
                 ),
               compound,
               ...
              )
```

The file/line/from/to-fields of an annotated term describe the "source position" of the term, as follows:

file The canonical file name of the source file (an atom), or the empty atom `''` if the source is not a file or is not known.

line The line number in the source stream (positive integer).

from, to The exact term position as integer offsets in the source stream, starting at *from* and ending at $to - 1$.

The extra arguments for the transformation predicate are a pair of annotated terms for the original and transformed term. The predicate will be supplied with the annotated term for the original term if available, and the predicate is responsible for specifying the annotated term for the transformed term—the structure of the transformed annotated term must match the annotated term structure expected for the transformed term. If no annotated information is available, the original annotated term will be a variable, and the predicate must not bind the transformed annotated term.
For an example, here is a source annotation aware version of the previous **trans_a/2** example:

```
[eclipse 1]: [user].
...
```

```
trans_a(a(X,Y), b(Y), AnnA, AnnTrans) :-
        number(X),
        X > 0,
        ( var(AnnA) ->
              true      % no source information, leave AnnTrans as var
        ;
              AnnA = annotated_term{term:a(_AnnX, AnnY),
                                    file:File, line:Line,
                                    from:From,to:To},
              AnnTrans = annotated_term{term:b(AnnY),
                                    type: compound,
                                    file:File, line:Line,
                                    from:From,to:To}
         ).

:- local macro(a/2, trans_a/4, []).

Yes (0.23s cpu)
[eclipse 2]: read_annotated(user, X, Y).
 a(3,bar(X)).

X = b(bar(X))
Y = annotated_term(b(annotated_term(bar(annotated_term(X, var('X'), user, 18,
654, 655)), compound, user, 18, 650, 654)), compound, user, 18, 646, 648)
```

In the example, the main functor of the transformed predicate, **b/1**, inherits the annotation information for the original term's principal functor, **a/2**. The argument Y in the transformed term takes the annotation information from the corresponding argument in the original term. The source annotation aware transformation predicate facility is provided to allow the user to access the details of how the subterms of the original term are mapped to the transformed term. Without this extra information, the whole of the transformed term is given the source information (source position, source file etc.) of the original source term. This extra information is useful when the subterms are goals, because without the extra information, source tracing of these goals during debugging will not be done.

13.3 Definite Clause Grammars — DCGs

Grammar rules are described in many standard Prolog texts ([3]). In ECL^{i}PS^e they are provided by a predefined global[3] 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.
```

[3]So that the user can redefine it with a local one.

```
grammar_head --> non_terminal, [','], terminal.

grammar_body --> grammar_body, [','], grammar_body.
grammar_body --> grammar_body, [';'], grammar_body.
grammar_body --> grammar_body, ['->'], grammar_body.
grammar_body --> grammar_body, ['|'], grammar_body.
grammar_body --> iteration_spec, ['do'], 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 syntactically identical to prolog goals (atom, compound term or variable), the terminals are lists of prolog terms (typically characters or tokens). Every term is transformed, unless it is enclosed in curly brackets. The control constructs like conjunction **,/2**, disjunction (**;/2** or **|/2**), the cut (**!/0**), the condition (**->/1**) and do-loops need not be enclosed in curly brackets.
The grammar can be accessed with the built-in **phrase/3**. The first argument of **phrase/3** is the name of the grammar to be used, the second argument 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] ; ....
```

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) --> pronoun(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].

pronoun(plural) --> [you].
```

The above grammar may be applied by using **phrase/3**. 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 input 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

A grammar rule is translated to a Prolog clause 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 calling **expand_clause/2**:

```
[eclipse 1]: expand_clause(p(X) --> q(X), Expanded).

X = X
Expanded = p(X, _250, _243) :- q(X, _250, _243)
Yes (0.00s cpu)
[eclipse 2]: expand_clause(p(X) --> [a], Expanded).

X = X
Expanded = p(X, _251, _244) :- 'C'(_251, a, _244)
Yes (0.00s cpu)
```

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 input sources other than a list, e.g., to parse directly from an I/O stream:

```
:- local 'C'/3.
'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.
```

Here is another redefinition of **'C'/3**, using a similar idea, which allows direct parsing of ECL^{i}PS^e strings as sequences of characters:

```
:- local 'C'/3.
'C'(String-Pos0, Char, String-Pos1) :-
        string_code(Pos0, String, Char),
        Pos1 is Pos0+1.

anagram --> [].
anagram --> [_].
anagram --> [C], anagram, [C].
```

This can then be applied to a string as follows:

```
[eclipse 1]: phrase(anagram, "abba"-1, "abba"-5).
yes.
[eclipse 2]: phrase(anagram, "abca"-1, "abca"-5).
no (more) solution.
```

Unlike the default definition, these redefinitions of **'C'/3** are not bi-directional. Consequently, the grammar rules using them can only be used for parsing, not for generating sentences.
Note that every grammar rule uses that 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:

Events
Events have the following properties:

- they may occur asynchronously (when posted by the environment) or synchronously (when raised by the program itself);
- they are handled synchronously by a handler goal that is inserted into the resolvent;
- the handler can cause the interrupted execution to fail or to abort;
- the handler can interact with the interrupted execution only via non-logical features (e.g., global variable or references);
- the handler can cause waking of delayed goals via symbolic triggers.

Errors
Errors can be viewed as 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.

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, but may be mapped into a sychronous event;
- certain predefined actions (like aborting) can be performed asynchronously.

14.1 Events

14.1.1 Event Identifiers and Event Handling

Events are identified by names (atoms) or by anonymous handles.
When an event is raised, 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**).

Creating Named Events

A named event is created by defining a handler for it using **set_event_handler/2**:

```
:- set_event_handler(hello, my_handler/1).
my_handler(Event) :-
    <code to deal with Event>
```

A handler for a named event can have zero or one arguments. When invoked, the first argument is the event identifier, in this case the atom `hello`. It is not possible to pass other information to the handler.
The handler for a defined event can be queried using **get_event_handler/3**.

Creating Anonymous Events

An anonymous event is created with the built-in **event_create/3**:

```
..., event_create(my_other_handler(...), [], Event), ...
```

The built-in takes a handler goal and creates an anonymous event handle Event. This handle is the only way to identify the event, and therefore must be passed to any program location that wants to raise the event. The handler goal can be of any arity and can take arbitrary arguments. Typically, these arguments would include the Event handle itself and other ground arguments (variables should not be passed because when the event is raised, a copy of the handler goal with fresh variables will be executed).

14.1.2 Raising Events

Events can be raised in the following different ways:

- Explicitly by the ECL^{i}PS^e program itself, using **event/1**.
- By foreign code (C/C++) using the `ec_post_event()` function.
- Via signals/interrupts by setting the interrupt handler to **event/1**.
- Via I/O streams (e.g., queues can be configured to raise an event when they get written into).
- Via timers, so-called after-events

Raising Events Explicitly

To raise an event from within ECL^{i}PS^e code, call **event/1** with the event identifier as its argument. If no handler has been defined, a warning will be raised:

```
?- event(hello).
WARNING: no handler for event in hello
Yes (0.00s cpu)
```

The event can be an anonymous event handle, e.g.,

```
?- event_create(writeln(handling(E)), [], E), event(E).
handling($&(event,"371bqz"))
E = $&(event,"371bqz")
Yes (0.00s cpu)
```

Raising events explicitly is mainly useful for test purposes, since it is almost the same as calling the handler directly.

Raising Events from Foreign Code

To raise an event from within foreign C/C++ code, call

```
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 is embedded into a foreign code host program.

Timed Events ("after" events)

An event can be triggered after a specified amount of elapsed time. The event is then handled sychronously by ECL^{i}PS^e. These events are known as "after" events, as they are set up so that the event occurs *after* a certain amount of elapsed time. They are setup by one of the following predicates:

event_after(+*EventId*, +*Time*) This sets up an event *EventId* so that the event is raised once after *Time* seconds of elapsed time from when the predicate is executed. *EventId* is an event identifier and *Time* is a positive number.

event_after_every(+*EventId*, +*Time*) This sets up an event *EventId* so that the event is raised repeatedly every *Time* seconds: first *Time* seconds after the invocation of the predicate, then *Time* seconds after that event was raised, and so on.

events_after(+*EventList*) This sets up a series of after events specified in the list *EventList*, which contains events of the form *EventId-Time*, or *EventId-every(Time)* (specifying a single event or a repeated event respectively).

The *Time* parameter is actually the minimum of elapsed 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, i.e., where it can make a procedure call.

Once an after event has been set up, it is pending until it is raised. In the case of events caused by an invocation of **event_after_every/2**, the event will always be pending because it is raised repeatedly. A pending event can be cancelled so that it will not be raised.

cancel_after_event(+*EventId*, -*Cancelled*) This finds and cancels all pending after events with name *EventId* and returns the actually cancelled ones in a list.

current_after_events(-*Events*) This returns a list of all pending after events.

The after event mechanism allows multiple events to make use of the timing mechanism independently of each other. The same event can be setup multiple times with multiple calls

to **event_after/2** and **event_after_every/2**. The **cancel_after_event/2** predicate will cancel all instances of an event.
By default, the after event feature uses the `real` timer. The timer can be switched to the `virtual` timer, in which case the elapsed time measured is user CPU time.[1] This setting is specified by the ECL^{i}PS^e environment flag `after_event_timer` (see **get_flag/2**, **set_flag/2**). Note that if the timer is changed while some after event is still pending, these events will no longer be processed. The timer should therefore not be changed once after events are initiated.
Currently, the `virtual` timer is not available on the Windows platform. In addition, the users should not make use of these timers for their own purposes if they plan to use the after event mechanism.

14.1.3 Events and Waking

Using the suspension and event handling mechanisms together, a goal can be added to the resolvent and executed after a defined elapsed time. To achieve this, the goal is suspended and attached to a symbolic trigger, which is triggered by an afer-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. Note that if the execution finishes before the suspended goal is due to be woken up, it will also not enter the resolvent and will thus not be executed.
The following is an example of waking a goal with a timed event. Once `monitor(X)` is called, the current value of X will be printed every second until the query finishes or is backtracked over:

```
:- 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).

:- monitor(Var), <do_something>.
```

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**.

14.1.4 Aborting an Execution with Events

Typically, event handlers would perform some action and then succeed, letting the interrupted exectuion continue unharmed. Event handlers for asynchronous events should never fail, because the failure will be inserted in a random place in the resolvent, and the effect will be

[1]This is time that the CPU spends on executing user code, i.e., the ECL^{i}PS^e program.

unpredictable. It is however sometimes useful to allow an asynchronous event to abort an execution (via **throw/1**), e.g., to implement timeouts.[2]
When dealing with events that occur asynchronously (in particular after-events), and event handlers that cause the execution to abort, it is often a problem that event handlers may be interrupted or preempted by other event handlers. This can be avoided by use of the event-defer mechanism. An event can be declared with the defer-property, which means that all further event handling is temporarily suppressed as soon as the handling of this event begins. In this case, the event handler is responsible for reenabling event handling explicitly before returning by calling **events_nodefer/0**. For instance:

```
:- set_event_handler(my_event, defers(my_handler/0)).
my_after_handler :-       % event handling is deferred at this point
        <deal with event>,
        events_nodefer.   % allow other events to be handled again
```

In the presence of other event handlers which can cause aborts, this will protect the handler code from being preempted.

14.2 Errors

Error handling is one particular use of events. The main property of error events is that they have a culprit goal, i.e., 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 C. User-defined errors have atomic names, they are the same as events. 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 **throw/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 built-in that is implemented as an external, these choicepoints are discarded.[3]

[2] Since implementing reliable timeouts is a nontrivial task, we recommend the use of **lib(timeout)** for this purpose.

[3] This is necessary because the compiler recognises simple predicates as deterministic at compile time and so

14.2.1 Error Handlers

The predicate **set_event_handler/2** is used to assign a procedure as an error handler. The call

```
set_event_handler(ErrorId, PredSpec)
```

sets the event handler for error type *ErrorId* to the procedure specified by *PredSpec*, which must be of the form *Name/Arity*.
The corresponding predicate **get_event_handler/3** may be used to identify the current handler for a particular error. The call

```
get_event_handler(ErrorId, PredSpec, HomeModule)
```

will, provided *ErrorId* is a valid error identifier, unify *PredSpec* with the specification of the current handler for error *ErrorId* 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_event_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(ErrorId, Goal)
error(ErrorId, Goal, Module)
```

are provided to raise the error *ErrorId* (an error number or a name atom) with the culprit *Goal*. Inside tool procedures it is usually necessary to use **error/3** in order to pass the context 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 error numbers, 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

```
util: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.

---

if a simple predicate were to cause the invocation of a non-deterministic error handler, the generated code might no longer be correct.

14.2.2 Arguments of Error Handlers

An error handler has four optional arguments:

1. The first argument is the number or atom that identifies the error.

2. 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.

3. The third argument is only defined for a subset of the existing errors. If the error occurred inside a tool body, it holds the context module, otherwise it is identical to the fourth argument.[4]

4. The fourth argument is the lookup module for the culprit goal. This is needed for example when the handler wants to call the culprit reliably, using a qualified call via **:/2**.

The error handler is free to ignore some of these arguments, i.e., it can have any arity from 0 to 4. 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 **throw/1** may be made (see the section on the **catch/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 catch-block (i.e., back to the corresponding call of the **catch/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.

The following two templates are the most common for error handlers. The first simply prints an error message and aborts:

```
my_error_handler(ErrorId, Goal, ContextModule) :-
        printf(error, "Error %w in %w in module %w%n",
                [ErrorId,Goal,ContextModule]),
        abort.
```

The following handler tries to repair the error and call the goal again:

```
my_error_repair_handler(ErrorId, Goal, ContextModule, LookupModule) :-
        % repair the error
        ... some code to repair the cause for the error ...
        % try call the erroneous goal again
        LookupModule : Goal @ ContextModule.
```

[4]Note that some events are not errors but are used for different purposes. In those cases the second and third argument are sometimes used differently. See Appendix C for details.

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, and it is recommended to use the event mechanism described above (the `ec_post_event()` library function) to communicate between the host application and the ECL^{i}PS^e code. However, even in this setting, ECL^{i}PS^e can also handle signals directly, provided the programmer sets up a corresponding interrupt handler.

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 cannot be caught by the user (e.g., the *kill* signal), trying to establish a handler for them yields an error message. Note that *PredSpec* should be one of the predefined handlers. The use of general user defined predicates is deprecated because of portability considerations.
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_event_handler**:

get_interrupt_handler(*N*, *PredSpec*, *HomeModule*)

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.
The following is the list of predefined 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, i.e., the interrupt is ignored.

throw/1
: Invoke **throw/1** with the interupt's symbolic name.

abort/0
: Invoke `throw(abort)`.

halt/0
: Terminate the ECL^{i}PS^e process.

internal/0
: Used by ECL^{i}PS^e to implement internal functionality like the profiler. This is not intended to be used by the user.

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 being called when the appropriate interrupt occurs. This general asynchronous interrupt handling is not supported on all hardware/platforms, neither in an embedded ECL^{i}PS^e (including the TkECL^{i}PS^e development environment), and is therefore deprecated.

Chapter 15

Debugging

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:

- 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

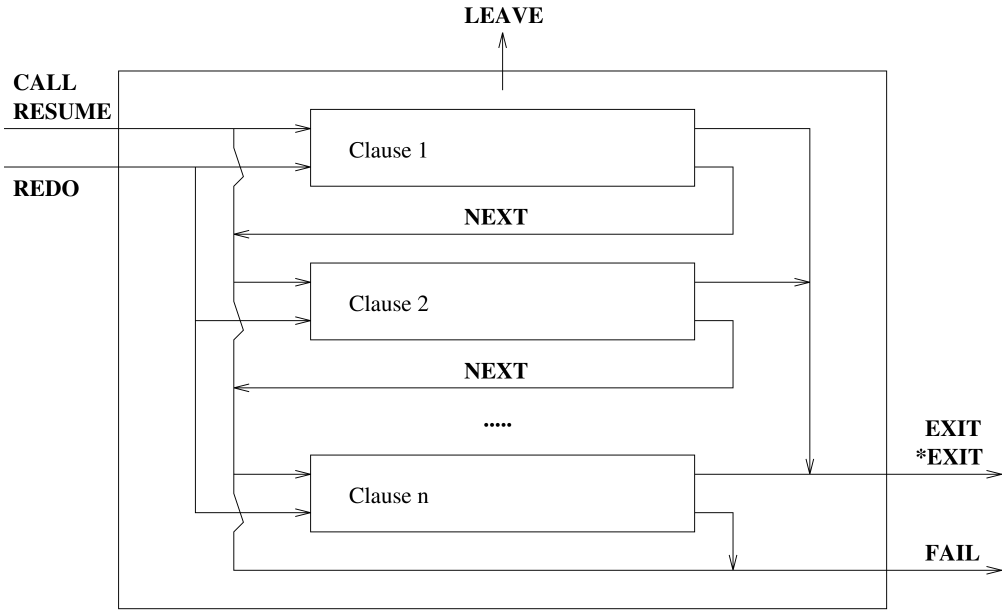


Figure 15.1: The box model

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 exceptions. Exceptions are either raised implicitly by built-in predicates (in which case the built-in itself exits via the LEAVE port), or explicitly through a call to **throw/1** (or **exit_block/1**). All ancestors of the predicate that raised the exception will subsequently exit via a LEAVE port, until a **catch/3** (or **block/3**) is found, whose second argument matches the exception. This invocation of **catch/3** then passes a NEXT port (at which point the exception has been caught), and then execution continues via a normal call of the recovery goal (the third argument of the **catch/3**).

As with the FAIL port, no argument values 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.

15.1.1 Breakpoints

Breakpoints can be set on specific calls to a predicate, i.e., on a specific body goal in the source, so that the debugger will stop only at a CALL port only when that specific body goal is executed. A breakpoint is specify by giving the source file and the line number where the body goal is. For example, if the following predicate is in a file called `newtop`, with the following line numbers:

```
243  check_words([],[]).
244  check_words([Word|Words],[RevWord|RevWords]) :-
245      check_words(Words,RevWords).
```

The breakpoint for the body goal `check_words(Words, RevWords)` would be `newtop:245`. Note that the file name must be sufficiently specified for ECLiPSe to find the file from your current working directory.

For a call that has a breakpoint set, the execution will stop when the call is made, i.e., at the CALL port for that specific body goal.

15.2 Format of the Tracing Messages

All trace messages are output to the `debug_output` stream.
The format of one trace line is as follows:

```
S+(4) 2 *EXIT<5> module:foo(one, X, two)   %>
12 3  4 5 6   7    8       9              10
```

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, and a `#` shows that the specific call has a break-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 an exception was raised by either a built-in predicate error condition or a call to **throw/1** or **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. This only appears if the goal is executing at a different priority than 12, the normal priority. The number between the angled brackets shows the priority (between 1 and 11) that the goal is executed at.

8. For the tty debugger, 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. With tkeclipse, the module name is not displayed with the traceline, instead, you can get the information by right holding the mouse button over the trace line in the call stack window.

9. The goal is printed according to the current instantiations of its variables. Arguments of the form `...` represent subterms 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.

10. The prompt of the debugger, which means that it is waiting for a command from the user. Note there is no prompt when tkeclipse tracer is used.

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 to 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 **skipped/1**, **unskipped/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 textual 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 leap mode.
For the TkECL^{i}PS^e graphical toplevel, the debugger may be switched on by starting the tracer from the Tools menu before executing the query. This puts the debugger in `creep` mode.
When the debugger is in `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, or a breakpoint on a specific call, 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 need not 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.
```

In tkeclipse, the debugger can also be started in this way. The tracer tool will popup at the appropriate predicate if it has not been invoked already. The `start_tracing` flag can also be set with the predicate browser tool.

[1]Provided the call has been compiled in debug_compile mode, or the call is a meta-call.

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 ancestors, 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`, i.e., `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.
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 tracing off permanently.

q **query the failure culprit**
The purpose of this command is to find out why a goal has failed (FAIL) or was aborted with an exception (LEAVE). It prints the invocation number of the goal which caused the failure. You can then re-run the program in `creep` mode and type `q` at the first command prompt. This will then offer you to jump to the CALL port of the culprit goal.

```
[eclipse 3]: p.
  (1) 1 CALL  p   %> skip
  (1) 1 FAIL  p   %> query culprit
failure culprit was (3) - rerun and type q to jump there   %> nodebug? [y]
No (0.00s cpu)

[eclipse 4]: p.
  (1) 1 CALL  p   %> query culprit
failure culprit was (3) - jump to invoc: [3]?
  (3) 3 CALL  r(1)   %> creep
  (3) 3 FAIL  r(...)   %> creep
  (2) 2 FAIL  q   %> creep
  (1) 1 FAIL  p   %> creep
No (0.01s cpu)
```

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 a port other than the specified one 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 display 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 display 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.

w **write source context for current goal**
Lists the source lines around the current goal displayed by the trace line, showing the context of the goal. For example:

```
  (230) 4 CALL  check_word(what, _5824)   %> write source lines
Source file: /homes/user/EclipseTests/Chatq/newtop
  241  :- mode check_words(+,-).
  242
  243  check_words([],[]).
  244  check_words([Word|Words],[RevWord|RevWords]) :-
  245>    check_word(Word,RevWord),
  245      check_words(Words,RevWords).
  246
  247  :- mode check_word(+,-).
  248
   %>
```

The listing shows the line numbers for the source lines, with a `>` marking the line with the current goal. Note it is the actual body goal that is shown, rather than the predicate definition as in the `.` command. An optional numeric argument can be given before the command, specifying the number of lines surrounding (i.e., before and after) the current goal that should be listed:

```
   %> 2write source lines
Source file: /homes/user/EclipseTests/Chatq/newtop
  243  check_words([],[]).
  244  check_words([Word|Words],[RevWord|RevWords]) :-
  245>     check_word(Word,RevWord),
  245      check_words(Words,RevWords).
  246
   %>
```

Source is only shown if the source information is available—that is, the code has to be compiled debuggable from a file, and not all goals have source information; for example, goals in meta-calls (e.g., those inside a **call/1**). Also, source context cannot be shown at a RESUME port.

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. 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:

```
  (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 5.1). 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 notation0) 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 must 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 5.1). 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 **capital/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` (i.e., `CTRL-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`, setting it to `nodebug`.

15.7 Extending the Debugger

15.7.1 User-defined Ports

The standard set of ports in the debugger's box model can be extended by the programmer. This facility is not so much intended for applications, but rather for libraries that want to allow debugging in terms of concepts of the library. Specific ports can be used to identify the interesting events during execution of the library code (while the standard tracing of the library internals can be suppressed by compiling the library in nodebug-mode).

The system provides 4 primitives that can generate 4 kinds of box model ports. When inserted into 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:

- **trace_call_port(+*Port*, ?*Invoc*, ?*Term*)** is used to create new ports similar to CALL ports, but the port name can be chosen freely. Such a port creates a new box. There must be a corresponding **trace_exit_port/0** to exit the box on success.

- **trace_exit_port** is used in conjunction with **trace_call_port/3** to exit a box on success.

- **trace_point_port(+*Port*, ?*Invoc*, ?*Term*)** is used to create a standalone port, i.e., a port that causes the tracer to create a trace line, but does not create, enter or leave any box.

- **trace_parent_port(+*Port*)** is used to create an additional port for the parent box, but does not enter or leave the box.

For example, **trace_call_port/3** and **trace_exit_port/0** can be used to create a more readable trace in the presence of source transformations. Imagine that the goal `Y is X*X-1` has been flattened into the goal sequence `*(X,X,T),-(T,1,Y)`. By inserting the trace primitives the debugger can still show the original source before transformation:

```
p(X,Y) :-
        trace_call_port(call,_, Y is X*X-1),
        *(X,X,T),
        -(T,1,Y),
        trace_exit_port.
```

The trace then looks like this:

```
[eclipse 8]: p(3,Y).
(1) 1 CALL  p(3, Y)   %> creep
(2) 2 CALL  Y is 3 * 3 - 1   %> skip
(2) 2 EXIT  8 is 3 * 3 - 1   %> creep
(1) 1 EXIT  p(3, 8)   %> creep
Y = 8
```

Another example is the insertion of additional ports for existing boxes, in particular the current parent box:

```
p :-
        trace_parent_port(clause1),
        writeln(hello),
        fail.
p :-
        trace_parent_port(clause2),
        writeln(world).
```

This gives rise to the following trace:

```
?- p.
  (1) 1 CALL  p   %> creep
```

```
  (1) 1 CLAUSE1  p   %> creep
S (2) 2 CALL  writeln(hello)   %> creep
hello
S (2) 2 EXIT  writeln(hello)   %> creep
  (3) 2 CALL  fail   %> creep
  (3) 2 FAIL  fail   %> creep
  (1) 1 NEXT  p   %> creep
  (1) 1 CLAUSE2  p   %> creep
S (4) 2 CALL  writeln(world)   %> creep
world
S (4) 2 EXIT  writeln(world)   %> creep
  (1) 1 EXIT  p   %> creep
Yes (0.00s cpu)
```

Note that the additional ports share the parent's invocation number, so the `i` command can be used to skip from one to the other.

15.7.2 Attaching a Different User Interface

The tracer consists of a **trace generation** component (which is part of the ECL^{i}PS^e runtime kernel), and a **user interface** (which is part of the development system). The standard ECL^{i}PS^e distribution contains two user interfaces, a console-based one, and a graphical one which is part of TkECL^{i}PS^e. A programmable tracer interface (OPIUM/LSD) is under development in the group of Mireille Ducasse at IRISA/Rennes. Connecting new interfaces is relatively easy, for more detailed information contact the ECL^{i}PS^e development team.

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

Development Support Tools

This chapter describes some of the tools and libraries provided by ECL^{i}PS^e that assist in program development and the analysis of program runtime behaviour.

16.1 Available Tools and Libraries

ECL^{i}PS^e provides a number of different tools and libraries to assist the programmer with program development:

Document Tools for generating documentation from ECLiPSe sources.

Lint Generates warning messages for dubious programming constructs and violation of naming conventions for an ECLiPSe source module or file.

Pretty_printer Tools for pretty-printing a file in different formats.

Xref Enables the analysis of an ECLiPSe source module or file for the construction of a predicate call graph.

In addition, ECL^{i}PS^e provides several tools that aid in the understanding of a programs runtime behaviour:

Coverage Records the frequency at which various parts of the program are executed.

Debugger Provides a low level view of program activity. Chapter 15 presents a comprehensive look at debugging of ECL^{i}PS^e programs.

Display matrix Shows the values of given terms in a graphical window. Chapter 4 discusses the use of this tool.

Mode Analyser Collects statistics about the invocation modes of predicates within a running program in order to assist in the generation of compiler invocation mode directives.

Port Profiler Collects statistics about the running program in terms of box model port counters.

Timing Profiler Samples the running program at regular intervals to give a statistical summary of where the execution time is spent.

Visualisation framework A graphical environment for the visualisation of search and propagation in constraint programs. The *Visualisation Tools Manual* discusses the use of this environment.

This section focuses on the program development libraries and two complementary runtime analysis tools, the **profiler** and the **coverage** library. Throughout this chapter, the use of each of the tools is demonstrated on the following *n-queens* code:

```
:- module(queen).
:- export queen/2.

queen(Data, Out) :-
        qperm(Data, Out),
        safe(Out).

qperm([], []).
qperm([X|Y], [U|V]) :-
        qdelete(U, X, Y, Z),
        qperm(Z, V).

qdelete(A, A, L, L).
qdelete(X, A, [H|T], [A|R]) :-
        qdelete(X, H, T, R).

safe([]).
safe([N|L]) :-
        nodiag(L, N, 1),
        safe(L).

nodiag([], _, _).
nodiag([N|L], B, D) :-
        D =\= N - B,
        D =\= B - N,
        D1 is D + 1,
        nodiag(L, B, D1).
```

16.2 Heuristic Program Checker

The Heuristic Program Checking tool generates warning messages for dubious programming constructs and violation of naming conventions for an ECLiPSe source module or file. It is loaded as follows:

```
:- lib(lint).
```

The heuristic rules currently enforced are based on the style guide of Appendix E. These rules are somewhat limited in scope. The library is distributed as source and serves to provide a framework for the addition of a more comprehensive set of rules that are tailored to each individual developer.
Consider the following typographic mistakes in the *n-queens* example:

```
queen(Data, Out) :-
        qperm(Datas, Out),
        safe(Out).

n0diag([], _, _).
```

The tool is invoked using the **lint/1** predicate with the source file specified as an atom or string:

```
?- lint(queen).

--- File /tmp/queen.ecl, line 4:
Singleton variables: [Data, Datas]

--- File /tmp/queen.ecl, line 22:
Questionable predicate name: n0diag

Yes (0.01s cpu)
```

The checker identifies `Data` and `Datas` as being singleton variables and is dubious of the `n0diag` predicate name. Both are the result of programmer error, `Datas` should read `Data` and `n0diag` as `nodiag`. The **lint/2** predicate allows a list of options to be specified that turn on and off the heuristic rules.

16.3 Document Generation Tools

The document generation tools library provides a set of predicates for the generation of documentation from ECL^{i}PS^e program sources. The tools generate documentation by processing the **comment/2** directives in each source file. The following is an example comment for the *n-queens* example:

```
% comment for queen/2
:- comment(queen/2, [

    summary: "Program that solves the attacking Queens problem for
              an arbitrary number of queens.",

    index: ["NQueens Problem"],

    args: ["Data": "List modelling initial state of queens on board.",
       "Args": "Solution list of Y-coordinate of each queen on the
        board."],

    amode: queen(+,-),
    amode: queen(-,+),
    amode: queen(+,+),

    resat: yes,
```

```
 fail_if: "A solution cannot be found where all queens are safe
           from attack by every other.",

 see_also:
     [queens8/1, queensN/1],

 desc: html("The problem is to arrange a specified number of queens
        on a chessboard such that no queen attacks any other queen
        The predicate takes a list representing the initial state
        of the queens on the board, with each element representing
        a queen and its current Y-coordinate. If a solution is
        found, a list is returned specifying the safe Y-coordinate
for each queen.")
]).  % end of comment directive for queen/2
```

There are two pertinent predicates for document generation. The first, **icompile/2** generates an information file (with the extension `.eci`) by extracting information from a source file (whose extension is `.ecl`). The second, **eci_to_html/3**, processes this information file to produce readable HTML and plain text files. By default, these files are placed in a subdirectory with the same name as the information file, but without the extension. The generated files are `index.html`, containing a summary description of the library, plus one HTML and one plain text file for each predicate that was commented using a **comment/2** directive in the source.
The following produces the `queen.eci` file and a `queen` directory in the current directory. Within the queen directory reside `index.html`, `queen-2.html` and `queen-2.txt`:

```
?- lib(document).
document.ecl compiled traceable 83620 bytes in 0.04 seconds
Yes (0.04s cpu)

?- icompile(queen, ".").
queen.ecl  compiled traceable 1432 bytes in 0.01 seconds
/examples/queen.eci generated in 0.00 seconds.
Yes (0.01s cpu)

?- eci_to_html(queen, ".", "").
Yes (0.00s cpu)
```

16.4 Cross Referencing Tool

The cross referencing library **xref** analyses an ECLiPSe source module or file and builds its predicate call graph. The graph can either be returned in the format of `lib(graph_algorithms)`, as text, or as a graphical display.
The **xref/2** predicate is invoked as `xref(File, Options)`. This generates a call graph for the file *File* according to the *Options* list. The options specify the format of the graph to be generated, whether calls to built in predicates are displayed and whether it is a caller or callee graph:

```
?- xref:xref(queen, []).
```

```
nodiag / 3 calls:
        nodiag / 3

qdelete / 4 calls:
        qdelete / 4

qperm / 2 calls:
        qdelete / 4
        qperm / 2

queen / 2 calls:
        qperm / 2
        safe / 1

safe / 1 calls:
        nodiag / 3
        safe / 1

Yes (0.01s cpu)
?- xref:xref(queen,[builtins:on,output:daVinci]).
WARNING: module 'daVinci' does not exist, loading library...
daVinci.ecl compiled traceable 5644 bytes in 0.01 seconds
```

The first `xref` predicate call generates a textual call graph for the `queen` module, while the second generates the `daVinci` graph illustrated in figure 16.1.

16.5 Pretty Printer Tool

The pretty printer library provides a set of predicates for the printing of a file's contents as a file in a number of formats. In particular, an ECL^{i}PS^e source file can be converted into an HTML document with proper indentation, syntax colouring, hyperlinks from predicate uses to definition, and hyperlinks to documentation.
The **pretty_print/2** predicate is used to print the file, or list of files. A list of options can be given which modifies the format of the output file, its location, etc. The following creates subdirectory `pretty` in the current directory. Within the `pretty` directory reside `index.html` and `queen.html`, where `queen.html` is the `queen` module pretty printed in HTML:

```
?- pretty_printer:pretty_print(queen).
Writing /examples/pretty/queen.html
```

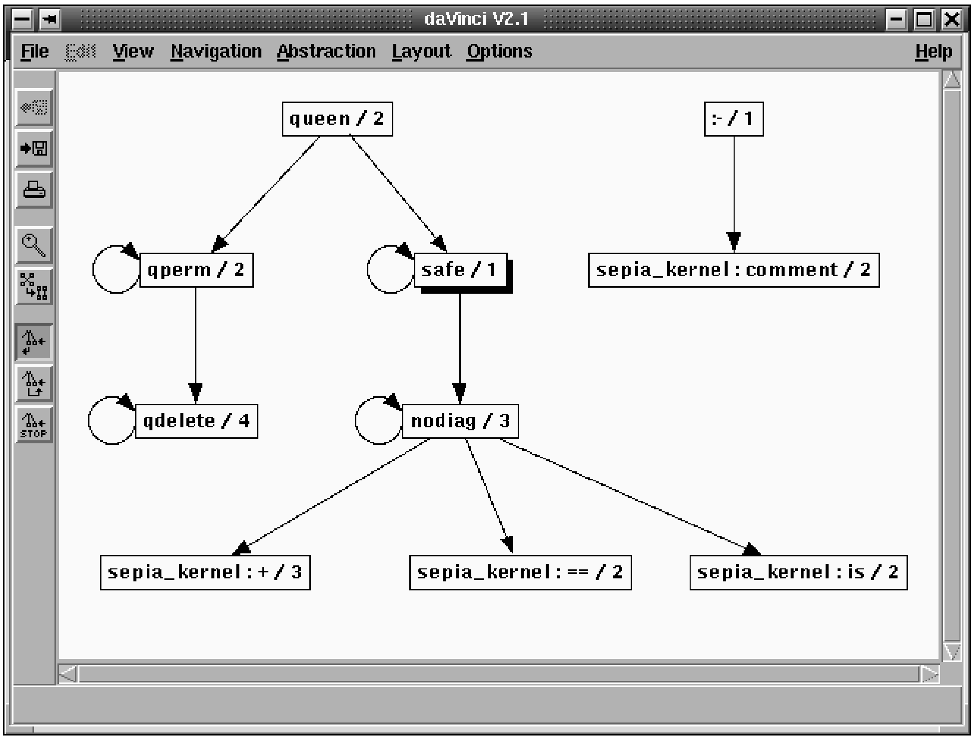


Figure 16.1: Call graph for queen example with built-in predicates

16.6 Timing Profiler

The profiling tool helps to find *hot spots* in a program that are worth optimising. To use the profiler, start ECL^{i}PS^e or TkECL^{i}PS^e with the `-P` command line option (or use the EC_OPTION_WITH_PROFILE via the C interface, or the with_profiler option via the Tcl interface, see D). It is not necessary to compile the profiled code in a special way; this profiler works independently of compiler optimizations and debug mode.
To profile the execution of a particular goal, use **profile/1**:

```
?- profile(Goal).
```

or select the *Time Profile* option from TkECL^{i}PS^e's Query-menu. The profiler then executes the *Goal* in the profiling mode, which means that every 100th of a second the execution is interrupted and the profiler records the currently executing procedure. For example

```
?- profile(queen([1,2,3,4,5,6,7,8,9],Out)).
goal succeeded

  PROFILING STATISTICS
  --------------------
Goal:             queen([1, 2, 3, 4, 5, 6, 7, 8, 9], Out)
Result:   success
Sampling rate:   every 0.01s process_cputime
Samples taken:   2
```

```
Thread cputime:   0.03s

Predicate             Module        %Time   Time   %Cum
--------------------------------------------------------
qdelete          /4   eclipse       50.0%   0.01s  50.0%
nodiag           /3   eclipse       50.0%   0.01s 100.0%

Out = [1, 3, 6, 8, 2, 4, 9, 7, 5]
Yes (0.14s cpu)
```

The profiler output contains the following information:

1. The line `Goal:` shows the goal which was profiled.
2. The line `Result:` indicates whether the specified goal *succeeded*, *failed* or *aborted*. The **profile/1** predicate itself always succeeds.
3. The next lines show the sampling rate and the number of samples taken.
4. The next line shows the time spent in the calling thread.
5. Finally the predicates which were being executed when the profiler sampled, ranked in decreasing sample count order are shown.

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 a single 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.
The problem with the results displayed above is that the sampling frequency is too low when compared to the total user time spent executing the goal. In fact in the above example the profiler was only able to take two samples before the goal terminated.
The frequency at which the profiler samples is fixed, so in order to obtain more representative results one should have an auxiliary predicate which calls the goal a number of times, and compile and profile a call to this auxiliary predicate. e.g.,

```
queen_100 :-
  (for(_,1,100,1) do queen([1,2,3,4,5,6,7,8,9],_Out)).
```

Note that, when compiled, the above **do/2** loop would be efficiently implemented and not cause overhead that would distort the measurement. Section 5.2 presents a detailed description of logical loops.

```
?- profile(queen_100).
goal succeeded

  PROFILING STATISTICS
  --------------------
Goal:             queen_100
Result:   success
Sampling rate:   every 0.01s process_cputime
Samples taken:   319
Thread cputime:   3.19

Predicate             Module       %Time   Time   %Cum
--------------------------------------------------------
nodiag          /3  eclipse       52.2%   1.67s  52.2%
qdelete         /4  eclipse       27.4%   0.87s  79.6%
qperm           /2  eclipse       17.0%   0.54s  96.5%
safe            /1  eclipse        2.8%   0.09s  99.4%
queen           /2  eclipse        0.6%   0.02s 100.0%

Yes (3.33s cpu)
```

In the above example, the profiler takes over three hundred samples resulting in a more accurate view of where the time is being spent in the program. In this instance we can see that more than half of the time is spent in the **nodiag/3** predicate, making it an ideal candidate for optimisation. This is left as an exercise for the reader.
Limitation: in ECL^{i}PS^{e}7.0, only the engine that called profile/1 is profiled.

16.7 Port Profiler

The port profiler is a performance analysis tool based on the idea of counting of events during program execution. The events that are counted are defined in terms of the "box model" of execution (the same model that the debugger uses, see chapter 15.1). In this box model, predicates are entered though *call*, *redo* or *resume* ports, and exited through *exit*, **exit*, *fail* or *leave* ports. Some other interesting events are indicated by ports as well (*next*, *else*, *delay*).
The usage is as follows:

1. Compile your program in debug mode, as you would normally do during program development.
2. Load the **port_profiler** library
3. Run the query which you want to examine, using **port_profile/2**:

   ```
   ?- port_profile(queen([1,2,3,4],Out), []).
   ```

This will print the results in a table like the following:

PREDICATE		CALLER		call	exit	fail	*exit	redo
-	/3	nodiag	/3	46	46	.	.	.
=\=	/2	nodiag	/3	46	45	1	.	.
qperm	/2	qperm	/2	30	28	.	16	14
qdelete	/4	qperm	/2	20	18	.	12	10
nodiag	/3	nodiag	/3	17	14	3	.	.
nodiag	/3	safe	/1	17	7	10	.	.
+	/3	nodiag	/3	17	17	.	.	.
qdelete	/4	qdelete	/4	10	9	.	3	2
qperm	/2	queen	/2	1	.	.	11	10
safe	/1	queen	/2	11	1	10	.	.
safe	/1	safe	/1	7	4	3	.	.
queen	/2	trace_body	/2	1	.	.	1	.

Each row of the table shows the information for a particular predicate (by default split according to different caller predicates). The table is sorted according to entry port count ($call + redo + resume$). The port counts give information about:

- what are the most frequently called predicates (`call` ports);
- whether predicates failed unexpectedly (`fail` ports);
- whether predicates exited nondeterministically (`*exit` ports), i.e., whether they left behind any choice-points for backtracking;
- whether nondeterministically exited predicates were ever re-entered to find alternative solutions (`redo` ports);
- whether predicates did internal backtracking (`next` ports) in order to find the right clause (this may indicate suboptimal indexing);
- how often predicates were delayed and resumed.

For more details about different options and output formats, see the description of **port_profiler** in the *Reference Manual*.

16.8 Line coverage

The line coverage library provides a means to ascertain exactly how many times individual clauses are called during the evaluation of a query.
The library works by placing *coverage counters* at strategic points throughout the code being analysed. These counters are incremented each time the evaluation of a query passes them. There are three locations in which coverage counters can be inserted.

1. At the beginning of a code block.
2. Between predicate calls within a code block.
3. At the end of a code block.

A code block is defined to be a conjunction of predicate calls, i.e., a sequence of goals separated by commas.
The counter values do not only show whether all code points were reached but also whether subgoals failed or aborted (in which case the counter before a subgoal will have a higher value than the counter after it).

16.8.1 Compilation

In order to add the coverage counters to code, it must be compiled with the **ccompile/1** predicate which can be found in the **coverage** library.
The **ccompile/1** predicate (note the initial 'c' stands for coverage) can be used in place of the normal **compile/1** predicate to compile a file with coverage counters.
The following shows the results of compiling the *n-queens* example:

```
?- coverage:ccompile(queen).
queen.ecl  compiled traceable 6016 bytes in 0.01 seconds
coverage: inserted 20 coverage counters into module queen

Yes (0.14s cpu)
```

Once compiled, predicates can be called as usual and will (by default) have no visible side effects. Internally however, the counters will be incremented as the execution progresses. The following demonstrates this for a single solution to the **queen/2** predicate:

```
?- queen:queen([1,2,3,4,5,6,7,8,9], Out).
```

The counter results are retrieved as demonstrated in the subsequent section. The two argument predicate **ccompile/2** can take a list of `name:value` pairs which can be used to control the exact manner in which coverage counters are inserted. The documentation for the **ccompile/2** predicate provides for a full list of the available flags.

16.8.2 Results

To generate an HTML file containing the coverage counter results, the **result/1** predicate is used:

```
?- coverage:result(queen).
Writing /examples/coverage/queen.html
index.pl   compiled traceable 335304 bytes in 0.17 seconds

Yes (0.18s cpu)
```

This creates the result file `coverage/queens.html` which can be viewed using any browser. It contains a pretty-printed form of the source, annotated with the values of the code coverage counters as described above. As a side effect, the coverage counters will be reset.

16.9 Mode analysis

The **mode_analyser** library is a tool that assists in the generation of the **mode/1** directive for predicate definitions. This directive informs the compiler that the arguments of the specified predicate will always have the corresponding form when the predicate is called. The compiler utilises this information during compilation of the predicate in order to generate more compact and/or faster code. Specifying the mode of a predicate that has already been compiled has no effect, unless it is recompiled. If the specified procedure does not exist, a local undefined procedure is created.

The mode analyser inserts instrumentation into the clause definitions of predicates during compilation in order to record mode usage of each predicate argument. The code should then be run (as many times as is necessary to capture the most common invocations of each predicate undergoing analysis). Finally, the results of the analysis are requested and the suggested mode annotations for each predicate are displayed.

The usage is as follows:

1. Load the **mode_analyser** library:

   ```
   ?- lib(mode_analyser).
   ```

2. Compile your program with the mode analyser:

   ```
   ?- analyse(queen).
   ```

3. Run the query which most accurately exercises the invocation modes of the defined predicates:

   ```
   ?- queen:queen([1,2,3,4],Out).
   ```

4. Generate the results for the module into which the program was compiled:

   ```
   ?- result([verbose:on])@queen.
   ```

This will print the results as follows:

```
Mode analysis for queen : qdelete / 4:
        Results for argument 1:
                -: 23   *: 0    +: 0    ++: 0
        Results for argument 2:
                -: 0    *: 0    +: 0    ++: 23
        Results for argument 3:
                -: 0    *: 0    +: 0    ++: 23
        Results for argument 4:
                -: 0    *: 0    +: 23   ++: 0

        qdelete(-, ++, ++, +)

Mode analysis for queen : nodiag / 3:
        Results for argument 1:
                -: 0    *: 0    +: 0    ++: 62
        Results for argument 2:
                -: 0    *: 0    +: 0    ++: 62
        Results for argument 3:
                -: 0    *: 0    +: 0    ++: 62

        nodiag(++, ++, ++)
```

```
Mode analysis for queen : qperm / 2:
        Results for argument 1:
                -: 0    *: 0    +: 0    ++: 41
        Results for argument 2:
                -: 0    *: 0    +: 41   ++: 0

        qperm(++, +)

Mode analysis for queen : queen / 2:
        Results for argument 1:
                -: 0    *: 0    +: 0    ++: 1
        Results for argument 2:
                -: 1    *: 0    +: 0    ++: 0

        queen(++, -)

Mode analysis for queen : safe / 1:
        Results for argument 1:
                -: 0    *: 0    +: 0    ++: 38

        safe(++)
```

NOTE: It is imperative to understand that the results of mode analysis are merely suggestions for the invocation modes of a predicate based on runtime information. If there are potential predicate invocation modes that were not exercised during runtime, the tool is unable to account for them in its analysis. For the mode specifier '-' the mode analyser does not determine whether the variable occurs in any other argument (i.e., is aliased), this must be manually verified. In summary, the programmer must verify that the suggested modes are correct before using the directive in the code. If the instantiation of the predicate call violates its mode declaration, no exception is raised and its behaviour is undefined.
For more details about invocation mode analysis see the description of **mode_analyser** in the *Reference Manual*.

Chapter 17

Attributed Variables

17.1 Introduction

The **attributed variable** is a special ECL^{i}PS^{e}data type which represents a variable together with attached attributes. In the literature, attributed variables are sometimes referred to as "metaterms". The name *metaterm* originates from its application in meta-programming: for an object-level program, a metaterm looks like a variable, but for a meta-program the same variable is just a piece of data which, possibly together with additional meta-level information, forms the metaterm.
The attributed variable 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 in comparison 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*.

17.2 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 17.7).

17.3 Syntax

The most general attributed variable syntax is

Var{Name_1:Attr_1, Name_2:Attr_2, ..., 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 variable's name, e.g.,

```
p(X, X{attr:Attr})
```

or

```
p(X{attr:Attr}, X)
```

both describe the same term, which has two occurrences of a single attributed variable with attribute `attr:Attr`. The following is a syntax error (even when the attributes are identical):

```
p(X{attr:Attr}, X{attr:Attr})
```

17.4 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*.

17.5 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 whose name is the instantiation of *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 17.7).

17.6 Attribute Modification

Often an extension must 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(*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** 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.

17.7 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** 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* [, ?*SuspAttr*])**

The first argument is the term that was unified with the attributed variable, it is either a non-variable or another attributed variable. The second argument is the contents of the attribute slot corresponding to the extension. Note that, at this point in execution, the orginal attributed variable no longer exists, because it has already been bound to *Term*. The optional third argument is the suspend-attribute of the former variable; it may be needed to wake the variable's 'constrained' suspension list.

The handler's job is to determine whether the binding is allowed with respect to the attribute. This could for example involve checking whether the bound term is in a domain described by the attribute. For variable-variable bindings, typically the remaining attribute must be updated to reflect the intersection of the two individual attributes. In case of success, suspension lists inside the attributes may need to be scheduled for waking.

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 simultaneously, e.g. during a head unification or during the unification of two compound terms. The handlers are only invoked at certain trigger points (usually before the next regular predicate call). Woken goals will start executing once all unify-handlers are done.

test_unify: a unifiability test which is not supposed to trigger constraints propagation. It is used by 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. The handler's job is to determine whether *Attribute* allows unification with *Term* (not considering effects of woken goals). During the execution of the handler, the attributed variable may be bound to *Term*, however when all attribute handlers succeed, all bindings are undone again, and no waking occurs.

compare_instances: computation of instance, subsumption and variance relationship, as performed by the built-ins **compare_instances/3**, **instance/2** and **variant/2**. The handler procedure is

> **instance_handler(-*Res*, ?*TermL*, ?*TermR*)**

and its arguments are similar to the ones of the **compare_instances/3** predicate. The handler is invoked with one or both of *TermL* and *TermR* being attributed variables. The task of the handler is to examine the two terms, and compute their instance relationship with respect to the extension attribute in question. The handler must bind *Res* to `=` iff the terms are variants, `<` iff *TermL* is a proper instance of TermR, or `>` iff *TermR* is a proper instance of *TermL*) with respect to the attribute under consideration. If the terms are not unifiable with respect to this attribute, the handler must fail.

Even though one of *TermL* and *TermR* is guaranteed to be an attributed variable, they might not have the particular attribute that the handler is concerned with. The handler must therefore be written to correctly deal with all combinations of an attributed (but potentially uninstantiated attribute) variable with any other term.

copy_term: the handler is invoked by either **copy_term/2** or **copy_term_vars/3**. The handler procedure is

copy_handler(?*AttrVar*, ?*Copy*)

AttrVar is the attributed variable encountered in the copied term, *Copy* 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 *Copy* is still a free variable or if it was already bound to an attributed variable by a previous handler.

suspensions: this handler is invoked by the **suspensions/2** predicate to collect all the suspension lists inside the attribute. The handler call pattern is

suspensions_handler(?*AttrVar*, -*ListOfSuspLists*, -*Tail*)

AttrVar is an attributed variable. The handler should bind *ListOfSuspLists* to a list containing all the attribute's suspension lists and ending with *Tail*.

delayed_goals_number: handler is invoked by the **delayed_goals_number/2** predicate. The handler call pattern is

delayed_goals_number_handler(?*AttrVar*, -*Number*)

AttrVar 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.

get_bounds: This handler is used by the predicate **get_var_bounds/3** to retrieve information about the lower and upper bound of a numeric variable. The handler should therefore only be defined if the attribute contains that kind of information. The handler call pattern is

get_bounds_handler(?*AttrVar*, -*Lwb*, -*Upb*)

The handler is only invoked if the variable has the corresponding (non-empty) attribute. The handler should bind *Lwb* and *Upb* to numbers (any numeric type) reflecting the attribute's information about lower and upper bound of the variable, respectively. If different attributes return different bounds information, **get_var_bounds/3** will return the intersection of these bounds. This can be empty ($Lwb > Upb$).

set_bounds: This handler is used by the predicate **set_var_bounds/3** to distribute information about the lower and upper bound of a numeric variable to all its existing attributes. The handler should therefore only be defined if the attribute can incorporate this kind of information. The handler call pattern is

set_bounds_handler(?*AttrVar*, +*Lwb*, +*Upb*)

The handler is only invoked if the variable has the corresponding (non-empty) attribute. *Lwb* and *Upb* are the numbers that were passed to **set_var_bounds/3**, and the handler is expected to update its own bounds representation accordingly.

print: attribute printing in **write/1,2**, **writeln/1,2**, **printf/2,3** when the `m` option is specified. The handler procedure is

print_handler(?*AttrVar*, -*PrintAttr*)

AttrVar is the attributed variable being printed, *PrintAttr* 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, or the print handler fails, the attribute will not be printed.

The following handlers are still supported for compatibility, but their use is not recommened:

pre_unify: this is another handler which can be invoked on normal unification, but it is called *before* the unification itself occurs. The handler procedure is

pre_unify_handler(?*AttrVar*, +*Term* [, -*Goals*])

The first argument is the attributed variable to be unfied, the second argument is the term it is going to be unified with. The optional third argument can be used to return goals that will be called after all pre-unify handlers for this variable have finished, and the variable has been bound. The handlers itself should not bind any variables. If multiple attributed variables were bound in a single unification, all these bindings are first undone, then the handlers are called and the variables re-bound one by one. This handler is provided for compatibility with SICStus Prolog and its use is not recommended. It can be used together with a `unify` handler, which is called afterwards.

delayed_goals: this handler is superseded by the suspensions-handler, which should be preferred. If there is no suspensions- handler, this handler is invoked by the obsolete **delayed_goals/2** predicate. The handler procedure is

delayed_goals_handler(?*AttrVar*, ?*GoalList*, -*GoalCont*)

AttrVar 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.

17.7.1 Printing Attributed Variables

The different output predicates treat attributed variables differently. The **write/1** predicate prints the attributes using the print-handlers, while **writeq/1** prints the whole attribute, 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}.

A simple print handler can just return the attribute literally, like

```
print_attr(_{Attr}, PrintAttr) ?- PrintAttr=Attr.
```

An attributed variable X{m:a} with print handler **print_attr/2** for the m-attribute, can thus be printed in different ways, e.g., [1]

```
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.

17.8 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.

17.9 Examples of Using Attributed Variables

17.9.1 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 (second clause of **unify_enum/2**).

[1]The attribute suspend is always present and defined by system coroutining.

```
[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.

17.10 Attribute Specification

The structures notation (see section 5.1) 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.,

```
suspend{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.

Chapter 18

Advanced Control Features

18.1 Introduction

This chapter introduces the control facilities that distinguish the ECL^{i}PS^e language from Prolog by providing a computation rule that is more flexible than simple left-to-right goal selection. The core feature is the ability to suspend the execution of a goal at some point during execution, and resume it under certain conditions at a later stage. Together with attributed variables, these facilities are the prerequisites for the implementation of constraint propagation and similar data-driven algorithms.

18.2 Concepts

18.2.1 The Structured Resolvent

The term **resolvent** originates from Logic Programming. It is the set of all goals that must be satisfied. The computation typically starts with a resolvent consisting only of the top-level goal (the initial query). This then gets successively transformed (by substituting goals that match a clause head with an instance of the clause body, i.e., 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.    % clause 1
q :- true.    % clause 2
r :- q.       % clause 3
```

and the goal p, the resolvent goes through the following states before the goal is proven (by reduction to true) and the computation terminates:

```
p --1--> (q,r) --2--> (true,r) ----> (r) --3--> (q) --2--> true
```

While in Prolog the resolvent is always processed from left to right like in this example, 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.

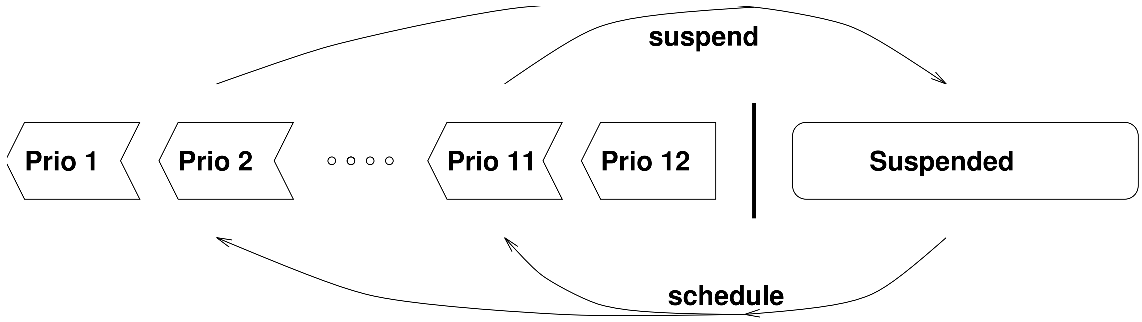


Figure 18.1: Structure of the resolvent

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 18.1 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.

18.2.2 Floundering

The case that a subgoal remains suspended (delayed) at the end of the computation is sometimes referred to as **floundering**. When floundering occurs, it means that the resolvent could not be reduced to true or fail, and that the answer bindings that have been found are valid only under the assumption that the remaining delayed goals are in fact true. Since such a conditional answer is normally not satisfactory (even though it may be correct), it is then necessary to change the control aspect of the program. The solution would usually be to either make further variable instantiations or to change control annotations. The aim is to get the delayed goals out of the suspended state and into the scheduled state, where they will eventually be executed and reduced. As a rule of thumb, goals will not suspend when all their arguments are fully instantiated. Therefore, a program that makes sure that all its variables are instantiated at the end of computation will typically not suffer from floundering.

18.3 Suspending Built-Ins and the Suspend-Library

Basic ECL^{i}PS^e has two built-in predicates whose behaviour includes suspending: the sound negation built-in **~/1** and the sound disequality predicate **~=/2**. Instead of succeeding or failing, they will suspend when their arguments are insufficiently instantiated to make a decision. For example

```
?- X ~= 3.
X = X
There is 1 delayed goal.
Yes (0.00s cpu)
```

Here, the system does not have enough information to decide whether the query is true or false. The goal remains delayed and we have a case of floundering (the ECL^{i}PS^e toplevel indicates this situation by printing a message about delayed goals at the end of the computation).

However, when the variable which was responsible for the suspension gets instantiated later, the delayed goal will be resumed (woken) and either succeed, fail, or suspend again. In the following example, the disequality predicate initially suspends, but wakes up later and succeeds or fails, respectively:

```
?- X ~= 3, X = 4.
X = 4
Yes (0.00s cpu)
?- X ~= 3, X = 3.
No (0.00s cpu)
```

Further predicate implementations with the same behaviour (delay until all arguments are ground) can be found in the **suspend** library **lib(suspend)**. In particular, it implements all common arithmetic predicates plus the constraints defined by the Common Arithmetic Solver Interface (see Constraint Library Manual), for instance

```
=:=/2, =\=/2,  >=/2,  =</2,  >/2,  </2,
 $=/2, $\=/2, $>=/2, $=</2, $>/2, $</2,
 #=/2, #\=/2, #>=/2, #=</2, #>/2, #</2,
integers/1, reals/1
```

The solver will suspend these predicates until all their arguments are ground.[1]
The suspend library is loaded into ECL^{i}PS^e on start-up, but the constraints associated with the suspend solver are not imported. To use them, either import the suspend library to the current module, or call the constraint qualified with the module:

```
suspend:(X > 2), suspend:(X #=< 5)
```

18.4 Development System Support

As seen in the above example, the top level loop indicates floundering by printing a message about delayed goals. The command line toplevel then prompts and offers to print a list of all delayed goals. The Tkeclipse development environment provides better support in the form of the Delayed Goals Viewer, which can be used to look at all delayed goals or a filtered subset of them.
The tracer supports advanced control features via the box-model ports DELAY and RESUME. It also shows goal priorities (if they deviate from the default priority) in angular brackets.

18.5 Declarative Suspension: Delay Clauses

For delaying calls to user-defined Prolog predicates, ECL^{i}PS^e provides several alternatives, the first being **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>.
```

[1] Note that more powerful versions of these constraints exist in other solvers such as the interval solver lib(ic).

A 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 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 goal with the delay clause head is not the usual Prolog unification, but rather unidirectional pattern matching (see also section 5.5). This means that the variables in the call cannot be bound by the matching, if such a binding would be necessary to perform the unification, it will fail instead. For example, the head of the delay clause

  ```
  delay p(a, X) if var(X).
  ```

  does not match the goal `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 [4].
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** (see **nonground/3**)
  - **\==/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. Experience shows that the above four primitives suffice to express most usual conditions.

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 **\==/2** predicate as a delaying condition is useful mainly in calls 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.

CAUTION: It may happen that the symbol `:-` is erroneously used instead of `if` in the delay clause. To indicate this error, the compiler complains about redefinition of the built-in predicate **delay/1**.

18.6 Explicit suspension with suspend/3

While delay-clauses are an elegant, declarative way of specifying how a program should execute, it is sometimes necessary to be more explicit about suspension and waking conditions. The built-in predicate **suspend/3** is provided for this purpose[2]. It allows one to explicitly create a suspended goal, specify its priority and its exact waking conditions.
When

suspend(*Goal*, *Prio*, *CondList*)

is called, *Goal* will be suspended with priority *Prio* and it will wake up as soon as one of the conditions specified in the *CondList* is satisfied. This list contains specifications of the form

[2]**suspend/3** is itself based on the lower-level primitives **make_suspension/3** and **insert_suspension/4**, which are described below.

```
Vars -> Cond
```

to denote that as soon as one of the variables in the term *Vars* satisfies 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 waking condition, e.g.,

```
[X,Y]->inst
```

means that as soon as one (or both) of the variables *X*, *Y* is 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 loaded library attributes. For example, when the interval solver library lib(ic) is loaded, either of

```
[A,B]->ic:min
[A,B]->ic:(min of ic)
```

triggers the suspended goal as soon as the minimum element of the domain of either *A* or *B* are updated (see Constraint Library Manual, IC Library).
Another admissible form of condition *Cond* is

```
trigger(Name)
```

which suspends the goal on the global trigger condition *Name* (see section 18.7.3).
Using **suspend/3**, we can rewrite our first delay-clause example from above as follows:

```
report_binding(X) :-
        ( var(X) ->
            suspend(report_binding(X), 0, X->inst)
        ;
            printf("Variable has been bound to %w\n", [X])
        ).
```

Here, when the predicate is called with an uninstantiated argument, we explicitly suspend a goal with the condition that it be woken as soon as X becomes instantiated. The priority is given as 0, which indicates the default priority (0 is not a valid priority itself). Running this code produces the following:

```
?- report_binding(X).
X = X
There is 1 delayed goal.
Yes (0.00s cpu)
```

When X is later instantiated, it will wake up and print the message:

```
?- report_binding(X), writeln(here), X = 99.
here
Variable has been bound to 99
X = 99
Yes (0.00s cpu)
```

18.7 Waking conditions

The usual purpose of suspending a goal is to wait and resume it later when more information about its arguments is available. In Logic Programming, this is usually the case when certain events related to variables occur. When such an event occurs, the suspended goal 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 suspended goal is executed.
The event which causes a suspended goal 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.

18.7.1 Standard Waking Conditions on Variables

There are three very general standard waking conditions which can be used with any variable. They are, in order of increasing generality:

inst: wake when a variable gets instantiated;

bound: wake when a variable gets instantiated or bound to another variable;

constrained: wake when a variable gets instantiated or bound to another variable or becomes otherwise constrained.

Each condition subsumes the preceding, more specific ones.

Waking on Instantiation: inst

To wake a goal when a variable gets instantiated, the `inst` condition is used. For example the following code suspends a goal until variable X is instantiated:

```
?- suspend(writeln(woken(X)), 0, X->inst).
X = X
There is 1 delayed goal.
Yes (0.00s cpu)
```

If this variable is later instantiated (bound to a non-variable), the goal executes in a data-driven way:

```
?- suspend(writeln(woken(X)), 0, X->inst), X = 99.
woken(99)
X = 99
Yes (0.00s cpu)
```

If we specify several instantiation conditions for the same goal, the goal will wake up as soon as the first of them occurs:

```
?- suspend(writeln(woken(X,Y)), 0, [X,Y]->inst), X = 99.
woken(99, Y)
X = 99
Y = Y
Yes (0.00s cpu)
```

It is not possible to specify a conjunction of conditions directly!
Let us now suppose we want to implement a predicate **succ/2**, such that `succ(X, Y)` is true when Y is the next integer after X. If we want the predicate to act as a lazy test, we must let it suspend until both variables are instantiated. This can be programmed as follows:

```
succ_lazy(X, Y) :-
        ( var(X) -> suspend(succ_lazy(X,Y), 0, X->inst)
        ; var(Y) -> suspend(succ_lazy(X,Y), 0, Y->inst)
        ; Y =:= X+1
        ).
```

The conjunctive condition "wait until X and Y are instantiated" is implemented by first waiting for X's instantiation, then waking up and re-suspending waiting for Y's instantiation.
A more eager implementation of **succ/2** would delay only until a single variable argument is left, and then compute the variable from the nonvariable argument:

```
succ_eager(X, Y) :-
        ( var(X) ->
            ( var(Y) ->
                suspend(succ_eager(X,Y), 0, [X,Y]->inst)
            ;
                X is Y-1
            )
        ;
            Y is X+1
        ).
```

Here, we suspend only in the case that both arguments are variables, and wake up as soon as either of them gets instantiated.
Waiting for groundness of a term can be done in a way similar to the way **succ_lazy/2** waited for both arguments to be instantiated: we pick any variable in the nonground term and wait for its instantiation. If this happens, we check whether other variables remain, and if yes, we re-suspend on one of the remaining variables. The following predicate waits for a term to become ground, and then calls arithmetic evaluation on it:

```
eval_lazy(Expr, Result) :-
        ( nonground(Expr, Var) ->
            suspend(eval_lazy(Expr,Result), 0, Var->inst)
        ;
            Result is Expr
        ).
```

We have used the built-in predicate **nonground/2** which tests a term for groundness and returns one of its variables if it is nonground. Note also that in this implementation the same **eval_lazy/2** goal gets woken and re-suspended possibly many times. See section 18.9 below for how to address this inefficiency.

Waking on Binding: bound

Sometimes it is interesting to wake a goal when the number of variables among its arguments is reduced. This happens not only when a variable disappears due to instantiation, but also

when two variables get unified (the result being a single variable). Consider the **succ_eager/2** predicate above: we know that a goal like `succ_eager(X,X).` must always fail because an integer cannot be equal to its successor. However, the above implementation does not detect this case until X gets instantiated.
The `bound` waking condition subsumes the `inst` condition, but also wakes when any two of the variables in the condition specification get unified with each other (aliased). Using this property, we can improve the implementation of **succ_eager/2** as follows:

```
succ_eager1(X, Y) :-
        ( var(X) ->
            ( var(Y) ->
                X \== Y,
                suspend(succ_eager1(X,Y), 0, [X,Y]->bound)
            ;
                X is Y-1
            )
        ;
            Y is X+1
        ).
```

This gives us the desirable behaviour of failing as soon as possible:

```
?- succ_eager1(X, Y), X = Y.
No (0.00s cpu)
```

Note that the built-in predicate **~=/2** is a similar case and uses the `bound` waking condition for the same reason.

Waking on Constraining: constrained

In plain Prolog, variable instantiation is the only way in which a single variable can become more constrained. In the presence of constraints, there are other ways. The most obvious example are variable domains: when a variable's domain gets reduced, the variable becomes more constrained. This means that a delayed goal that previously still had a chance to succeed, could now have become impossible to satisfy, and should therefore be checked again.
The purpose of the `constrained` waking condition is to make it possible to wake a suspended goal whenever a variable becomes more constrained in a general sense. Having this general notion of constrained-ness makes it possible to write generic libraries that do interesting things with constraints and constrained variables without their implementation having to be linked to a particular constraint-solver[3].
The `constrained` waking condition subsumes the `bound` condition (which in turn subsumes the `inst` condition). While goals suspended on the `inst` and `bound` conditions are woken implicitly by the unification routine, libaries which implement domain variables are responsible for notifying the system when they constrain a variable. They do so by invoking the built-ins **notify_constrained/1** and **wake/0** which is the generic way of telling the system that a variable has been constrained.
The simplest application using the `constrained` condition is a little debugging support predicate that prints a variable's current partial value (e.g., domain) whenever it changes:

[3]Examples of such libraries are **branch_and_bound**, **changeset**, **chr/ech**, **propia**, **repair**, **visualisation**.

```
report(X) :-
        ( var(X) ->
            writeln(constrained(X)),
            suspend(report(X), 1, X->constrained)  % (re)suspend
        ;
            writeln(instantiated(X))
        ).
```

This now works with any library that implements a notion of constrainedness, e.g., the interval solver library(ic):

```
?- report(X), X :: 1..5, X #> 2, X #< 4.
constrained(X)
constrained(X{1 .. 5})
constrained(X{3 .. 5})
instantiated(3)
X = 3
Yes (0.01s cpu)
```

The **report/1** predicate is woken when the domain is initally attached to X, whenever the domain gets reduced, and finally when X gets instantiated.

18.7.2 Library-defined Waking Conditions on Variables

Constraint-solver libraries typically define additional, specialised waking conditions for the type of variable that they implement. For instance, the interval solver lib(ic) defines the following conditions:

min: wake when the minimum domain value changes;

max: wake when the maximum domain value changes;

hole: wake when the domain gets a new hole;

type: wake when the variable type changes from real to integer.

Obviously, these conditions only make sense for domain variables that are created by the lib(ic) library, and are mainly useful for implementing extensions to this library, e.g., new constraints. The library-defined waking conditions can be used with **suspend/3** by using one of the following syntactic forms:

```
[A, B]->ic:min
[A, B]->ic:(min of ic)
```

Using these conditions, we can define a more specialised form of the above **report/1** predicate which only wakes up on the specified ic-domain changes:

```
report_ic(X) :-
        ( var(X) ->
            writeln(newdomain(X)),
            suspend(report_ic(X), 1, [X->ic:min,X->ic:max,X->ic:hole])
        ;
            writeln(instantiated(X))
        ).
```

The behaviour is similar to above, the predicate wakes up on every domain change:

```
?- X::1..5, report_ic(X), X#> 2, X #< 4.
newdomain(X{1 .. 5})
newdomain(X{3 .. 5})
instantiated(3)
X = 3
Yes (0.00s cpu)
```

Note that we now have to set up the delayed goal *after* the variable already has a domain. This is because the ic-specific waking conditions can only be used with ic-variables,[4] not with domain-less generic variables.

18.7.3 Global Symbolic Waking Conditions: Triggers

Although waking conditions for a goal are usually related to variables within the goal's arguments, it is also possible to specify symbolic waking conditions which are unrelated to variables. These are called **triggers** and are identified simply by an arbitrary name (an atom). Goals can be suspended on such triggers, and the trigger can be pulled explicitly by program code in particular circumstances. By combining triggers with the event mechanism (chapter 14) it is even possible to wake goals in response to synchronous or asynchronous events.
A goal is suspended on a trigger using the syntax *trigger(Name)* in **suspend/3** as in the following example:

```
?- suspend(writeln(woken), 0, trigger(happy)).
There is 1 delayed goal.
Yes (0.00s cpu)
```

The built-in **trigger/1** can then be used to wake the goal:

```
?- suspend(writeln(woken), 0, trigger(happy)), trigger(happy).
woken
Yes (0.00s cpu)
```

Of course, symbolic triggers can be used together with other waking conditions to specify alternative reasons to wake a goal.

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 **bb_min/3** and **minimize/2**.

A suspension should be attached to the `postponed` trigger only when

[4]More precisely, variables which have an ic-attribute, see chapter 17.

- 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 originally woke on 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.

18.8 Lower-level Primitives

Suspended goals are actually represented by a special opaque data type, called **suspension**, which can be explicitly manipulated under program control using the primitives defined in this section. 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.

18.8.1 Suspensions and Suspension Lists

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, e.g., `X > Y`.

The goal module The module from which the goal was called.

The scheduling priority The priority with which the goal will be scheduled when it becomes woken.

The run priority The priority under which the goal will eventually be executed.

The state This indicates the current position of the suspension within the resolvent. It is either suspended (sleeping), scheduled or executed (dead).

Additional data Debugging information etc.

Suspensions which should be woken by the same event are grouped together in a **suspension list**. Suspension lists are either stored in an attribute of an attributed variable or attached to a symbolic trigger.

18.8.2 Creating Suspended Goals

The most basic primitive to create a suspension is

make_suspension(*Goal*, *Priority*, *Susp*)

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** is not used directly, but implicitly via **suspend/3,4** (described in section 18.6) 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, but see section 18.9 for an exception). 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 denotes the process of both scheduling and eventual execution.
By default, suspensions are printed as follows (the variants with invocation numbers are used when the debugger is active):

'SUSP-_78-susp'	sleeping suspension with id _78
'SUSP-_78-sched'	scheduled suspension with id _78
'SUSP-_78-dead'	dead suspension with id _78
'SUSP-123-susp'	sleeping suspension with invocation number 123
'SUSP-123-sched'	scheduled suspension with invocation number 123
'SUSP-123-dead'	dead suspension with id invocation number 123

It is possible to change the way suspensions are printed by defining a **portray/3** transformation for the term type `goal`.

18.8.3 Operations on Suspensions

The following summarises the predicates that can be used to create, test, decompose and destroy suspensions.

make_suspension(*Goal*, *Priority*, *Susp*) Create a suspension 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 not a suspension or 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` (debugger invocation number).

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 18.9).

kill_suspension(*Susp*) Convert the suspension *Susp* into an executed one, i.e., remove the suspended goal from the resolvent. This predicate is meta-logical as its use may change the semantics of the program.

18.8.4 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.

18.8.5 Attaching Suspensions to Variables

Suspensions are attached to variables by means of the attribute mechanism. For this purpose, a variable attribute must 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 the above, 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` 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 initialized 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.

18.8.6 User-defined Suspension Lists

Many important attributes and suspension lists are either provided by the suspend-attribute or by libraries like the interval solver library lib(ic). For those suspension lists, initialization and waking is taken care of by the library code.

For the implementation of user-defined suspension lists, the following low-level primitives are provided:

init_suspension_list(+*Position*, +*Attribute*) Initializes argument *Position* of *Attribute* to an empty suspension list.

merge_suspension_lists(+*Pos1*, +*Attr*, +*Pos2*, +*Attr2*) Appends the first of two suspension lists (argument *Pos1* of *Attr1*) to the end of the second (argument *Pos2* of *Attr2*). NOTE: The append is destructive, i.e., the second list is modified.

enter_suspension_list(+*Pos*, +*Attr*, +*Susp*) Adds the suspension *Susp* to the suspension list in the argument position *Pos* of *Attr*. The suspension list can be pre-existing, or the argument could be uninstantiated, in which case a new suspension list will be created.

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, i.e., suspensions which are no longer useful are removed destructively. See section 18.8.8 for more details on waking.

18.8.7 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).

18.8.8 Scheduling Suspensions for Waking

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 scheduling priority (see figure 18.1). A suspension list can be passed to the scheduler by either of the predicates **schedule_suspensions/1** (for triggers) or **schedule_suspensions/2** (for uder-defined suspension lists). 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 does it start 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.[5]

18.9 Demon Predicates

A common pattern when implementing data-driven algorithms is the following variant of the **report/1** example from above:

```
report(X) :-
      suspend(report1(X), 1, X->constrained).      % suspend

report1(X) :-
        ( var(X) ->
            writeln(constrained(X)),
            suspend(report(X), 1, X->constrained)  % re-suspend
        ;
            writeln(instantiated(X))               % die
```

[5]This mechanism may be reconsidered in a future release.

```
        ).
```

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. Declaratively, 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
      ;
          writeln(instantiated(X)),
          kill_suspension(Susp)     % remove from the resolvent
      ).
```

18.10 More about Priorities

For the scheduled goals, 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 18.1). Each goal which is being executed is executed under a current priority. The priority of the currently executing goal can be determined with **get_priority/1**. This priority is

- normally inherited from the caller
- implicitly set to the goal's run_priority during waking
- explicitly set using **call_priority(*Goal*, *Prio*)**

All goals started from the ECL^{i}PS^e top-level loop or from the command line with the `-e` option have priority 12.

Priority-based execution is driven by a scheduler: it picks up the scheduled suspension with the highest scheduling priority. If its scheduling 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 under its run_priority (which may be higher than the scheduling priority). This is repeated until there are no suspensions with priority higher than that of the current goal.
Note that suspensions have two distinct priorities attached: the scheduling priority determining the order of execution, and the run_priority determining the atomicity of execution.

18.10.1 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, i.e., it will not be interrupted by goals with lower priority 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)
```

Note that woken goals are automatically executed under their run_priority (default 2), which usually make the use of **call_priority(*Goal*, *Prio*)** unnecessary.

18.10.2 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.

18.11 Details of the Execution Mechanism

18.11.1 Particularities of Waking by Unification

Goals that are suspended on the `inst` or `bound` waking conditions 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 on several suspending variables (as alternative waking conditions). This means that when one suspending variable is bound, several delayed goals may be woken at once. The order of executing woken 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 unifications 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 normally in-line expanded and thus simple, but can be regular when inlining is suppressed, e.g., by the `pragma(noexpand)` directive.

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 simple goals at the beginning of its body, they are considered part of the head (**extended head**) 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 also considered a simple goal and is therefore always executed *before* waking any pending suspended goals. This is important to know 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

```
filter(_P,[],[]) :- !.
filter(P,[N|LI],[N|NLI]) :-
        N mod P =\= 0,
        !,
        filter(P,LI,NLI).
filter(P,[N|LI],NLI) :-
        filter(P,LI,NLI).

delay integers(_, List) if var(List).
integers(_, []).
```

```
integers(N, [N|Rest]) :-
        N1 is N + 1,
        integers(N1, Rest).

?- integers(2, Ints), filter(2, Ints, [X1,X2]).
```

The idea here is that **integers/2** fills a list with integers on demand, i.e., whenever new list elements appear. The predicated filter/3 removes all integers that are a multiple of *P*. In the example query, the call to **integers/2** initially delays. When **filter/3** is called, Ints gets instantiated in the head unification of the second clause of **filter/3**, which will wake up **integers/2**. However, since the second clause of **filter/3** has an extended head which extends up to the cut, **integers/2** will not actually be executed until after the cut. Therefore, *N* is not yet instantiated at the time of the arithmetic test and causes an error message.

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.

As a consequence, the example can be fixed by inserting `true` at the beginning of the second clause. However, a preferable and more robust way is using the if-then-else construct, which always forces waking suspended goals before executing the condition. This would also be more efficient by avoiding the creation of 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)
        ).
```

18.11.2 Cuts and Suspended 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 delay without being woken before the cut, or if the head unification of the clause with the cut wakes any nondeterministic delayed goal, 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:

- Delaying *out of* the scope of a cut: a cut can be executed after some calls preceding it in the clause (or children of these calls) delay. When they are then woken later, they may cause the whole execution to fail instead of just the guard before the cut.

- Delaying *into* the scope of a cut: the head unification of a clause with cuts can wake delayed goals. If they are nondeterministic, the cut in the body of the waking clause will commit even the woken goals

18.12 Simulating the Delay-Primitives of other Systems

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 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. In the clause body, a delay clause allows more flexibility by supporting programming with (a subset of) built-ins. In general, it is a matter of taste whether specifying delay-conditions or execute-conditions is more straightforward. However, the semantics of delay clauses is certainly more intuitive in that missing delay clauses simply imply no delay, while missing when-declarations imply a most general `when ever` declaration.

Chapter 19

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.

19.1 Waiting for Instantiation

Goals that are to be woken when one or more variables become instantiated use the `inst` list. For instance, the following show how to implement a predicate **freeze/2**, such that the call `freeze(Term, Goal)` delays and is woken as soon as any variable in *Term* becomes instantiated:

```
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)
    ).
```

19.2 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 unifiable 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 unifiable 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.
- In the following program, 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-unifiable. 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.

19.3 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 unifiability 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 unifable 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 20

Memory Organisation And Garbage Collection

20.1 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:                  [1.12, 0.09, 2.74] seconds
session_time:           2.74 seconds
event_time:             2.74 seconds
global_stack_used:      1936 bytes
global_stack_allocated: 4456448 bytes
global_stack_peak:      4456448 bytes
trail_stack_used:       64 bytes
trail_stack_allocated:  262144 bytes
trail_stack_peak:       4456448 bytes
control_stack_used:     564 bytes
control_stack_allocated:262144 bytes
control_stack_peak:     262144 bytes
local_stack_used:       492 bytes
local_stack_allocated:  262144 bytes
local_stack_peak:       262144 bytes
shared_heap_allocated:  1613824 bytes
shared_heap_used:       1411000 bytes
private_heap_allocated: 73728 bytes
private_heap_used:      36992 bytes
gc_number:              1
gc_collected:           23472.0 bytes
gc_area:                23560 bytes
```

```
gc_ratio:               99.6264855687606 %
gc_time:                0.0 seconds
dictionary_entries:     3252
dict_hash_usage:        2117 / 8192
dict_hash_collisions:   314 / 2117
dict_gc_number:         2
dict_gc_time:           0.01 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 20.2.

20.1.1 The Shared/Private Heap

The heap is used to store a variety of data:

compiled code: The heap is used to store compiled Prolog code. Consequently its size is increased by the various **compile**-predicates, the **assert**-family and by **load/1**. Space 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 (see **trimcore/0**), since the removed code may still be under execution. Erasing a module also reclaims all the memory occupied by the module's predicates.

non-logical storage: All facilities for storing information across backtracking use the heap to do so. This includes the handle-based facilities (bags, shelves) as well as the name-based facilities (records, non-logical variables and arrays). As a general rule, when a stored term is overwritten, the space for the old value is reclaimed. All memory related to a non-logical store is reclaimed when the store is destroyed (e.g., using **erase_array/1**, **erase_all/1**, **bag_abolish/1**, **shelf_abolish/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 (see **set_flag/2**). The dictionary is designed to hold several thousand 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 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 heap. They are freed when the stream is closed.

allocation in C-externals: If third party libraries or external predicates written in C/C++ call `malloc()` or related C library functions, this space is also allocated from the heap. It is the allocating code's responsibility to free this space if it becomes unused.

Note that the distinction between shared and private heap is only relevant for parallel ECL^{i}PS^e systems, where multiple workers share the shared heap, but have their own private heap and stacks.

20.1.2 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. 20.1.3).

20.1.3 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.

20.1.4 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. 5.4). 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 20.2 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.

20.1.5 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.

20.2 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 predicate calls 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), e.g., for time-critical execution sequences.

Predicate calls related to the stack collector are:

set_flag(gc_policy, adaptive): This option affects the triggering heuristics of the garbage collector, together with the `gc_interval` setting. The `adaptive` policy (the default) minimises garbage collection time.

set_flag(gc_policy, fixed): As above, but the `fixed` policy minimises space consumption.

set_flag(gc_interval, *Nbytes*): Specify how often the collector is to be 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

[1] Provided that the underlying operating system supports this.

garbage collections. This is normally only necessary when the `gc_policy` is set to fixed. With `gc_policy` set to adaptive, the collection intervals will be adjusted automatically.

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, e.g., before the tail-recursive call in something like

```
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 (in bytes) reclaimed by all the garbage collections.

statistics(gc_area, *Bytes*): The average global stack area that was scanned by each garbage collection. This number should be close to the amount selected with `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 cpu time 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 garbage collections performed on the dictionary during this ECL^{i}PS^e session.

statistics(dict_gc_time, *Seconds*): The total cpu time spent by all dictionary garbage collections.

Chapter 21

Operating System Interface

21.1 Introduction

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. The interface is mostly compatible across Unix and Windows operating systems.

21.2 Environment Access

A number of predicates and global flags is provided to get more or less useful information from the operating system environment.

21.2.1 Command Line Arguments

Arguments provided on the UNIX (or DOS) command line are accessed by the built-ins **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. If the first argument of **argv/2** is the atom all, then a list of all command line arguments is returned.

21.2.2 Environment Variables

On 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.
```

The environment variables available on Window is version dependent, and is not a recommended method of passing information.

21.2.3 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)`.

21.2.4 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** built-in 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**.

21.2.5 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.

21.2.6 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.

21.3 File System

A number of built-in predicates is provided for dealing with files and directories. Here we consider only the file as a whole, for opening files and accessing their contents refer to chapter 11.

21.3.1 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.

21.3.2 Looking at Directories

To look at the contents of a directory, **read_directory/4** is available. It takes a directory pathname and a file name 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.
```

21.3.3 Checking Files

For checking the existence of files, **exists/1** or the more powerful **existing_file/4** 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)`; not all entries are meaningful under Windows), and accessibility information. It fails when the specified file does not exist. Refer to the reference manual or **help/1** for details.

21.3.4 Renaming and Removing Files

For these basic operations with files, **rename/2** and **delete/1** are provided.

21.3.5 File names

The file names used by ECL^{i}PS^e is in the Unix format, including on Window platforms, with the addition that the disk such as `C:` is indicated by `//C/`, so a Windows file name such as `"C:\my\path\name.ecl"` should be writen as `"//C/my/path/name.pl"`. The utility predicate **os_file_name/2** provides for the conversion between the format used in ECL^{i}PS^e and the Operating Systems' format.
The utility **pathname/4** is provided to ease the handling of file names. It takes a full pathname and cuts it into the directory pathname, the file name proper and a suffix (the part beginning with the last dot in the string). It also expands symbolic pathnames, starting with `~`, `~user` or `$var`.

```
[eclipse 1]: Name = "~octopus/prolog/file.pl",
        pathname(Name, Path, File, Suffix).

Path = "/usr/octopus/prolog/"
File = "file.pl"
Name = "~octopus/prolog/file.pl"
Suffix = ".pl"
yes.
```

21.4 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.

21.4.1 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` streams 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.

21.4.2 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, end_of_line, "", _, 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.
```

21.4.3 Interprocess Signals

The UNIX (or the appropriate Windows) 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.

Chapter 22

Interprocess Communication

ECL^{i}PS^e contains built-in predicates that support interprocess communications using sockets. Sockets 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**.

22.1 Socket Domains

Currently there are two available domains, `unix` and `internet`. The communication in the `unix` domain is limited to a single machine running under an Unix operating system, 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. It can also connect two processes on the same machine. 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 `get_flag(hostname, Host)`. The port identifies the channel on the host which is used for the communication. This is available under both Unix and Windows operating systems.

22.2 Stream Connection (internet domain)

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. The socket addresses 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.
```

22.3 Datagram Connection (internet domain)

This type of communication is the most general one offered by ECL^{i}PS^e. It is based on packets sent from one process to another, perhaps across a network. Any machine which is reachable over the network can participate in the communication.
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

[1]The packet boundaries are not of much interest in Prolog because every Prolog term represents itself a message with clear boundaries.

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.

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. Note that this example is Unix specific.

```
% 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, end_of_line, "", _, 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(, Command),
```

```
    exec(['rsh', RHost, 'eclipse', Host, Port, '-b',
       '/home/lp/micha/sepia4/up.pl', '-e', 'start'], [], _),
    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
```

22.4 Stream Connection (unix domain)

The sequence of operations is the same as for the internet domain, however in the unix domain the socket addresses are the file names:

```
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.
```

22.5 Datagram Connection (unix domain)

This is similar to datagram connection in the internet domain, except that it is limited to communications between two processes on the same Unix machine.
Again, like in the internet domain, the connection must be established in both directions if bi-direction communication is required:

```
server:
    % Make a named socket and read two terms from it
    [eclipse 10]: socket(unix, datagram, s), bind(s, '/tmp/sock').

    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.
```

Chapter 23

Language Dialects, ISO Prolog and Porting Prolog Applications

The ECL^{i}PS^e system has evolved from the Edinburgh family of Prolog systems, and thus shares many properties with other systems in the same tradition. It also supports the ISO Prolog Standard from 1995 and its 2005, 2012 and 2017 corrigenda.
However, the default programming language dialect used with ECL^{i}PS^e (known as `eclipse_language`) is a separate and unique dialect, which is the result of design decisions taken for conceptual, practical and occasionally historical reasons.
To run an application written in another Prolog dialect on ECL^{i}PS^e, one has basically two choices: Using a compatibility package, or modifying the program.

23.1 Using compatibility language dialects

The ECL^{i}PS^e compatibility language dialects are the fastest way to get a program running that was originally written for a different system. The dialects are implemented as libraries. The module system makes it possible for different application modules to use different language dialects.
To use a particular language dialect, prefix your program with a **module/3** directive that specifies the desired language dialect, for example

```
:- module(mymodule, [], iso).
```

Here, the last argument of the module/3 directive indicates the language (the default being eclipse_language). It is not advisable to use `:-lib(iso)` or `:-ensure_loaded(library(iso))` within an eclipse_language module, because this would lead to import conflicts between the different versions of built-in predicates.
Examples of supported language dialects are

- ISO Standard Prolog (**iso_strict**, **iso** and **iso_light**).
- C-Prolog (**cprolog**), one of the oldest and most influential Prolog implementations.
- Quintus Prolog (**quintus**), an early influential commercial system.
- SICStus Prolog (**sicstus**), an academic and commercial system based on Quintus.

- SWI Prolog (**swi**), a popular Prolog with large user base.

See the Reference Manual for details on the compatibility provided by the language dialects. The language dialects are just modules which provide the necessary code and exports to emulate a particular Prolog dialect. This module is imported instead of the default `eclipse_language` dialect which provides the ECL^{i}PS^e language. The source code of the language dialect module 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.
For quick experiments with a language dialect, ECL^{i}PS^e can be started with a different `default_language` option (see appendix D), e.g.

```
% eclipse -L <dialect>
```

This will give you a toplevel prompt in the given language dialect. The same effect can be achieved by setting the ECLIPSEDEFAULTLANGUAGE environment variable to the name of the chosen dialect.

23.1.1 ISO Prolog

The ISO Prolog standard [1] is supported in three variants:

- The **iso_strict** dialect provides an implementation of ISO Standard Prolog and complies strictly with ISO/IEC 13211-1 (Information Technology, Programming Languages, Prolog, Part 1, General Core, 1995) and the technical corrigenda ISO/IEC 13211-1 TC1 (2007), TC2 (2012) and TC3 (2017).

- The **iso** dialect provides an implementation of ISO Standard Prolog and in addition includes ECLiPSe functionality that does not conflict with the standard.

- The **iso_light** dialect provides the essence of ISO features without aiming for full conformance (in particular with respect to error handling), and may include ECLiPSe extensions that go beyond what the letter of the standard allows.

The specification of implementation-defined features stipulated by the standard can be found in the reference manual for **iso_strict** and **iso**.

23.1.2 Compiler versus interpreter

The following problem can occur despite the use of compatibility packages: 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.

23.2 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. For example, name clashes are fixed more easily by using the **local/1** declaration rather than by renaming the clashing predicate in the whole application program.

- Due to lack of standardisation, some subtle differences in the syntax exist between Prolog systems. See A.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 5.4).

- I/O predicates of the **see** and **tell** group are not built-ins 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**.

- In ECL^{i}PS^e, some built-ins 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** built-in, using **:/2** to access the original version:

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

  A third alternative 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, it may be necessary to add **discontiguous/1** directives if you want to compile such procedures.

23.3 Exploiting the features of ECL^{i}PS^e

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:

- Compiler features relevant for performance can be found in section 6.7.
- Use ECL^{i}PS^e's nonlogical **storage** facilities (section 10), which 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 5.
- The predicates **get_flag/2**, **get_flag/3**, **get_file_info/3**, **get_stream_info/3** and **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).
- The TkECL^{i}PS^e GUI provides many features that should make developing programs easier than with the traditional tty interface.
- 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.

Appendix A

Syntax

A.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 ([2]).

A.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.

A.2.1 Character Classes

The following character classes exist:

Character Class	Notation Used	Default Members
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	line feed
atom_quote	AQ	'
string_quote	SQ	"
list_quote	LQ	`
chars_quote	CQ	
radix	RA	
ascii	AS	
solo	SL	! ;
special	DS	([{)] } , |
line_comment	CM	%
escape	ES	\
first_comment	CM1	/
second_comment	CM2	*
symbol	SY	# + - . : < = > ? @ ^ ~ $ &
terminator	TS	

The character class of any character can be modified by a **chtab-declaration**.

A.2.2 Groups of characters

Group Type	Notation	Valid Characters
alphanumerical	ALP	UC UL LC N
non escape	NES	any character except escape
sign	SGN	+ -

A.2.3 Valid Tokens

Terms are defined in terms of tokens, and tokens are defined in terms of characters and character classes. Individual tokens can be read with the predicates **read_token/2** and **read_token/3**. The description of the valid tokens follows.

Atoms

```
ATOM    = (LC ALP*)
        | (SY | CM1 | CM2 | ES)+
        | (AQ (NES | ESCSEQ)* AQ)
        | SL
        | []
        | {}
        | |
```

If the syntax option `doubled_quote_is_quote` is enabled, two immediately consecutive AQ characters may occur inside an AQ-quoted sequence, and will be interpreted as a single occurrence of the quote within the name. If the syntax option `bar_is_no_atom` is active, the vertical bar cannot be used as an atom, unless quoted.

Numbers

1. **integers**

   ```
   INT = [SGN] N+
   ```

2. **based integers**

   ```
   INTBAS = [SGN] N+ (AQ | RA) (N | LC | UC)+
   ```

   The base must be an integer between 1 and 36 included, the value being valid for this base.

   If the syntax option `iso_base_prefix` is active, the syntax for based integers is instead

   ```
   INTBAS = [SGN] 0 (b | o | x) (N | LC | UC)+
   ```

   which allows binary, octal and hexadecimal numbers respectively.

3. **character codes**

   ```
   INTCHAR = [SGN] (0 (AQ|RA)|AS) CHARCONST
   ```

   For all plain characters, CHARCONST is just that character, and the value of the integer is the character code of that character. For special characters, see the detailed definition of CHARCONST below A.2.3.

4. **rationals**

   ```
   RAT = [SGN] N+ UL N+
   ```

5. **floats**

   ```
   FLOAT = [SGN] N+ . N+ [ (e | E) [SGN] N+ | Inf | NaN ]
         | [SGN] N+        (e | E) [SGN] N+
   ```

If the syntax option `float_needs_point` is active, then only the first alternative (with floating point) is valid syntax.

6. **bounded reals**

   ```
   BREAL = FLOAT UL UL FLOAT
   ```

   where the first float must be less or equal to the second.

If the syntax option `blanks_after_sign` is active, then blank space (`BS*`) is allowed between the sign and the following digits.

Strings

```
STRING = SQ (NES | ESCSEQ | SQ BS* SQ)* SQ
```

Text enclosed in SQ (string_quote) characters is parsed as a constant of type string. By default, the double quote `"` is the SQ character.
By default, consecutive strings are concatenated into a single string literal. This behaviour can be disabled by the syntax option `no_string_concatenation`. If the strings are consecutive without intervening blank space, the `doubled_quote_is_quote` causes the doubled quotes to be interpreted as a single occurrence of the quote within the string.

Lists of numeric character codes

```
LIST = LQ (NES | ESCSEQ | LQ BS* LQ)* LQ
```

Text enclosed in LQ (list_quote) characters is parsed as a list of numeric character codes. For example, if the double quote `"` is defined as list_quote, then `"abc"` is parsed as `[97,98,99]`.
Concatenation and doubled quotes are handled as for SQ-quoted strings.

Lists of single-character atoms

```
LIST = CQ (NES | ESCSEQ | CQ BS* CQ)* CQ
```

Text enclosed in CQ (chars_quote) characters is parsed as a list of single-atom characters. For example, if the double quote `"` is defined as chars_quote, then `"abc"` is parsed as `['a','b','c']`.
Concatenation and doubled quotes are handled as for SQ-quoted strings.

Variables

```
VAR = (UC | UL) ALP*
```

End of clause

```
EOCL = . (BS | NL | <end of file>) | TS | <end of file>
```

If the syntax option `eof_is_no_fullstop` is active, then end-of-file alone does not act as EOCL.

Escape Sequences within Quotes

Within quoted constants (atoms, strings, character lists), the following escape sequences ESC-SEQ may occur, and lead to the corresponding special character being inserted into the quoted item.

ESCSEQ =	**Result**	**Syntax option**
ES `a`	ASCII alert (7)	
ES `b`	ASCII backspace (8)	
ES `f`	ASCII form feed (12)	
ES `n`	ASCII newline (10)	
ES `r`	ASCII carriage return (13)	
ES `t`	ASCII tabulation (9)	
ES `v`	ASCII vertical tab (11)	
ES `e`	ASCII escape (27)	not iso_restrictions
ES `d`	ASCII delete (127)	not iso_restrictions
ES `s`	ASCII space (32)	not iso_restrictions
ES (ES|AQ|SQ|LQ|CQ)	the ES,AQ,SQ,LQ or CQ character	
ES NL	ignored	
ES `c` (BS|NL)*	ignored	not iso_restrictions
ES three octal digits	character with given octal character code	not iso_escapes
ES octal digits ES	character with given octal character code	iso_escapes
ES `x` hex digits ES	character with given hexadecimal character code	

It is illegal for any other character to follow the ES. If the syntax option `iso_escapes` is active, the octal escape sequence can be of any length and must be terminated with an ES character. Some sequences are disabled by the `iso_restrictions` option.

Character Constants

An integer character constant (see 3) is by default introduced by the sequence `0'` and followed by CHARCONST, which is defined as one of the following:

CHARCONST =	**Represents**	**Syntax option**
(ALP|SL|DS|CM|CM1|CM2|SY|TS)	that character	
<SPACE>	ASCII space (32)	
(SQ|LQ|CQ)	the SQ,LQ or CQ character	
ES (ES|AQ|SQ|LQ|CQ)	the ES,AQ,SQ,LQ or CQ character	
ES `a`	ASCII alert (7)	
ES `b`	ASCII backspace (8)	
ES `f`	ASCII form feed (12)	
ES `n`	ASCII newline (10)	
ES `r`	ASCII carriage return (13)	
ES `t`	ASCII tabulation (9)	
ES `v`	ASCII vertical tab (11)	
AQ AQ	the AQ character itself	iso_escapes and do
ES octal digits ES	character with given octal character code	iso_escapes
ES `x` hex digits ES	character with given hexadecimal character code	
<TAB>	ASCII tabulation (9)	not iso_escapes
NL	ASCII newline (10)	not iso_escapes
AQ	the AQ character itself	not iso_escapes
ES	the ES character itself	not iso_escapes
ES `e`	ASCII escape (27)	not iso_restrictions
ES `d`	ASCII delete (127)	not iso_restrictions
ES `s`	ASCII space (32)	not iso_restrictions

It is recommended to use only those sequences that are recognised universally, i.e. independent of syntax option settings. The other sequences are present for compatibility with various Prolog dialects. The syntax options `iso_escapes` and `iso_restrictions` disable several of those. The AQ AQ sequence is of dubious value – it is recommended to write `0'\'` instead of `0'''`.

A.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 ; goals
```

```
term_h                 ::=    term_h(0)
                        |     term(1200)

term_g                 ::=    term_g(0)
                        |     term(1200)

term(0)                ::=     VAR            /* not a term_h */
                        |      attr_var       /* not a term_h */
                        |      ATOM
                        |      structure
                        |      structure_with_fields
                        |      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)
                        |     fxx(N)  term(N-1)  term(N-1)
                        |     fxy(N)  term(N-1)  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)

attr_var                 ::=    VAR { attributes }
                                /* Note: no space before { */

attributes               ::=    attribute
                          |     attribute , attributes

attribute                ::=    qualified_attribute
                          |     nonqualified_attribute

qualified_attribute      ::=    ATOM : nonqualified_attribute

nonqualified_attribute   ::=    term_a

structure                ::=    functor ( termlist )
                                /* Note: no space before ( */

structure_with_fields    ::=    functor { termlist }
                          |     functor { }
                                /* Note: no space before { */

subscript                ::=    structure list
                          |     VAR list
                                /* Note: no space before 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
                     |     INTCHAR
                     |     RAT
                     |     FLOAT
                     |     BREAL

bterm               ::=    ( clause )
                     |     { clause }

functor             ::=    ATOM                 /* arity > 0 */

rulech              ::=    :-
                     |     ?-
```

A.3.1 Comments

There are two types of comments: bracketed comments, which are enclosed by CM1-CM2 and CM2-CM1, and the end-of-line comment, which is enclosed by CM and NL. Both types of comment behave as separators. When the syntax option `nested_comments` is on (the default is off), bracketed comments can be nested.

A.3.2 Operators

In Prolog, the user is able to modify the syntax dynamically by explicitly declaring new operators. The built-in **op/3** performs this task. As in Edinburgh Prolog, a lower precedence value means that the operator binds more strongly (1 strongest, 1200 weakest).
Any atom (whether symbolic, alphanumeric, or quoted) can be declared as an operator. Once an operator has been declared, the parser will accept the corresponding operator notation, and certain output built-ins will produce the operator notation if possible. There are three classes of operators: prefix, infix and postfix.

- When `f` is declared as a prefix unary operator (`fx` or `fy`), then the term `f(X)` can alternatively be written as `f X`.
- When `f` is declared as a prefix binary operator (`fxx` or `fxy`), then the term `f(X,Y)` can alternatively be written as `f X Y`.
- When `f` is declared as a postfix operator (`xf` or `yf`), then the term `f(X)` can alternatively be written as `X f`.
- When `f` is declared an an infix operator (`xfx`, `xfy` or `yfx`), then the term `f(X,Y)` can alternatively be written as `X f Y`.

An operator can belong to more than one class, e.g., the plus sign is both a prefix and an infix operator at the same time.
In the associativity specification of an operator (e.g., `fx`, `yfx`), `x` represents an argument whose precedence must be lower than that of the operator. `y` represents an argument whose precedence

must be lower or equal to that of the operator. y should be used if one wants to allow chaining of operators (i.e., if one wants them to be associative). The position of the y will determine the grouping within a chain of operators. For example:

```
Example declaration       will allow          to stand for
---------------------------------------------------------------
:- op(500,xfx,in).        A in B              in(A,B)
:- op(500,xfy,in).        A in B in C         in(A,in(B,C))
:- op(500,yfx,in).        A in B in C         in(in(A,B),C)
:- op(500,fx ,pre).       pre A               pre(A)
:- op(500,fy ,pre).       pre pre A           pre(pre(A))
:- op(500, xf,post).      A post              post(A)
:- op(500, yf,post).      A post post         post(post(A))
:- op(500,fxx,bin).       bin A B             bin(A,B)
:- op(500,fxy,bin).       bin A bin B C       bin(A,bin(B,C))
```

Operator declarations are usually local to a module, but they can be exported and imported. The operator visible in a module is either the local one (if any), an imported one, or a predefined one. Some operators are pre-defined (see Appendix B on page 241). They may be locally redefined if desired.
Note that parentheses are used to build expressions with precedence zero and thus to override operator declarations.[1]

A.3.3 Operator Ambiguities

Unlike the canonical syntax, operator syntax can lead to ambiguities.

- For instance, when a prefix operator is followed by an infix or postfix operator, the prefix is often not meant to be a prefix operator, but simply the left hand side argument of the following infix or postfix. In order to decide whether that is the case, ECL^{i}PS^e uses the operator's relative precedences and their associativities, and, if necessary, a two-token lookahead. If this rules out the prefix-interpretation, then the prefix is treated as a simple atom. In the rare case where this limited lookahead is not enough to disambigute, the prefix must be explicitly enclosed in parentheses.

- Another source of ambiguity are operators which have been declared both infix and postfix. In this case, ECL^{i}PS^e uses a one-token lookahead to check whether the infix-interpretation can be ruled out. If yes, the operator is interpreted as postfix, otherwise as infix. Again, in rare cases parentheses may be necessary to enforce the interpretation as postfix.

- When a binary prefix operator is followed by an infix operator, then either of them could be the main functor. Faced with the ambiguity, the system will prefer the infix interpretation. To force the binary prefix to be recognised, the infix must be enclosed in parentheses.

A.4 Syntax Differences between ECL^{i}PS^e and other Prologs

ECL^{i}PS^e supports the following extensions of Prolog syntax:

[1]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.

- Attributed variables: `X{Attr}`.
- Rational numbers: `3_4`.
- Bounded real numbers: `1.99__2.01`.
- Array subscripts: `Matrix[3,4]`.
- Structures with named fields: `emp{age:33,salary:33000}`.
- Binary prefix operators: `some X p(X)`.

Some of these extensions can be disabled via syntax option settings (this is done for example by the compatibility packages). In addition to the above extensions, the following minor differences exist between default ECL^{i}PS^e syntax and most Prolog systems:

- In ECL^{i}PS^e, end of file is accepted as fullstop.
- By default, an unquoted vertical bar can be used as an atom or functor (controlled by the syntax option `bar_is_no_atom`).
- By default, operators with precedence higher than 1000 are allowed in a comma-separated list of terms, i.e., structure arguments and lists. The ambiguity is resolved by considering commas as argument separators rather than operators inside the term. Thus, for example,

  ```
  p(a :- b, c)
  ```

  is accepted and parsed as `p/2`. This behaviour can be disabled (and turned into a syntax error) by setting the syntax option `limit_arg_precedence`.
- By default, double-quoted items are parsed as strings, not as character lists. This behaviour can be changed via **set_chtab/2** which allows string-quotes, list-quotes and atom-quotes to be redefined.
- By default, consecutive string- or list-quotes have the effect of concatenating the quoted items, while consecutive atom-quotes have no special meaning. This can be changed by using the syntax option `doubled_quote_is_quote`.
- By default, blank space between a sign and a number is significant: When there is no space between sign and number, the sign is taken as part of the number. With space, the sign is taken as prefix operator. This is controlled by the syntax option `blanks_after_sign`.

A.5 Changing the Parser's behaviour

Some of these properties can be changed by choosing one of the following syntax options (see `syntax_options` in the description of **get_flag/2**). The following options exist (unless otherwise noted, the options are disabled by default):

bar_is_no_atom: disallow the use of an unquoted vertical bar as atom or functor, except when it occurs in infix-position.

bar_is_semicolon: translate occurrences of unquoted infix vertical bars into terms with functor ;/2, e.g. `(a|b) = (a;b)`.

based_bignums: Allow base notation even to write integers longer than the wordsize (this implies they are always positive because the most significant bit is not interpreted as a sign bit).

blanks_after_sign: ignore blank space between a sign and a number (by default, this space is significant and will lead to the sign being taken as prefix operator rather than the number's sign). Also allow signs of numbers to be quoted.

doubled_quote_is_quote: parse a pair of quotes within a quoted item as one occurrence of the quote within the item (atom, string, or character list). If this option is off (the default), this is simply parsed as two consecutive items. In the case of strings and character lists, these consecutive items are concatenated into a single literal, unless the no_string_concatenation options is set.

eof_is_no_fullstop: do not treat end-of-file as a fullstop.

float_needs_point: require floating point numbers to be written with a decimal point, e.g. `1.0e-3` instead of `1e-3`.

iso_escapes: ISO-Prolog compatible escape sequences within strings and atoms.

iso_base_prefix: allow binary, octal or hexadecimal numbers to be written with 0b, 0o or 0x prefix respectively, and disallow the `base'number` notation.

iso_restrictions: enable all ISO-Prolog syntax restrictions that are not controlled by individual settings. This includes: disallowing operators as operands of operators; disallowing an atom to be declared as both an infix and a postfix operator; restrictions on changing operator properties for comma, vertical bar, and the empty-bracket atoms.

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 (default).

no_array_subscripts: disallow the ECL^{i}PS^e specific array-subscript syntax.

no_attributes: disallow the ECL^{i}PS^e specific syntax for variable attributes in curly braces.

no_blanks: do not allow blanks between functor an opening parenthesis (default).

no_curly_arguments: disallow the ECL^{i}PS^e specific syntax for structures with named arguments in curly braces.

no_string_concatenation: do not parse consecutive string literals as a single (concatenated) string.

plus_is_no_sign: do not interpret a plus sign preceding a number as the number's sign (effectively ignoring it), but treat it as a possible prefix operator `+/1`.

read_floats_as_breals: read all floating point numbers as bounded reals rather than as floats. The resulting breal is a small interval enclosing the true value of the number in decimal notation.

syntax_errors_fail: the predicates of the read-family fail when encountering a syntax error (after printing an error message). Without this option, the predicates throw an error term of the form `error(syntax_error(MessageString), context(...))`.

var_functor_is_apply: allow variables as functors, and parse a term like X(A,B,C) as if it were apply(X,[A,B,C]).

A number of further syntax options is provided for the purpose of parsing non-Prolog-like languages, in particular the Zinc family:

atom_subscripts: allow subscripts after atoms, and parse a term like a[B,C] as if it were subscript(a,[B,C]).

general_subscripts: allow subscripts after atoms, parenthesized subterms and subscripted terms, and parse a term such as a[B][C] as if it were written in the form subscript(subscript(a,[B]),[C]), or a term such as (a-b)[C] as if it were subscript(a-b,[C]).

curly_args_as_list: parse the arguments of a term in curly brackets as a list, i.e., parse {a,b,c} as {}([a,b,c]) instead of the default {}((a,b,c)).

Syntax option settings can be local to a module or exported, e.g.,

```
:- local syntax_option(not nl_in_quotes).
:- export syntax_option(var_functor_is_apply).
```

A.6 Short and Canonical Syntax

The following table summarises the correspondence between the short syntax forms (supported by the parser and the term writer) and their corresponding canonical forms. Usually, the programmer does not have to be concerned about the canonical represention because the short syntax is accepted by the parser and reproduced by the term writer (unless canonical writing is explicitly requested).

Known as	Short	Canonical	Active
List	[A|B]	.(A,B)	always
Curly brackets	{A}	{}(A)	always
Subscripted variable	X[...]	subscript(X, [...])	default
Subscripted struct	S[...]	subscript(S, [...])	default
Declared structure	f{...}	with(f, [...])	default
Attributed variable	X{...}	'with attributes'(X, [...])	default
Variable functor	X(...)	apply(X, [...])	optional

Here A and B stand for arbitrary terms, X for a variable, S for a compound term in canonical syntax, f for an arbitrary functor, and the ellipsis for a comma-separated sequence of arbitrary terms.

Appendix B

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**.

Prec	Assoc	Operators
1200	xfx	[-->, :-, ?-, if]
1200	fx	[:-, ?-]
1190	fy	[help]
1190	fx	[delay]
1180	fx	[-?->]
1170	xfy	[else]
1160	fx	[if]
1150	xfx	[then]
1100	xfy	[;, do, '|']
1050	xfy	[->]
1050	xfx	[*->, except, from]
1050	fy	[import, reexport]
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	[+, -, /\, \/]
400	yfx	[*, /, //, <<, >>, div, mod, rem]
200	xfy	[^]
200	fy	[+, -, \]

Appendix C

Events

We list here the ECL^iPS^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	Context Module

If the context module is unknown, a free variable is passed.

C.1 Event Types

C.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 / 4
5	type error	error_handler / 4
6	out of range	error_handler / 4
7	string contains unexpected characters	error_handler / 2
8	bad argument list	error_handler / 2

C.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 / 4
23	comparison trap	compare_handler / 4
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

C.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 / 4
43	multiple definition postfix/infix	error_handler / 2
44	record already exists	error_handler / 2
45	record does not exist	undef_record_handler / 2
50	trying to modify a read-only ground term	error_handler / 2
60	referring to an undefined procedure	error_handler / 4
61	inconsistent tool redefinition	error_handler / 4
62	inconsistent procedure redefinition	error_handler / 4
63	procedure not dynamic	error_handler / 4
64	procedure already dynamic	dynamic_handler / 3
65	procedure already defined	error_handler / 4
66	trying to modify a system predicate	error_handler / 4
67	procedure is not yet loaded	error_handler / 4
68	calling an undefined procedure	call_handler / 4
69	autoload event	autoload_handler / 4
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	referring to a deprecated predicate	declaration_warning_handler / 3
76	predicate declared but not defined	declaration_warning_handler / 3
77	predicate used but not declared or defined	declaration_warning_handler / 3
78	calling a procedure with a reserved name	error_handler / 2

C.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
84	referring to non-exported predicate	declaration_warning_handler / 3
85	referring to non-existing module	declaration_warning_handler / 3
86	lookup module does not exist	no_lookup_module_handler / 4
87	attempt to redefine an existing local item	warning_handler / 3
88	attempt to redefine an existing exported item	warning_handler / 3
89	attempt to redefine an already imported item	warning_handler / 3
90	procedure is already reexported	error_handler / 4
91	not a tool procedure	error_handler / 2
92	trying to redefine an existing local procedure	error_handler / 4
93	trying to redefine an existing exported procedure	error_handler / 4
94	trying to redefine an existing imported procedure	error_handler / 4
96	ambiguous import	ambiguous_import_resolve / 3
97	module already exists	error_handler / 2
98	key not correct	error_handler / 2
99	unresolved ambiguous import	ambiguous_import_warn / 3
100	accessing a procedure defined in another module	undef_dynamic_handler / 3
101	trying to erase a module from itself	error_handler / 2

C.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: unexpected comma	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
123	illegal iteration specifier in do-loop	error_handler / 4
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

C.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	true / 0
134	procedure clauses are not consecutive	compiler_error_handler / 2
135	trying to redefine a protected procedure	compiler_error_handler / 2
136	trying to redefine a built-in predicate	compiler_error_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 / 3
148	bad pragma	pragma_handler / 3
149	code unit loaded	unit_loaded_handler / 3

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
138	Variable name (atom)	undefined
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	
148	Clause	Module

The second argument for the event 139 depends on the predicate where it was raised:

- **compile/1,2** - (file name, 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

C.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	toplevel_prompt / 2
154	toplevel: start of query execution	true / 0
155	toplevel: print values	print_values / 3
156	toplevel: print answer	tty_ask_more / 2
157	error exit	error_exit / 0
158	toplevel: entering break level	start_break / 3
159	toplevel: leaving break level	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 context 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

Event	Second Argument	Third Argument
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") calls **throw/1**, ECL^{i}PS^e is not exited. This is a way to prevent accidental exits from the system. Failure of the handler is ignored.

C.1.8 Macro Transformation Errors, Lexical Analyser

Event	Event Type	Default Event Handler
160	global macro transformation already exists	error_handler / 4
161	macro transformation already defined in this module	macro_handler / 3
162	no macro transformation defined in this module	warning_handler / 2
163	illegal attempt to remove the last member of a character class	error_handler / 2
164	toplevel: print banner	tty_banner / 2
165	can't compile an attributed variable (use add_attribute/2,3)	error_handler / 2
166	file successfully processed	record_compiled_file_handler / 3
167	initialization/finalization goal failed or aborted	warning_handler / 3

The event 164 is raised whenever the toplevel loop is restarted.

Event	Second Argument	Third Argument
164	the banner string	

C.1.9 I/O, Operating System, External Interface

Event	Event Type	Default Event Handler
170	system interface error	system_error_handler / 4
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
177	shared library not found	error_handler / 2
190	end of file reached	eof_handler / 4
191	output error	output_error_handler / 4
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
213	Error in external function	error_handler / 2
214	Licensing problem	error_handler / 2

C.1.10 Debugging, Object Files

Event	Event Type	Default Event Handler
230	uncaught exception	error_handler / 2
231	default help/0 message	fail / 0
249	debugger new suspensions event	bip_delay / 0
250	debugger init event	trace_start_handler_tty / 0
251	debugger builtin fail event	bip_port / 4
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 builtin call event	bip_port / 4
259	debugger builtin exit event	bip_port / 4
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 an eclipse object file	compiled_file_handler / 3
265	bad eclipse object 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
252	trace_line{port:Port,frame:Frame}	undefined
264	(File, [], [])	undefined
265	(File, [], [])	undefined

C.1.11 Extensions

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	cost_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

C.2 Stack Overflows

When a stack overflows, the system performs a **throw/1** with an appropriate exit tag, i.e.,

global_trail_overflow for overflows of the global/trail stack that holds all the program's data structures.

local_control_overflow for overflows of the local/control stack that holds information related to the control flow.

These exits can be caught by wrapping a goal that is likely to overflow the stacks into an appropriate **catch/3**, e.g.,

```
..., catch(big_goal(X), global_trail_overflow, react_to_overflow), ...
```

In the debugger, you can locate the overflow by jumping to a LEAVE port (z command). See chapter 20 for more details on memory usage.

C.3 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.

C.4 User-Defined Events

User-defined events should use atomic event names rather than numbers. See the description of **set_event_handler/2**.

Appendix D

Command Line and Startup Options

D.1 Command Line Options

The ECL^{i}PS^e system has several parameters which may be specified on the command line at invocation time. All the parameters are available with the command line version of `eclipse`; with `tkeclipse`, only the `-g` and `-l` parameters are available. The parameters are as follows:

−b file The same as −f file.

−f file Compile the file `file` before starting the session. Multiple `-f` options are allowed. The file name is expected to be in the operating system's syntax. The file is processed by **ensure_loaded/1**, i.e., it can be a precompiled file or a source file, and file extensions are added as specified there.

−e goal Instead of starting an interactive toplevel, 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 `-f` option and is executed afterwards. Only one `-e` option is allowed.

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, 2 for abort).

When you only have a runtime installation of eclipse, the -e option is compulsory because a runtime system does not have an interactive toplevel.

−g size This option specifies the limit to which the memory consumption of the ECL^{i}PS^e global/trail stack can grow. The size is specified in kilobytes (followed by an optional `K`), in megabytes (followed by `M`) or in gigabytes (followed by `G`). The default is `512M` on 64-bit, or `256M` on 32-bit machines. The amount required for this stack depends on the program's data structures and may have to be increased for very large applications.

−l size This option specifies the limit to which the memory consumption of the ECL^{i}PS^e local/control stack can grow. The size is specified in kilobytes (followed by an optional `K`), in megabytes (followed by `M`) or in gigabytes (followed by `G`). The default is `128M` on 64-bit, or `64M` on 32-bit machines. The local/control stack is unlikely to require more than this default. If it does, it is probably caused by a programming error.

−D directory This options allows one to explicitly specify the ECL^{i}PS^e installation directory, i.e., the directory in which the system tries to find the ECL^{i}PS^e runtime system and libraries. This option overrides (and renders unnecessary) any setting of the ECLIPSEDIR environment variable (Unix) or, respectively, an ECLIPSEDIR registry entry (Windows) that may be in effect.

−L language The name of the language dialect used in the "top level module". The default is `eclipse_language`, other possible values include `iso`, `iso_strict`, `quintus` etc. This property can also be set via an `ECLIPSEDEFAULTLANGUAGE` environment variable.

−t module The name of the "top level module". This is an initially empty module, created by the system, which serves as the context for `-f` and `-e` options, and in which interactive toplevel queries are executed. This can be an arbitrary name, as long as it does not conflict with important library names. The default is `eclipse`.

−P Enable support for the sampling profiler (see library(**profile**). This will cause a slight slow-down in execution.

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

D.2 TkECL^{i}PS^e Startup Settings

`tkeclipse` accepts the same −g and −l options as `eclipse` for setting stack sizes.
In addition, `tkeclipse` on UNIX reads settings from the files `.tkeclipserc` (for toplevel parameters) and `.tkeclipsetoolsrc` (for development tools parameters). The system will first search for these files in the current directory, and then in the user's home directory. Each parameter is specified on a separate line in the appropriate file, in the format

```
parameter value
```

On Windows, the settings are instead from the registry keys

```
HKEY_CURRENT_USER\\Software\\IC-Parc\\ECLiPSe\\tkeclipserc
HKEY_CURRENT_USER\\Software\\IC-Parc\\ECLiPSe\\tkeclipsetoolsrc
```

where each parameter is specified as a string value under the appropriate parameter key. In either case, the parameters can be modified and saved using the TkECL^{i}PS^e Preference Editor. The parameters corresponding to the ECL^{i}PS^e command line settings are

`globalsize` **(positive integer)** The maximum size to which the global/trail stack area can grow to. The unit is megabytes. Overridden by the -g option.

`localsize` **(positive integer)** The maximum size to which the local/control stack area can grow to. The unit is megabytes. Overridden by the -l option.

`initquery` **(string)** An ECL^{i}PS^e query that TkECL^{i}PS^e will execute immediately after startup. It can be used to perform user defined initialisations.

`default_language` **(string)** The default language name (default: eclipse_language). Can be overridden by ECLIPSEDEFAULTLANGUAGE environment variable.

`default_module` **(string)** The default toplevel module name (default: eclipse).

`with_profiler` **(0/1)** Enable support for the **profiler** (default: 0).

Appendix E

Style Guide

Every ECL^{i}PS^e programming project should adopt a number of style rules. This appendix gives only a sample set of rules, which can serve as a guideline. Project teams should adapt them to their own needs and taste.

E.1 Style rules

1. There is one directory containing all code and its documentation (using sub-directories).
2. Filenames are of the form `[a-z][a-z_]+` with the extension `.ecl`.
3. One file per module, one module per file.
4. Each module is documented with comment directives.
5. All required interfaces are defined in separate spec files which are included in the source with a *comment include* directive. This helps to separate specification and implementation code.
6. The actual data of the problem is loaded dynamically from the Java interface; for stand-alone testing data files from the data directory are included in the correct modules.
7. The file name is equal to the module name.
8. Predicate names are of the form `[a-z][a-z_]*[0-9]*`. Underscores are used to separate words. Digits should only be used at the end of the name. Words should be English.
9. Variable names are of the form `[A-Z_][a-zA-Z]*[0-9]*`. Separate words with capital letters. Digits should only be used at the end. Words should be English.
10. The code should not contain singleton variables, unless their names start with `_`. The final program must not generate singleton warnings.
11. Each exported predicate is documented with a comment directive.
12. Clauses for a predicate must be consecutive.
13. Base clauses should be stated before recursive cases.

14. Input arguments should be placed before output arguments.
15. Predicates which are not exported should be documented with a single line comment. It is possible to use comment directives instead.
16. The sequence of predicates in a file is top-down with a (possibly empty) utility section at the end.
17. All structures are defined in one file (e.g., `flow_structures.ecl`) and are documented with comment directives.
18. Terms should not be used; instead use named structures.
19. When possible, use do-loops instead of recursion.
20. When possible, use separate clauses instead of disjunction or if-then-else.
21. There should be no nested if-then-else constructs in the code.
22. All input data should be converted into structures at the beginning of the program; there should be no direct access to the data afterwards.
23. All integer constants should be parametrized via facts. There should be no integer values (others than 0 and 1) in rules.
24. The final code should not use failure-loops; they are acceptable for debugging or testing purposes.
25. Cuts (`!`) should be inserted only to eliminate clearly defined choice points.
26. The final code may not contain open choice points, except for alternative solutions that still can be explored. This is verified with the tracer tool in the debugger.
27. Customizable data facts should always be at the end of a file; their use is deprecated.
28. The predicate **member/2** should only be used where backtracking is required; otherwise use **memberchk/2** to avoid creating redundant choice points.
29. The final code may not contain dead code except in the file/module `unsupported.ecl`. This file should contain all program pieces which are kept for information/debugging, but which are not part of the deliverable.
30. The test set(s) should exercise 100 percent of the final code. Conformity is checked with the line coverage profiler.
31. Explicit unification (**=/2**) should be replaced with unification inside terms where possible.
32. There is a top-level file (`top.ecl`) which can be used to generated all on-line documentation automatically.
33. For each module, a module diagram is provided.
34. For the top-level files, component diagrams are provided.
35. Don't use **','/2** to make tuples.

36. Don't use lists to make tuples.
37. Avoid **append/3** where possible, use accumulators instead.

E.2 Module structure

The general form of a module is:

1. module definition
2. module comment or inclusion of a spec file
3. exported/reexported predicates
4. used modules
5. used libraries
6. local variable definitions
7. other global operations and settings
8. predicate definitions

E.3 Predicate definition

The general form of a predicate definition is:

1. predicate comment directive
2. mode declaration
3. predicate body

Appendix F

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 with `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 (the default) it is possible (and sometimes useful) to create cyclic data structures. For example, the unification of *X* and *g(X)* in

   ```
   X = g(X)
   ```

   will result in a cyclic structure

   ```
   X = g(g(g(g(g(...)))))
   ```

   Not all ECL^{i}PS^e built-in predicates handle cyclic terms gracefully. Care must be taken with all predicates which traverse the whole term, e.g., **copy_term/2**, **term_hash/4**, **writeq/2**, **assert/1**, **compile_term/1**. These will typically loop or overflow a stack when applied to cyclic terms. Note however that, starting from version 5.6, cyclic terms are allowed in all heap copying predicates (**setval/2**, **bag_enter/2**, **shelf_set/3**, **store_set/3**, **record/2**, etc.).

Index

Bibliography

[1] Joint Technical Committee ISO/IEC JTC 1. Information technology – Programming languages – Prolog – Part 1: General Core. Technical Report ISO/IEC 13211-1, 1995.

[2] D.L. Bowen. DEC-10 Prolog User's Manual. D.A.I. occasional paper 27, University of Edinburgh, December 1981.

[3] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[4] 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.

[5] *ECLiPSe 3.4 Extensions User Manual*, 1994.

[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] Stefano Novello and Joachim Schimpf. *ECLiPSe Embedding and Interfacing Manual*, 1999.

[12] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[13] Joachim Schimpf. Logical loops. In Peter. J. Stuckey, editor, *Proceedings of the 18th International Conference on Logic Programming*, pages 224–238. Springer, July/August 2002.

[14] D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI, October 1983.