Quintus Prolog Reference Manual

Quintus Computer Systems, Inc.
2345 Yale St, Palo Alto, CA 94306

Written by David Bowen, with contributions from Lawrence Byrd and Bill Kornfeld, and some material from the original DEC10 Prolog manual by Fernando Pereira, Luis Pereira and David Warren.

This manual corresponds to Quintus Prolog Release 1.0.

The information in this document is subject to change without notice and should not be construed as a commitment by Quintus Computer Systems. Every effort has been made to ensure the accuracy of this document, but Quintus Computer Systems, Inc. assumes no responsibility for any errors that may appear.

Copyright (C) 1985 Quintus Computer Systems, Inc.

Table of Contents

CHAPTER 1

INTRODUCTION

The purpose of this Reference Manual is to provide a complete description of the Quintus Prolog system. You do not need to read all of this before starting to use Quintus Prolog. On the contrary, it is organized so that you should be able to dip into it quickly to find out whatever you need to know about the system.

The syntax and semantics of Quintus Prolog are described in chapter 2 and chapter 3. The remaining chapters describe all the built-in predicates. These are divided up into chapters on the basis of the kinds of functions they perform. Of particular interest, if you are reading this manual, is the chapter on the on-line help system (chapter 4). This allows you to access the manual directly from Prolog whenever you need to look something up.

1-1. Notation used in this Manual

Each built-in predicate definition is headed by a goal template such as

```
setof(?X,+Goal,-Set)
```

Here X and the others are meta-variables which name the arguments so that we don't have to keep saying "its first argument" and so on. The characters which precede the meta-variables will seem familiar if you know the mode declarations of Prolog-20; their significance is as follows:

- `+` This argument is an input. It should initially be instantiated.
- `-` This argument is an output. It is returned by the system. That is, the output value is unified with any value which was supplied for this argument. The predicate fails if this unification fails.
- `?` This argument does not fall into either of the above categories. It is not necessarily an input nor an output, and it need not be instantiated.

1-2. Error Conditions

There are a number of different types of error conditions:

1. An instantiation fault arises when a built-in predicate is called with insufficiently instantiated arguments. This can mean that an uninstantiated variable was passed where an input value (+ argument) was expected. There are also predicates like name/2 which requires that at least one of its arguments be instantiated.
2. A type error arises when an argument to a built-in predicate is of the wrong type, for example when a structure is supplied and an

atom is expected.

3. There are various error conditions associated with input and output. For example, trying to open a non-existent file causes an error.

4. There are errors associated with breaching various restrictions made by the system, such as trying to re-define a built-in predicate or operator.

5. There are errors associated with breaching various limits of the system, such as trying to compile a very large procedure.

6. There are errors associated with breaching various external limits, such as running out of virtual space or running out of input/output channels.

Error conditions normally arise only in the execution of built-in predicates. The action taken varies: the predicate may simply fail, or it may give an error message. After an giving an error message, predicates usually fail, but sometimes they abort the program, and sometimes they succeed. In this manual, the definition of each built-in predicate includes a description of its behavior on errors.

Programs should not be made to depend on the fact that any specific built-in predicate simply fails in the event of an instantiation fault or a type error. This is an area which is likely to change in future releases.

1-3. Invoking Prolog

The general form of command for invoking Prolog from Unix is

```
prolog [saved-state] [+ [file-name]]
```

or

```
saved-state [+ [file-name]]
```

where the brackets signify that their contents are optional and

prolog is the name used at your site to refer to the default Prolog executable provided by Quintus.

saved-state is the name of a file containing a saved Prolog state. This saved state must have been produced using save_program/1 (see section 14-1) or a similar predicate.

+ signifies that you want to run Prolog under the Emacs interface.

file-name is the name of a file to be read into Emacs.

When Prolog is invoked it normally (but not necessarily, with your own saved states) looks in your home directory for a file named 'prolog.ini', and if it finds one it consults it. Note that it is not a limitation that the

'prolog.ini' file is consulted rather than compiled. If there are procedures that you want compiled you can put them in another file and put a directive such as

```
:- compile(my_procs).
```

in your 'prolog.ini' file. Of course, if there are a lot of these procedures you may want to create your own saved state to avoid spending time recompiling them every time you enter Prolog.

CHAPTER 2

SYNTAX

This chapter describes the syntax of Quintus Prolog.

2-1. Terms

The data objects of the language are called terms. A term is either a constant, a variable or a compound term.

A constant is either a number (integer or floating point) or an atom. Constants are definite elementary objects, and correspond to proper nouns in natural language.

2-1-1. Integers

The printed form of an integer consists of a sequence of digits preceded optionally by a + or -. These are normally interpreted as base 10 integers. It is also possible to enter integers in other bases (1 through 36); this is done by preceding the digit string by the base followed by an apostrophe. If a base greater than 10 is used, the characters A-Z or a-z are used to stand for digits greater than 9. Examples of valid integer representations are:

```
1   -2345   85923   8'777   16'3F4A
```

2-1-2. Floating Point Numbers

A floating point number (float) consists of a sequence of digits with an embedded decimal point, optionally preceded by a sign and optionally followed by an exponent consisting of upper- or lower-case E and a signed base 10 integer. Examples of floating point numbers are:

```
1.0   -23.45   187.6E12   -0.0234e15   12.0E-2
```

2-1-3. Atoms

Examples of atoms are:

```
a   void   =   :=   'Anything in quotes'   []
```

An atom may take any of the following forms:

- Any sequence of alphanumeric characters (including '_'), starting with a lower case letter.
- Any sequence from the following set of characters:

+-*/\^<>=`~:.?@#$&

- Any sequence of characters delimited by single quotes. If the single quote character is included in the sequence it must be written twice, for example 'can''t'.

- Any of: ! ; [] {}
  Note that the bracket pairs are special: '[]' is an atom but '[' is not. However, when they are used as functors (see below) the forms [X] and {X} are allowed as alternatives to '[]'(X) and '{}'(X) respectively.

Note: it is recommended that you do not invent atoms beginning with the character '$', since it is just possible that such names can conflict with atoms with special significance for certain built-in predicates. Currently, the only atom which could conceivably cause a problem is '$VAR' which has special significance for write (see section 7-1-2), but there may be others in future.

2-1-4. Variables

Variables may be written as any sequence of alphanumeric characters (including '_') starting with either a capital letter or '_', for example

```
X   Value   A   A1   _3   _RESULT
```

If a variable is only referred to once, it does not need to be named and may be written as an anonymous variable, indicated by the underline character '_'.

2-1-5. Compound Terms

The structured data objects of the language are the compound terms. A compound term comprises a functor (called the principal functor of the term) and a sequence of one or more terms called arguments. A functor is characterized by its name, which is an atom, and its arity or number of arguments. For example, the compound term whose functor is 'point' of arity 3 and the arguments X, Y and Z, is written

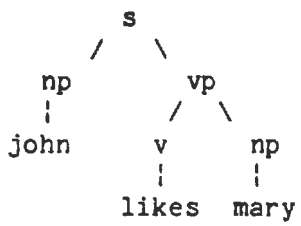
```
point(X,Y,Z)
```

Note that an atom is sometimes considered to be a functor of arity 0.

Functors are generally analogous to common nouns in natural language. One may think of a functor as a record type and the arguments of a compound term as the fields of a record. Compound terms are usefully pictured as trees. For example, the term

```
s(np(john),vp(v(likes),np(mary)))
```

would be pictured as the structure



Sometimes it is convenient to write certain functors as operators - binary functors (that is, functors of two arguments) may be declared as infix operators, and unary functors (that is, functors of one argument) may be declared as either prefix or postfix operators. Thus it is possible to write

```
X+Y     (P;Q)     X<Y      +X     P;
```

as optional alternatives to

```
+(X,Y)   ;(P,Q)   <(X,Y)   +(X)   ;(P)
```

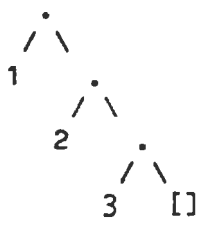
The use of operators is described fully in section 2-2 below.

2-1-6. Lists

Lists form an important class of data structures in Prolog. They are essentially the same as the lists of LISP: a list is either the atom

```
[]
```

representing the empty list, or else a compound term with functor '.' and two arguments which are respectively the head and tail of the list, where the tail of a list is another list. Thus a list of the first three natural numbers is the structure



which could be written, using the standard syntax, as

```
.(1,.(2,.(3,[])))
```

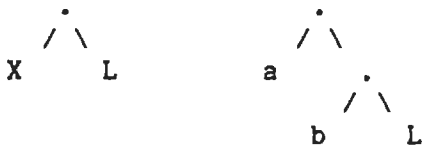
but which is normally written, in a special list notation, as

```
[1,2,3]
```

The special list notation used when the tail of a list is a variable is exemplified by

```
[X|L]     [a,b|L]
```

representing



respectively.

Note that the notation [X|L] does not add any new power to the language; it simply makes it more readable. The above examples could be written equally well as

```
.(X,L)    .(a,.(b,L))
```

For convenience, a further notational variant is allowed for lists of integers which correspond to ASCII character codes. Lists written in this notation are called strings. For example

```
"Humpty-Dumpty"
```

represents exactly the same list as

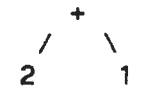
```
[72,117,109,112,116,121,45,68,117,109,112,116,121]
```

2-2. Operators

Operators in Prolog are simply a notational convenience. For example, '+' is an infix operator so that

```
2 + 1
```

is an alternative way of writing the term +(2,1). That is, 2+1 represents the data structure



and not the number 3. (The addition would only be performed if the structure were passed as an argument to an appropriate procedure, such as is; see section 8-2.)

Prolog syntax allows operators of three kinds - infix, prefix and postfix. An infix operator appears between its two arguments, while a prefix operator precedes its single argument and a postfix operator follows its single argument.

Each operator has a precedence, which is a number from 1 to 1200. The

precedence is used to disambiguate expressions where the structure of the term denoted is not made explicit through the use of parentheses. The general rule is that the operator with the highest precedence is the principal functor. Thus if '+' has a higher precedence than '/', then

```
a+b/c     a+(b/c)
```

are equivalent and denote the term "+(a,/(b,c))". Note that the infix form of the term "/(+(a,b),c)" must be written with explicit parentheses:

```
(a+b)/c
```

If there are two operators in the subexpression having the same highest precedence, the ambiguity must be resolved from the types of the operators. The possible types for an infix operator are

```
xfx     xfy     yfx
```

Operators of type 'xfx' are not associative: it is a requirement that both of the arguments of the operator must be subexpressions of lower precedence than the operator itself; that is, their principal functors must be of lower precedence, unless the subexpression is written in parentheses (which gives it zero precedence).

Operators of type 'xfy' are right-associative: only the first (left-hand) subexpression must be of lower precedence; the right-hand subexpression can be of the same precedence as the main operator. Left-associative operators (type 'yfx') are the other way around.

A functor named Name is declared as an operator of type Type and precedence Precedence by the command

```
:-op(Precedence,Type,Name).
```

The argument Name can also be a list of names of operators of the same type and precedence.

It is possible to have more than one operator of the same name, so long as they are of different kinds: infix, prefix or postfix. An operator of any kind may be redefined by a new declaration of the same kind. This applies equally to the operators which are built in to the system. Declarations for all these built-in operators can be found in section IV.

For example, the built-in operators '+' and '-' are declared by

```
:-op(500, yfx, [ +, - ]).
```

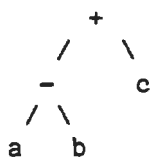
so that

```
a-b+c
```

is valid syntax, and means

```
(a-b)+c
```

or, pictorially



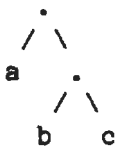
The list functor '.' is not a standard operator, but we could declare it thus:

```
:-op(900, xfy, .).
```

Then

```
a.b.c
```

would represent the structure



Contrasting this with the diagram above for a-b+c shows the difference between 'yfx' operators where the tree grows to the left, and 'xfy' operators where it grows to the right. The tree cannot grow at all for 'xfx' operators; it is simply illegal to combine 'xfx' operators having equal precedences in this way.

The possible types for a prefix operator are

```
fx      fy
```

and for a postfix operator they are

```
xf      yf
```

The meaning of the types should be clear by analogy with those for infix operators. As an example, if 'not' were declared as a prefix operator of type 'fy', then

```
not not P
```

would be a permissible way to write "not(not(P))". If the type were 'fx', the preceding expression would not be legal, although

```
not P
```

would still be a permissible form for "not(P)".

If these precedence and associativity rules seem rather complex, remember that you can always use parentheses when in any doubt.

2-3. Built-in Predicates for Handling Operators

2-3-1. op(+Precedence,+Type,+Name)

Declares the atom Name to be an operator of the stated Type and Precedence (refer to section 2-2). Name may also be a list of atoms in which case all of them are declared to be operators. If Precedence is 0, then the operator properties of Name (if any) are cancelled.

An appropriate error message is given if any of the arguments is not of the required form, but the goal always succeeds.

2-3-2. current_op(?Precedence,?Type,?Op)

The atom Op is currently an operator of type Type and precedence Precedence. Neither Op nor the other arguments need be instantiated at the time of the call, that is, this predicate can be used to find the precedence or type of an operator, or to backtrack through operators.

2-4. Syntax Restrictions

Note carefully the following syntax restrictions, which serve to remove potential ambiguity associated with prefix operators.

1. The arguments of a compound term written in standard syntax must be expressions of precedence below 1000. Thus it is necessary to write the expression "P:-Q" in parentheses.

   ```
   assert((P:-Q))
   ```

   because the precedence of the infix operator ':-', and hence of the expression "P:-Q", is 1200. Putting the expression in parentheses reduces its precedence to 0.

2. In a term written in standard syntax, the principal functor and its following '(' must not be separated by any intervening spaces, newlines etc. Thus

   ```
   point (X,Y,Z)
   ```

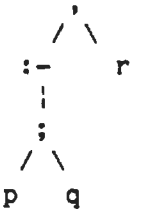
   is invalid syntax.

3. If the argument of a prefix operator starts with a '(', this '(' must be separated from the operator by at least one space or other non-printable character. Thus

   ```
   :-(p;q),r.
   ```

   (where ':-' is the prefix operator) is invalid syntax. The system

would try to interpret it as the structure:

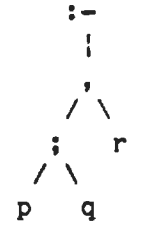


That is, it would take ':-' to be a functor of arity 1. However, since the arguments of a functor are required to be expressions of precedence below 1000, this interpretation would fail as soon as the ';' (precedence 1100) was encountered.

In contrast, the term:

```
:- (p;q),r.
```

is valid syntax and represents the following structure.



4. If a prefix operator is written without an argument, as an ordinary atom, the atom is treated as an expression of the same precedence as the prefix operator, and must therefore be written in parentheses where necessary. Thus the parentheses are necessary in

```
X = (?-)
```

since the precedence of '?-' is 1200.

2-5. Comments

Comments have no effect on the execution of a program, but they are very useful for making programs more comprehensible. Two forms of comment are allowed in Prolog:

1. The character '%' followed by any sequence of characters up to the end of the line.

2. The symbol '/*' followed by any sequence of characters (including new lines) up to '*/'.

2-6. Formal Syntax

A Prolog program consists of a sequence of sentences. Each sentence is a Prolog term. How terms are interpreted as sentences is defined in section 2-6-2 below. Note that a term representing a sentence may be written in any of its equivalent syntactic forms. For example, the binary functor ':-' could be written in standard prefix notation instead of as the usual infix operator.

Terms are written as sequences of tokens. Tokens are sequences of characters which are treated as separate symbols. Tokens include the symbols for variables, constants and functors, as well as punctuation characters such as parentheses and commas.

The interpretation of sequences of tokens as terms is defined in section 2-6-3 below. Each list of tokens which is read in (for interpretation as a term or sentence) has to be terminated by a full-stop token. Two tokens must be separated by a space token if they could otherwise be interpreted as a single token. Both space tokens and comment tokens are ignored when interpreting the token list as a term. A comment may appear at any point in a token list (separated from other tokens by spaces where necessary).

The interpretation of sequences of characters as tokens is defined in section 2-6-4 below. But first section 2-6-1 describes the notation used in the formal definition of Prolog syntax.

2-6-1. Notation

- Syntactic categories (or nonterminals) are printed in italics, for example *question*. Depending on the section, a category may represent a class of either terms, token lists, or character strings.

- A syntactic rule takes the general form

  C --> *F1* | *F2* | *F3*

  which states that an entity of category *C* may take any of the alternative forms *F1*, *F2*, *F3*, etc.

- Certain definitions and restrictions are given in ordinary English, enclosed in { } brackets.

- A category written as *C...* denotes a sequence of one or more *C*s.

- A category written as *?C* denotes an optional *C*. Therefore *?C...* denotes a sequence of zero or more *C*s.

- A few syntactic categories have names with arguments, and rules in which they appear may contain meta-variables in the form of italicized capital letters. The meaning of such rules should be clear from analogy with the definite clause grammars described in chapter 20.

- In section 2-6-3, particular tokens of the category Name are written as quoted atoms, while tokens which are individual punctuation characters are written literally.

2-6-2. Syntax of Sentences as Terms

```
sentence        --> clause | directive | grammar-rule

clause          --> non-unit-clause | unit-clause

directive       --> command | question | file-list

non-unit-clause --> ( head :- goals )

unit-clause     --> head
                        {where head is not otherwise a sentence}

command         --> ( :- goals )

question        --> ( ?- goals )

file-list       --> list

head            --> term
                        {where term is not a number or a variable}

goals           --> ( goals , goals )
                  | ( goals ; goals )
                  | goal

goal            --> term
                        {where term is not a number and
                         is not otherwise a goals}

grammar-rule    --> ( gr-head --> gr-body )

gr-head         --> nonterminal
                  | ( nonterminal , terminals )

gr-body         --> ( gr-body , gr-body )
                  | ( gr-body ; gr-body )
                  | nonterminal
                  | terminals
                  | gr-condition

nonterminal     --> term
                        {where term is not a number or variable
                         and is not otherwise a gr-body}

terminals       --> list | string

gr-condition    --> { goals }
```

2-6-3. Syntax of Terms as Tokens

```
term-read-in    --> subterm(1200) full-stop

subterm(N)      --> term(M)
                        {where M is less than or equal to N}

term(N)         --> op(N,fx)
                  | op(N,fy)
                  | op(N,fx) subterm(N-1)
                        {except the case '-' natural-number}
                        {if subterm starts with a '(', op must be
                         followed by a space}
                  | op(N,fy) subterm(N)
                        {if subterm starts with a '(', op must be
                         followed by a space}
                  | subterm(N-1) op(N,xfx) subterm(N-1)
                  | subterm(N-1) op(N,xfy) subterm(N)
                  | subterm(N) op(N,yfx) subterm(N-1)
                  | subterm(N-1) op(N,xf)
                  | subterm(N) op(N,yf)

term(1000)      --> subterm(999) , subterm(1000)

term(0)         --> functor ( arguments )
                        {provided there is no space between functor
                         and the '('}
                  | ( subterm(1200) )
                  | { subterm(1200) }
                  | list
                  | string
                  | constant
                  | variable

op(N,T)         --> functor
                        {where functor has been declared as an
                         operator of type T and precedence N}

arguments       --> subterm(999)
                  | subterm(999) , arguments

list            --> []
                  | [ listexpr ]

listexpr        --> subterm(999)
                  | subterm(999) , listexpr
                  | subterm(999) '|' subterm(999)

constant        --> atom | number

number          --> integer | float

atom            --> name
                        {where name is not a prefix operator}
```

```
integer         --> natural-number
                 ¦  - natural-number

float           --> unsigned-float
                 ¦  - unsigned-float

functor         --> name
```

2-6-4. Syntax of Tokens as Character Strings

```
token           --> name
                 ¦  natural-number
                 ¦  variable
                 ¦  string
                 ¦  punctuation-char
                 ¦  space
                 ¦  comment
                 ¦  full-stop

name            --> quoted-name
                 ¦  word
                 ¦  symbol
                 ¦  solo-char
                 ¦  []
                 ¦  {}

quoted-name     --> ' quoted-item... '

quoted-item     --> char  {other than '}
                 ¦  ''

word            --> capital-letter ?alpha...
                        {in the 'NOLC' convention only}

word            --> small-letter ?alpha...

symbol          --> symbol-char...
                        {except in the case of a full-stop
                         or where the first 2 chars are /* }

natural-number  --> digit...
                 ¦  base ' alphanumeric...
                         {where each alphanumeric must be less than the
                         base, counting 'a' as 10, 'b' as 11, etc}

base            --> digit...
                         {must be in the range 1 .. 36}

unsigned-float  --> simple-float
                 ¦  simple-float E exponent

simple-float    --> digit... decimal-point digit...
```

```
decimal-point   —> .

E               --> E ¦ e

exponent        --> digit...
                 ¦  - digit...
                 ¦  + digit...

variable        --> underline ?alpha...

variable        --> capital-letter ?alpha..
                          {in the 'LC' convention only}

string          —> " ?string-item... "

string-item     --> char  {other than "}
                 ¦  ""

space           --> layout-char...

comment         --> /* ?char... */
                          {where ?char... must not contain */}
                 ¦  % rest-of-line

rest-of-line    --> newline
                 ¦  ?not-end-of-line... newline

not-end-of-line --> {any character except newline}

newline         --> {ASCII code 10}

full-stop       --> . layout-char

char            --> layout-char
                 ¦  alpha
                 ¦  symbol-char
                 ¦  solo-char
                 ¦  punctuation-char
                 ¦  quote-char

layout-char     —> {any ASCII character code up to 32,
                    includes <blank>, <cr> and <lf>}

alpha           --> alphanumeric ¦ underline

alphanumeric    --> letter ¦ digit

letter          —> capital-letter ¦ small-letter

capital-letter  --> {any character from the list
                     ABCDEFGHIJKLMNOPQRSTUVWXYZ}

small-letter    --> {any character from the list
                     abcdefghijklmnopqrstuvwxyz}
```

digit --> {any character from the list 012346789}

symbol-char --> {any character from the list +-*/\^<>=`~:.?@#$&}

solo-char --> {any character from the list ;!%}

punctuation-char --> {any character from the list ()[]{},|}

quote-char --> {any character from the list '"}

underline --> {the character _}

2-6-5. Notes

1. The expression of precedence 1000 (that is, belonging to syntactic category term(1000)) which is written

   X,Y

   denotes the term

   ','(X,Y)

   in standard syntax.

2. The bracketed expression (belonging to syntactic category term(0))

   (X)

   denotes simply the term X.

3. The curly-bracketed expression (belonging to syntactic category term(0))

   {X}

   denotes the term

   '{}'(X)

   in standard syntax.

4. Note that, for example, "-3" denotes an integer whereas "-(3)" denotes a compound term which has the unary functor '-' as its principal functor.

5. The double quote character '"' within a string must be written duplicated. That is,

   """"

   represents a string of one double quote character only. Similarly

for the single quote character within a quoted atom.

CHAPTER 3

SEMANTICS

This chapter gives an informal description of the semantics of Quintus Prolog.

3-1. Programs

A fundamental unit of a logic program is the goal or procedure call:

```
gives(tom,apple,teacher)    reverse([1,2,3],L)    X<Y
```

A goal is merely a special kind of term, distinguished only by the context in which it appears in the program. The (principal) functor of a goal is called a predicate. It corresponds roughly to a verb in natural language, or to a procedure name in a conventional programming language.

A logic program consists simply of a sequence of statements called sentences, which are analogous to sentences of natural language.

A sentence comprises a head and a body. The head either consists of a single goal or is empty. The body consists of a sequence of zero or more goals (it may be empty). If the head is not empty, the sentence is called a clause.

If the body of a clause is empty, the clause is called a unit clause, and is written in the form

```
P.
```

where P is the head goal. We interpret this declaratively as

"P is true."

and procedurally as

"Goal P is satisfied."

If the body of a clause is non-empty, the clause is called a non-unit clause, and is written in the form

```
P :- Q, R, S.
```

where P is the head goal and Q, R and S are the goals which make up the body. We can read such a clause either declaratively as

"P is true if Q and R and S are true."

or procedurally as

"To satisfy goal P, satisfy goals Q, R and S."

A sentence with an empty head is called a directive, of which the most important kind is called a question and is written in the form

```
?- P, Q.
```

where P and Q are the goals of the body. Such a question is read declaratively as

"Are P and Q true?"

and procedurally as

"Satisfy goals P and Q.""

Sentences generally contain variables. A variable should be thought of as standing for some definite but unidentified object. This is analogous to the use of a pronoun in natural language. Note that a variable is not simply a writeable storage location as in most programming languages; rather it is a local name for some data object, like the variable of pure LISP. Note that variables in different sentences are completely independent, even if they have the same name -- the lexical scope of a variable is limited to a single sentence. To illustrate this, here are some examples of sentences containing variables, with possible declarative and procedural readings:

```
employed(X) :- employs(Y,X).
```

"Any X is employed if any Y employs X."

"To find whether a person X is employed,
find whether any Y employs X."

```
derivative(X,X,1).
```

"For any X, the derivative of X with respect to X is 1."

"The goal of finding a derivative for the expression X with
respect to X itself is satisfied by the result 1."

```
?- ungulate(X), aquatic(X).
```

"Is it true, for any X, that X is an ungulate and X is
aquatic?"

"Find an X which is both an ungulate and aquatic."

In any program, the procedure for a particular predicate is the sequence of clauses in the program whose head goals have that predicate as principal functor. For example, the procedure for a predicate concatenate of three arguments might well consist of the two clauses

```
concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
concatenate([],L,L).
```

where concatenate(L1,L2,L3) means "the list L1 concatenated with the list L2

is the list L3".

In Prolog, several predicates may have the same name but different arities. Therefore, when it is important to specify a predicate unambiguously, the form name/arity is used, for example concatenate/3.

Certain predicates are predefined by the Prolog system. Such predicates are called built-in predicates.

As we have seen, the goals in the body of a sentence are linked by the operator ',' which can be interpreted as conjunction ("and"). It is sometimes convenient to use an additional operator ';', standing for disjunction ("or"). (The precedence of ';' is such that it dominates ',' but is dominated by ':-'.) An example is the clause

```
grandfather(X,Z) :-
    (mother(X,Y); father(X,Y)), father(Y,Z).
```

which can be read as

```
"For any X, Y and Z,
    X has Z as a grandfather if
    either the mother of X is Y or the father of X is Y,
    and the father of Y is Z."
```

Such uses of disjunction can always be eliminated by defining an extra predicate - for instance the previous example is equivalent to

```
grandfather(X,Z) :- parent(X,Y), father(Y,Z).
parent(X,Y) :- mother(X,Y).
parent(X,Y) :- father(X,Y).
```

- and so disjunction will not be mentioned further in the following, more formal, description of the semantics of clauses.

The token '|', when used outside a list, is an alias for ';'. The aliasing is performed when terms are read in, so that

```
a :- b | c.
```

is read as if it were

```
a :- b ; c.
```

Note the double use of the '.' character. On the one hand it is used as a sentence terminator, while on the other it may be used in a string of symbols which make up an atom (for example, the list functor '.'). The rule used to disambiguate terms is that a '.' followed by a layout-character is regarded as a sentence terminator, where a layout-character is defined to be any character less than or equal to ASCII 32 (this includes space, tab, newline and all control characters).

3-2. Declarative and Procedural Semantics

The semantics of definite clauses should be fairly clear from the informal interpretations already given. However it is useful to have a precise definition. The declarative semantics of definite clauses tells us which goals can be considered true according to a given program, and is defined recursively as follows.

> A goal is true if it is the head of some clause instance and each of the goals (if any) in the body of that clause instance is true, where an instance of a clause (or term) is obtained by substituting, for each of zero or more of its variables, a new term for all occurrences of the variable.

For example, if a program contains the preceding procedure for concatenate, then the declarative semantics tells us that

```
concatenate([a],[b],[a,b])
```

is true, because this goal is the head of a certain instance of the first clause for concatenate, namely,

```
concatenate([a],[b],[a,b]) :- concatenate([],[b],[b]).
```

and we know that the only goal in the body of this clause instance is true, since it is an instance of the unit clause which is the second clause for concatenate.

Note that the declarative semantics makes no reference to the sequencing of goals within the body of a clause, nor to the sequencing of clauses within a program. This sequencing information is, however, very relevant for the procedural semantics which Prolog gives to definite clauses. The procedural semantics defines exactly how the Prolog system will execute a goal, and the sequencing information is the means by which the Prolog programmer directs the system to execute his program in a sensible way. The effect of executing a goal is to enumerate, one by one, its true instances. Here then is an informal definition of the procedural semantics.

> To execute a goal, the system searches forwards from the beginning of the program for the first clause whose head matches or unifies with the goal. The unification process (see "A Machine-Oriented Logic Based on the Resolution Principle" by J.A. Robinson, Journal of the ACM 12:23-44, January 1965) finds the most general common instance of the two terms, which is unique if it exists. If a match is found, the matching clause instance is then activated by executing in turn, from left to right, each of the goals (if any) in its body. If at any time the system fails to find a match for a goal, it backtracks; that is, it rejects the most recently activated clause, undoing any substitutions made by the match with the head of the clause. Next it reconsiders the original goal which activated the rejected clause, and tries to find a subsequent clause which also matches the goal.

For example, if we execute the goal expressed by the question

```
?- concatenate(X,Y,[a,b]).
```

we find that it matches the head of the first clause for concatenate, with X instantiated to [a|X1]. The new variable X1 is constrained by the new goal produced, which is the recursive procedure call

```
concatenate(X1,Y,[b])
```

Again this goal matches the first clause, instantiating X1 to [b|X2], and yielding the new goal

```
concatenate(X2,Y,[])
```

Now this goal will only match the second clause, instantiating both X2 and Y to []. Since there are no further goals to be executed, we have a solution

```
X = [a,b]
Y = []
```

That is, the following is a true instance of the original goal:

```
concatenate([a,b],[],[a,b])
```

If this solution is rejected, backtracking will generate the further solutions

```
X = [a]
Y = [b]

X = []
Y = [a,b]
```

in that order, by re-matching, against the second clause for concatenate, goals already solved once using the first clause.

3-3. Occur Check

Prolog's unification does not have an occur check; that is, when unifying a variable against a term the system does not check whether the variable occurs in the term. When the variable occurs in the term, unification should fail, but the absence of the check means that the unification succeeds, producing a circular term. Trying to print a circular term, or trying to unify two circular terms, will cause a loop. (You can always get out of a loop by typing ^C followed by an 'a' for abort.)

The absence of the occur check is not a bug or design oversight, but a conscious design decision. The reason for this decision is that unification with the occur check is at best linear on the sum of the sizes of the terms being unified, whereas unification without the occur check is linear on the size of the smallest of the terms being unified. In any practical programming language, basic operations are supposed to take constant time. Unification against a variable should be thought of as the basic operation of Prolog, but this can take constant time only if the occur check is omitted. Thus the

absence of an occur check is essential to make Prolog into a practical programming language. The inconvenience caused by this restriction seems in practice to be very slight. Usually, the restriction is only felt in toy programs.

3-4. The Cut

Besides the sequencing of goals and clauses, Prolog provides one other very important facility for specifying control information. This is the cut, written '!'. It is inserted in the program just like a goal, but is not to be regarded as part of the logic of the program and should be ignored as far as the declarative semantics is concerned.

The effect of the cut is as follows. When first encountered as a goal, cut succeeds immediately. If backtracking should later return to the cut, the effect is to fail the parent goal, that goal which matched the head of the clause containing the cut, and caused the clause to be activated. In other words, the cut operation commits the system to all choices made since the parent goal was invoked, and causes other alternatives to be discarded. The goals thus rendered determinate are the parent goal itself, any goals occurring before the cut in the clause containing the cut, and any subgoals which were executed during the execution of those preceding goals.

For example, the procedure

```
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).
```

can be used to test whether a given term is in a list:

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

returns the answer 'yes'. The procedure can also be used to extract elements from a list, as in

```
?- member(X,[d,e,f]).
```

With backtracking this will successively return each element of the list. Now suppose that the first clause had been written instead:

```
member(X,[X|L]) :- !.
```

In this case, the above call would extract only the first element of the list ('d'). On backtracking, the cut would immediately fail the whole procedure.

Another example:

```
x :- p, !, q.
x :- r.
```

This is analogous to "if p then q else r" in an Algol-like language.

Note that a cut discards all the alternatives that are subsequent to the parent goal, even when the cut appears within a disjunction. This means that the normal method for eliminating a disjunction by defining an extra predicate cannot be applied to a disjunction containing a cut.

CHAPTER 4

ON-LINE HELP

The documentation for Quintus Prolog comprises the User's Guide and this Reference Manual. This material is available both in a printed form, and also on-line using special built-in predicates which are described below. These predicates view the Manual and the Guide as a single tree structure which can be traversed by a user in order to find the information he or she needs. The easier way to use this help system is through the Emacs interface, but it is possible to do everything without it.

There are basically two ways of getting to information in the help system. The first is via a series of menus which corresponds to the hierarchy of chapters, sections and subsections. This method of access is analogous to looking through the Contents sections of the printed documents and is described in section 4-4 and section 4-5. The second method is by key-word search. This method is analogous to looking up the Index sections of the printed manuals and is described in section 4-3.

4-1. Help Files

The help system operates by reading help files into a special buffer if you are running under Emacs, or by writing them to the terminal (using the Unix "more" command) if you are not.

There are two types of help files: menu files and text files. If you are using Emacs, restricted and specialized key bindings apply in each of these types of window. For example, if you type a letter such as 'a' which would normally be inserted into the buffer, nothing happens; this is because you are not supposed to make changes to the help file. These key bindings are summarized in section 4-6-1 and section 4-6-2; the most important key binding is the question mark which will display the appropriate key-binding summary. After typing a question mark, you will probably want to type a 'b' to get back to where you were.

The organization and numbering of the help files corresponds directly to the organization and numbering of the printed manual. The text files correspond to "leaf" nodes of the manual tree, that is, to sections of the manual which have no subdivisions. The menu files correspond to the non-leaf nodes of the tree.

4-1-1. Menu Files

A menu file is distinguished by having the string "{menu}" at the end, although this is not displayed if you are viewing the file through Emacs. A menu consists of a numbered sequence of choices. Each choice has the form

```
Number - Subject                          {manual(SectionNumber)}
```

If you are running Prolog without Emacs you can select a menu choice by typing

```
| ?- manual(SectionNumber).
```

where SectionNumber is exactly as shown in the menu. Section numbers are in general of the form

```
Manual-Chapter-Section-Subsection
```

where Manual is either 'user', for the User's Guide, or 'ref' for the Reference Manual, and Chapter, Section and Subsection are numbers which correspond directly to a section in the printed Guide or Manual.

If you are running Prolog under Emacs then you will find that a special mode is entered when you are looking at a menu file (see section 4-6-1.)

4-1-2. Text Files

Text files are just ordinary files of text which correspond directly to sections or subsections in the printed manual. If you are running under Emacs, a special mode is entered when you are looking at a text file as a part of the help system (see section 4-6-2.) If you are not using Emacs you will see the string "{text}" at the bottom of the file; this is used purely to distinguish it from a menu file.

Occasionally you will see a cross reference in the text. Cross references look like this in the printed manual:

```
... see also section 1-2-3
```

and like this in the on-line manual

```
... see also {manual(ref-1-2-3)}
```

If you are not using Emacs, then you should type the goal

```
| ?- manual(ref-1-2-3).
```

in order to follow this cross reference. Under Emacs there is a more convenient way to do this: type 'x' followed by Return.

4-2. help

This simply gives basic information, such as how to start using the help system and how to exit from Prolog.

A hook is provided so that users can add to or replace this information: help/0 first calls user_help/0 and if that succeeds it does nothing else. Only if the call to user_help fails does the standard information get output.

4-3. help(+Topic)

This is the basic help command. It tries hard to accept any argument you give it as a topic for which help may exist in the manual. The argument is converted into a character string, and all the index entries which start with that string are combined into a menu which gives you a choice of entry points into the manual hierarchy.

It is not necessary to type the whole of the word which is the Topic you want information about. The fewer characters you type, the larger the menu is likely to be, because more index entries will begin with that character sequence. Thus

```
| ?- help(d).
```

will give you a menu of all index entries beginning with the letter 'd', and you can even try

```
| ?- help('').
```

(the null atom) to get a menu of the entire index!

4-4. manual

This gives you a menu of the top layer of the manual hierarchy. This menu gives you the choice between the User's Guide and the Reference Manual. Whichever you select, you will then be shown a menu of chapter titles. When you select a chapter you will see a menu of the section titles of that chapter, and so on.

See section 4-1-1 for a description of menu files. Also see manual/1, section 4-5, for how to get directly to a specific point in the manual hierarchy.

4-5. manual(+X)

This predicate has two purposes. One way of using it is by the goal

```
| ?- manual(SectionNumber).
```

where SectionNumber is in one of the forms

```
Manual-Chapter-Section-Subsection
Manual-Chapter-Section
Manual-Chapter
Manual
```

and Manual is either 'user' or 'ref', and Chapter, Section and Subsection are integers. This takes you directly to a particular point in the manual hierarchy.

The other way of using this predicate is

```
| ?- manual(Topic).
```

This is useful when you know that the manual contains a section about a topic and you want to get directly to it without any intervening menus. In particular, all the built-in predicates can be looked up directly like this. For example, manual(debug) will take you directly to the section of the manual defining the built-in predicate debug/0, while help(debug) will give you a menu referring to various sections of the manual which have something to do with debugging.

It is not necessary to type the whole of the word which is the Topic you want information about. You need only type enough characters to uniquely identify the Topic from the list of all possible topics. If the abbreviation you type is not unique, you will get a menu of the manual topics which start with the characters you supplied. Notice that this menu is different from the one produced by help/1; that menu would generally be larger since for any given topic it can include mention of places which just touch on that topic as well as the one place which covers that topic.

If there is no section in the manual which specifically covers Topic (or any topic beginning with the letters in Topic), then help(Topic) is tried (see section 4-3) in case there is some information somewhere on the Topic you have specified.

4-6. Emacs commands for using the Help System

There are two modes provided for viewing help files, depending on whether the file is a menu file or a text file. The commands available in each of these modes are listed below.

4-6-1. Menu Commands

The keys available when viewing a menu file of the help system under Emacs are:

<Space>	Move to the next menu entry (wraps around).
<Backspace>	Move to the previous menu entry (wraps around).
<Return>	Select the current menu entry.
b	Move Back to the previous help file viewed.
u	Move Up to the previous menu in the manual hierarchy.
q	Quit the help system.
Control-v	Scroll the menu one page forward.
<Escape>-v	Scroll the menu one page backward.

NOTE: If you are viewing this under Emacs, type 'b' to return to where you just were.

4-6-2. Text Commands

The keys available when viewing a text file of the help system under Emacs are:

<Space>	Scroll the text one page forward (same as Control-v).
<Backspace>	Scroll the text one page backward (same as Escape-v).
b	Move Back to the previous help file viewed.
u	Move Up to the previous menu in the manual hierarchy.
q	Quit the help system.
x	Move to the next cross-reference.
<Return>	Follow a cross-reference (use after 'x').

NOTE: If you are viewing this under Emacs, type 'b' to return to where you just were.

CHAPTER 5

LOADING PROGRAMS, CONSULTING AND COMPILING

There are two ways of loading programs into Prolog, consulting and compiling. Consulted code is interpreted and has the advantages of greater flexibility in debugging, and the fact that space is reclaimed when you re-consult a procedure or file after making some modifications. The advantage of compiling is that the program will run considerably faster and use much less space.

It is very easy to mix compiled and interpreted code, so that typically you will want to compile code which is well tested, and consult new, untested code which you are adding to it.

5-1. consult(+File)

consult(File) can also be written [File].

File should be an atom which is the name of a file of Prolog code, except that a ".pl" suffix to a file name may be omitted. If File does not end with the characters ".pl" then consult appends those characters before looking for the file. If the file name so formed does not exist, then the file name is searched for without the ".pl" suffix. If it still fails to find a file, an error indication is given, regardless of the setting of the fileerrors flag, and the goal succeeds without doing anything else.

Each procedure in the file is read in, and, after any previous version of that procedure has been deleted, it is added to the Prolog database in a form suitable for the interpreter.

In the event of an error, such as a syntax error or an attempt to redefine a built-in predicate, you will get an error indication and the clause causing the error will be ignored. You should correct the error and consult the file again.

There are a number of warnings which may appear when a file is consulted. These are designed to help you catch simple errors in your program, but you can turn some or all of them off if you wish: see section 5-3.

If there are any goals in the file being loaded, that is any terms with principal functor ':-'/1 or '?-'/1, then these are executed as they are encountered. The most useful type of goal to have in a file is one that consults another file, such as

```
:- [otherfile].
```

but you may have any goal you like. It should be noted that debugging will not be on in the execution of such a goal, regardless of the top-level debugging state, but you can always turn it on by an explicit call to debug/0 or trace/0; for example,

```
:- trace, myprog.
```

File can also be the atom 'user' which means that procedures are to be typed directly into Prolog from the terminal. If you are not running under the Emacs interface, a special prompt, '| ', is displayed on every new line while consulting from the terminal.

File can be a list of file names in which case all the named files are consulted.

If you are running under Emacs, you can consult a single procedure, a specified part of an edit buffer, or the whole of an edit buffer, using the Escape i command (see chapter 22).

5-2. compile(+File)

File should be an atom which is the name of a file of Prolog code, except that a ".pl" suffix to a file name may be omitted. If File does not end with the characters ".pl" then compile appends those characters before looking for the file. If the file name so formed does not exist, then the file name is searched for without the ".pl" suffix. If it still fails to find a file, an error indication is given, regardless of the setting of the fileerrors flag, and the goal succeeds without doing anything else.

Each procedure in the file is read in, and, after any previous version of that procedure has been deleted, it is added to the Prolog database in a compiled form.

Note that procedures which are declared to be dynamic (see section 16-1) are actually stored in the database in the same way as consulted procedures even if they are loaded via compile. This means that dynamic procedures lack the speed and space efficiency of compiled code; on the other hand they share the debugging flexibility of interpreted code.

In the event of an error, such as a syntax error or an attempt to redefine a built-in predicate, you will get an error indication and the clause causing the error will be ignored. You should correct the error and compile the file again.

There are a number of warnings which may appear when a file is compiled. These are designed to help you catch simple errors in your program, but you can turn some or all of them off if you wish: see section 5-3.

If there are any goals in the file being loaded, that is any terms with principal functor ':-'/1 or '?-'/1, then these are executed as they are encountered. The most useful type of goal to have in a file is one that compiles another file, such as

```
:- compile(otherfile).
```

but you may have any goal you like. It should be noted that debugging will not be on in the execution of such a goal, regardless of the top-level

debugging state, but you can always turn it on by an explicit call to debug/0 or trace/0; for example,

```
:- trace, myprog.
```

File can also be the atom 'user' which means that procedures are to be typed directly into Prolog from the terminal. If you are not running under the Emacs interface, a special prompt, '| ', is displayed on every new line while compiling from the terminal.

Finally, File can be a list of file names.

If you are running under Emacs, you can compile a single procedure, a specified part of an edit buffer, or the whole of an edit buffer, using the Escape k command (see chapter 22).

5-3. Style Checking

5-3-1. style_check(+X)

X should be one of the following atoms (or else the goal fails):

X = all	Turn on all style checking.
X = single_var	Turn on checking for clauses containing a single instance of a named variable, where variables which start with a '_' are considered not named.
X = discontiguous	Turn on checking for procedures for which all the clauses are not adjacent to one another in the file.
X = multiple	Turn on checking for multiple definitions for the same procedure in different files.

5-3-2. no_style_check(+X)

X should be one of the following atoms (or else the goal fails):

X = all	Turn off all style checking.
X = single_var	Turn off checking for clauses containing a single instance of a named variable, where variables which start with a '_' are considered not named.
X = discontiguous	Turn off checking for procedures for which all the clauses are not adjacent to one another in the file.
X = multiple	Turn off checking for multiple definitions for the same procedure in different files.

CHAPTER 6

CONTROL

6-1. +P , +Q

Conjunction: (P , Q) succeeds if P succeeds and then Q succeeds. This is not normally regarded as a built-in predicate, since it is part of the syntax of the language. However, it is like a built-in predicate in that you can say call((P , Q)) to execute P and then Q.

6-2. +P ; +Q

Disjunction: (P ; Q) succeeds if P succeeds or Q succeeds. Again, this is normally regarded as part of the syntax of the language but is like a built-in predicate in that you can say call((P ; Q)). The character "|" can be used as an alternative to ";".

6-3. !

Cut. When first encountered as a goal, cut succeeds immediately. If backtracking should later return to the cut, the effect is to cause the parent goal to fail. (The parent goal is the one which matched the head of the clause containing the cut.) See section 3-4.

6-4. call(+X)

If X is instantiated to an atom or compound term, then the goal call(X) is executed exactly as if that term appeared textually in its place, except that any cut ('!') occurring in X only cuts alternatives in the execution of X.

If X is not instantiated as described above, an error message is printed and the call fails.

6-5. \+ +P

This fails if the goal P has a solution, and succeeds otherwise. This is not real negation ("P is false"), which is not possible in Prolog, but negation-by-failure meaning "P is not provable". It is defined as if by

```
\+(P) :- P, !, fail.
\+(_).
```

Remember that with prefix operators such as this one it is necessary to be careful about spaces if the argument starts with a '('. For example:

```
| ?- \+ (P,Q).
```

is this operator applied to the conjunction of P and Q, but

```
| ?- \+(P,Q).
```

would require a predicate \+ of arity 2 for its solution. The prefix operator can, however, be written as a functor of one argument; thus

```
| ?- \+((P,Q)).
```

is also correct.

6-6. +P -> +Q ; +R

The conditional statement

"if P then Q else R"

Note that if P succeeds and Q then fails, backtracking into P does not occur. The -> acts like a cut except that its range is restricted to within the disjunction: it cuts away R and any choice points within P.

This construct could have been implemented in Prolog thus:

```
(P -> Q ; R) :- call(P), !, call(Q).
(P -> Q ; R) :- call(R).
```

Note that the operator precedences of the semicolon and -> are both greater than 1000, so that they dominate commas. For example,

```
p, q -> r, s ; t
```

is equivalent to

```
( (p,q) -> (r,s) ) ; t
```

6-7. +P -> +Q

When occurring other than as one of the alternatives of a disjunction, this is equivalent to

```
P -> Q ; fail.
```

(For a definiton of P->Q;R see section 6-6.) The -> cuts away any choice points in the execution of P, and it may be thought of as a local cut. Note that the operator precedence of -> is greater than 1000, so it dominates commas. Thus in:

```
f :- p, q -> r, s.
f.
```

the -> cuts away any choices in p or in q but unlike cut ('!') it does not cut

away the alternative choice for f.

6-8. true

Always succeeds. This could have been trivially defined in Prolog by the single clause

```
true.
```

6-9. otherwise

Always succeeds (same as **true**). **otherwise** is useful for laying out conditionals (see section 6-6) in a readable way.

```
(test1 ->
     goal1
;test2 ->
     goal2
;otherwise ->
     goal3
)
```

6-10. fail

Always fails.

6-11. false

Always fails (same as fail).

6-12. repeat

Succeeds immediately when called and whenever re-entered by backtracking. It is usually used to simulate the looping constructs found in traditional procedural languages. The general form of a repeat loop is the following:

```
repeat,
   action1,
   action2,
   ...,
   action<n>,
   test,
   !,
 ... rest of clause body ...
```

The effect of this is to execute action1 through action<n> in sequence. The test is then executed. If it succeeds, the loop is (effectively) terminated

by the cut ('!') (which cuts any choice points in the clause, including the one created by the repeat). A failure of the test will cause backtracking that will be caught by the repeat, which will succeed again and re-execute the actions.

The most common use of repeat loops has each of the actions always succeeding determinately, that is, not creating choice points. It tends to be a bit confusing if actions sometimes fail, so that backtracking starts before the test is reached, or if actions create choice points so that backtracking does not always go right back to the repeat.

The easiest way to understand the effect of repeat is to think of failures as "bouncing" back off them causing re-execution of the later goals. Note that the repeat loop can only be useful if one or more of the actions involves a side-effect, either a change to the database (such as an assert) or an I/O operation. Otherwise you would just be doing exactly the same thing each time around the loop (which would never terminate).

Repeat loops are not often needed; usually recursive procedure calls will lead to code that is easier to understand as well as being more efficient. There are certain circumstances, however, where repeat will lead to greater efficiency. An important property of Quintus Prolog is that any storage that has been allocated during a proof of a goal is recovered on backtracking through that goal. Thus, in the above example, any space allocated by any of the actions is (efficiently) reclaimed. This is not true of iterations implemented using recursion.

repeat could have been written in Prolog:

```
repeat.
repeat :- repeat.
```

CHAPTER 7

INPUT AND OUTPUT

Prolog provides two classes of predicates for input and output: those which handle individual characters, and those which handle complete Prolog terms.

Input and output happens with respect to streams. A stream can refer to a file or to the user's terminal. At any one time there is a current input stream and a current output stream. Input and output predicates fall into three categories:

1. those which use the current input or output stream;

2. those which use the standard input or output stream - these refer to the user's terminal;

3. those which take an explicit stream argument.

Initially, the current input and output streams both refer to the user's terminal, but there are predicates which allow you to change them when you want to read from, or write to, a file.

Quintus Prolog differs from Prolog-20 and other Prolog systems in that the Prolog top-level always refers to the standard input and output rather than the current input and output. Thus if you set the current output to some file and forget to close the file you don't suddenly find all your '| ?- ' prompts being sent to that file (and nothing to the terminal).

7-1. Input and Output of Terms

7-1-1. read(-X)

Reads the next term from the current input stream and unifies it with X. The term must be followed by a period ('.') followed by a space character (for example, a space, tab or linefeed; actually any character with ASCII code less than or equal to 32 may be used). The delimiting period and space character are removed from the input stream but are not a part of the term which is read.

The term is read with respect to current operator declarations. See section 2-2 for a discussion of operators.

read does not do anything until the term delimiter (period and space character) is encountered. Thus if you type at top level

```
| ?- read(X).
```

you will keep getting the prompt '|: ' every time you type a carriage return (except under Emacs where the prompt is not displayed) but nothing else will

happen, whatever you type, until you type a period.

When a syntax error is encountered, an error message is printed and then read tries again, starting immediately after the term delimiter of the erroneous "term". That is, read does not fail but perseveres until it eventually manages to read a term.

If the end of the current input stream has been reached, then read(X) will cause X to be unified with the atom end_of_file. If the end of the stream was reached on a previous input command, then this call will give an error message and abort the computation.

7-1-2. write(?X)

Writes the term X to the current output stream.

The term is written with respect to current operator declarations. See section 2-2 for a discussion of operators.

Atoms output by write can not in general be read back using read. E.g.

```
| ?- write('a b').
a b
```

If you want to be sure that the output can be read back by read you should use writeq which puts quotes around atoms when necessary.

Note that write does not output a period after the term it outputs. So if you want this term to be input by read you must mark the end of the term by explicitly outputting a period after it.

write treats terms of the form '$VAR'(N) specially; it writes 'A' if N=0, 'B' if N=1, ...'Z' if N=25, 'A1' if N=26, etc. Terms of this form are generated by numbervars/3 (see section 24-1).

7-1-3. display(?X)

Displays the term X on the standard output (normally this means the user's terminal) which is not necessarily the current output stream. Also, display ignores operator declarations and shows all compound terms in standard prefix form. For example, the command

```
:- display(a+b).
```

produces the following:

```
+(a,b).
```

Note that display does not output a period after the term it outputs. So if you want this term to be input by read you must mark the end of the term by explicitly outputting a period after it.

7-1-4. writeq(?X)

Writes the term X to the current output stream. This is the same as write(X) except that, when necessary, single quotes are put around atoms and functors to make them acceptable as input to read.

Note that writeq does not output a period after the term it outputs. So if you want this term to be input by read you must mark the end of the term by explicitly outputting a period after it.

writeq treats terms of the form '$VAR'(N) specially; it writes 'A' if N=0, 'B' if N=1, ...'Z' if N=25, 'A1' if N=26, etc. Terms of this form are generated by numbervars/3 (see section 24-1).

7-1-5. print(?X)

Prints X to the current output stream. By default, the effect of this predicate is the same as write/1, but you can change its effect by providing clauses for the predicate portray/1.

If X is a variable, then it is output using write(X). Otherwise a call is made to the user-definable procedure portray(X). If this succeeds, then it is assumed that X has been output and print exits (succeeds).

If the call to portray fails, and if X is a compound term, then write is used to write the principal functor of X and print is called recursively on its arguments. If X is atomic, it is written via write.

print treats lists ([_|_]) specially: it first gives the whole list to portray, but if this fails, it will only give each of the top level elements to portray. That is, portray is not called on all the tails of the sublists.

print/1 is called from within the system in two places:

1. to print the bindings of variables after a question has succeeded

2. to print a goal during debugging

7-2. Input and Output of Characters

This section describes built-in predicates that handle the input and output of single characters using the current input and current output streams. See also section 7-8 for input and output of characters specifically to the user's terminal, and section 7-5 for input and output of characters to a specified stream.

7-2-1. get0(-N)

Unifies N with the ASCII code of the next character from the current input stream. If there are no more characters in the stream, N is unified with -1. If the end of the stream was reached on a previous input command, then this call will give an error message and abort the computation.

7-2-2. get(-N)

Unifies N with the ASCII code of the next non-space character from the current input stream. Space characters are all those with ASCII codes less than or equal to 32; this includes space, tab, linefeed and all control characters. If there are no more non-space characters in the stream, N is unified with -1. If the end of the stream was reached on a previous input command, then this call will give an error message and abort the computation.

7-2-3. skip(+N)

Skips over characters from the current input stream to the first occurrence of the character with ASCII code N. N may be an integer expression. The most useful form of integer expression in this context is a single character in double quotes, for example, "a" which evaluates to 97, the ASCII code for the letter 'a', so that:

```
| ?- skip("a").
```

skips over (ignores) all input until the next occurrence of the letter 'a'.

An error indication is given if N is not a valid arithmetic expression. If N evaluates to a float, or an integer outside the range 0..127, the call simply fails.

7-2-4. put(+N)

Writes N to the current output stream. N should be a legal ASCII character code; or it may be an integer expression. The most useful form of integer expression in this context is a single character in double quotes, for example, "a" which evaluates to 97, the ASCII code for the letter 'a', so that:

```
| ?- put("a").
a
yes
```

An error indication is given if N is not a valid arithmetic expression. If N evaluates to a float, the call simply fails, and if it evaluates to an integer not in the range 0..127 then that integer is reduced modulo 128 to a number in the required range.

If the current output is the user's terminal, the character is not necessarily output immediately; see ttyflush, section 7-8-7.

7-2-5. nl

Starts a new line on the current output stream. That is a linefeed character (ASCII 10) is output, as if by

```
| ?- put(10).
```

7-2-6. tab(+N)

Writes N spaces to the current output stream. N may be an integer expression.

An error indication is given if N is not a valid arithmetic expression. If N evaluates to a negative integer the call simply succeeds without doing anything. If it evaluates to a float, the call fails.

If the current output is the user's terminal, the spaces are not necessarily output immediately; see ttyflush, section 7-8-7.

7-3. Stream Handling

7-3-1. Streams

Each input and output built-in predicate refers explicitly or implicitly to a stream. Streams may refer to a file or to the user's terminal. Each stream is either for input or for output, not both.

There is a limit of twenty input/output streams that can be open at any one time. Three of these streams are reserved for standard input, standard output and error output respectively. Standard input and output normally refer to your terminal, but may be redirected using Unix facilities. The error stream nearly always refers to the terminal, but can also be redirected.

The reserved streams are identified by the atoms 'user_input', 'user_output' and 'user_error'. In cases where it is unambiguous, the atom 'user' may be used in place of 'user_input' or 'user_output'. Other streams are identified by terms of the form

```
'$stream'(I,J)
```

where I and J are positive integers. The structure of such terms is irrelevant; it only matters that a term uniquely identifies a particular stream. Stream identifiers are returned by the predicate open/3 and can be passed as arguments to those procedures which need to use them.

If you try to perform input or output on a stream after it has been closed you

get an error indication.

7-3-2. open(+File,+Mode,-Stream)

File is a filename. Mode is one of the atoms 'read', 'write' or 'append'. The 'read' option is used for input. The 'write' and 'append' options are used for output. The 'write' option causes a new file to be created for output. The 'append' option opens an already existing file and adds output to the end of it. If the open request was successful a stream object is unified with Stream that can be subsequently used for input or output to the given file.

The 'read' and 'append' options generate error conditions if the file does not exist. These either cause an error indication followed by an abort (the default) or else simply fail, depending on the state of the fileerrors flag. See section 7-3-6.

7-3-3. open_null_stream(-Stream)

An output stream is opened which is not connected to any file. Characters or terms which are output to this stream go nowhere. This predicate is useful because various pieces of local state are kept for null streams: the predicates character_count/2, line_count/2, and line_position/2 can be used on these streams (see section 7-6).

7-3-4. close(+Stream)

The stream corresponding to Stream is closed. If Stream is a stream identifier but does not represent a currently open stream, an error indication is given and the call fails.

For Prolog-20 compatibility, this predicate also accepts an atom as its argument. If there is a single open stream with this atom as its associated file name, then this stream is closed. If there is more than one stream with the same file name, then one of them is selected arbitrarily and closed. If there is no stream corresponding to this file name then an error indication is given and the call fails.

If Stream is not a stream identifier or an atom, or if the specified stream cannot be closed, then an error indication will be given and execution aborted, unless you have turned off file errors with nofileerrors (see section 7-3-6) in which case the call simply fails.

7-3-5. current_stream(?File,?Mode,?Stream)

Stream is a currently open stream on file File in mode Mode where Mode is either 'read', 'write' or 'append'. None of the arguments need be initially instantiated. This predicate is non-determinate and can be used to backtrack

through all open streams. It fails when there are no (further) matching open streams.

The three special streams for the standard input, output and error channels are ignored by this predicate.

7-3-6. nofileerrors

Changes the fileerrors flag, so that the predicates see/1, tell/1, open/3 and close/1 simply fail, instead of producing an error message and then causing an abort, if the specified file cannot be opened or closed.

The fileerrors flag is only reset by an explicit call to fileerrors/0.

7-3-7. fileerrors

Cancels the effect of nofileerrors. It resets the fileerrors flag to its default state in which an error message is produced by see/1, tell/1, open/3 and close/1 if the specified file cannot be opened or closed. The error message is followed by an abort, that is, execution of the program is abandoned and the system returns to top level.

7-3-8. flush_output(+Stream)

Stream must be a stream open for output. Output to a stream does not necessarily get sent immediately; it is buffered. This predicate flushes the output buffer for the specified stream and thus ensures that everything that has been written to the stream is actually sent at that point.

There is also a predicate ttyflush (section 7-8-7) which is equivalent to flush_output(user).

7-3-9. set_input(+Stream)

This makes Stream the current input stream. Subsequent input predicates such as read/1 and get0/1 will henceforth use this stream.

7-3-10. set_output(+Stream)

This makes Stream the current output stream. Subsequent output predicates such as write/1 and put/1 will henceforth use this stream.

7-3-11. current_input(-Stream)

Stream is unified with the current input stream.

7-3-12. current_output(-Stream)

Stream is unified with the current output stream.

7-4. Stream Based Input and Output of Terms

The following predicates are alternatives to the ones described in section 7-1. These predicates take an explicit stream argument rather than using the current input and output streams. Opening files and associating them with streams is described in section 7-3.

read(S,T) like read/1, section 7-1-1, except that input is taken from stream S rather than from the current input stream.

write(S,T) like write/1, section 7-1-2, except that output is sent to stream S rather than to the current output stream.

display(S,T)
like display/1, section 7-1-3, except that output is sent to stream S rather than to the standard output stream (which normally means the terminal).

writeq(S,T) like writeq/1, section 7-1-4, except that output is sent to stream S rather than to the current output stream.

print(S,T) like print/1, section 7-1-5, except that output is sent to stream S rather than to the current output stream.

7-5. Stream Based Input and Output of Characters

The following predicates are alternatives to the ones described in section 7-2. These predicates take an explicit stream argument rather than using the current input and output streams. Opening files and associating them with streams is described in section 7-3.

get0(S,C) like get0/1, section 7-2-1, except that the character is input from stream S rather than from the current input stream.

get(S,C) like get/1, section 7-2-2, except that input is taken from stream S rather than from the current input stream.

skip(S,C) like skip/1, section 7-2-3, except that input is taken from stream S rather than from the current input stream.

put(S,C) like put/1, section 7-2-4, except the character is output on

stream S rather than on the current output stream.

nl(S) like nl/0, section 7-2-5, except that the linefeed is output on stream S rather than on the current output stream.

tab(S,N) like tab/1, section 7-2-6, except that the spaces are output on stream S rather than on the current output stream.

7-6. Reading the State of Opened Streams

7-6-1. character_count(+Stream,-N)

Stream is an open stream. N is unified with the total number of characters either read or written. A freshly opened stream has a character count of 0.

This predicate cannot be used on the standard input, output and error streams; it fails if this is attempted

7-6-2. line_count(+Stream,-N)

Stream is an open stream. N is unified with the total number of lines either read or written. A freshly opened stream has a line count of 1.

This predicate cannot be used on the standard input, output and error streams; it fails if this is attempted.

7-6-3. line_position(+Stream,-N)

Stream is an open stream. N is unified with the total number of characters either read or written on the current line. A fresh line has a line position of 0.

This predicate cannot be used on the standard input, output and error streams; it fails if this is attempted.

7-7. Prolog-20 Compatible File Handling

The following predicates are supplied for Prolog-20 compatibility. They specify an alternative, less powerful mechanism for dealing with files and streams.

The predicates seeing/1 and telling/1 are not fully Prolog-20 compatible. They are defined to be exactly the same as current_input/1 and current_output/1 respectively, that is they return a stream rather than a file name. The reason for this is that many programs use a seeing(X) followed by a see(X); consider the following example of a typical form of file processing:

```
process_file(F) :-
        seeing(Old),            % Save current file
        see(F),                 % Open file F
        repeat,
          read(T),              % Read a term
          process_term(T),      % Process the term
          T = end_of_file       % Loop back if not at end of file
        !,                      % Cut the backtrack loop
        seen,                   % Close the file
        see(Old).               % Restore old file
```

The seeing(Old)...see(Old) combination is intended to ensure that this clause leaves the current input unchanged. However, with the introduction of stream-based I/O, it is no longer the case that a file name uniquely identifies a stream: you can have more than one stream open on the same file. Thus if seeing returned a file name, then see(Old) could be ambiguous, and see resolves any such ambiguity arbitrarily. This arbitrary behavior is obviously undesirable, so seeing and telling now return streams which means that examples like the above will continue to work properly.

7-7-1. see(+X)

If X is a currently open input stream, then it is made the current input stream. If X is a stream which is not currently open for input then an error indication is given.

Otherwise, if X is an atom, it is taken to be a file name, and

- if there is no input stream currently associated with a file named X, then that file is opened for input and the new input stream thus created is made the current input stream. If it is not possible to open the file, an error indication may or may not occur depending on the state of the fileerrors flag (see section 7-3-6);
- if there is a single open input stream currently associated with a file of that name then it is made the current input stream;
- if there is more than one open input stream currently associated with a file of that name then one of them is chosen arbitrarily.

If X is not a stream or an atom, the predicate just fails.

It is important to remember to close streams when you have finished with them. Use seen/0 or close/1.

7-7-2. seeing(-S)

S is unified with the current input stream. This is exactly the same as current_input(S).

Note that this is different from the Prolog-20 definition of seeing which

returned a file name. If you need the file name you can get to it from the stream using current_stream/3 (section 7-3-5).

7-7-3. seen

Closes the current input stream. The current input stream is then set to be user_input, that is, the user's terminal.

7-7-4. tell(+F)

If X is a currently open output stream, then it is made the current output stream. If X is a stream which is not currently open for output then an error indication is given.

Otherwise, if X is an atom, it is taken to be a file name, and

- if there is no output stream currently associated with a file named X, then that file is opened for output and the new output stream thus created is made the current output stream. If it is not possible to open the file, an error indication may or may not occur depending on the state of the fileerrors flag (see section 7-3-6);

- if there is a single open output stream currently associated with a file of that name then it is made the current output stream;

- if there is more than one open output stream currently associated with a file of that name then one of them is chosen arbitrarily.

If X is not a stream or an atom, the predicate just fails.

It is important to remember to close streams when you have finished with them. Use told/0 or close/1.

7-7-5. telling(+S)

S is unified with the current output stream. This is exactly the same as current_output(S).

Note that this is different from the Prolog-20 definition of telling which returned a file name. If you need the file name you can get to it from the stream using current_stream/3 (section 7-3-5).

7-7-6. told

Closes the current output stream. The current output stream is then set to be user_output, that is, the user's terminal.

7-8. Prolog-20 Compatible Character I/O to Terminal

The predicates listed below are similar to those described in section 7-2 except that they always use the standard input and output, which normally refer to the user's terminal rather than to the current input or current output stream.

Given stream-based input/output, these predicates are really redundant. For example, you could write get0(user,C) instead of ttyget0(C).

7-8-1. ttyget0(-N)

Unifies N with the ASCII code of the next character input from the terminal.

7-8-2. ttyget(-N)

Unifies N with the ASCII code of the next non-space character from the current input stream. Space characters are all those with ASCII codes less than or equal to 32; this includes space, tab, linefeed and all control characters.

7-8-3. ttyskip(+N)

Skips over characters input from the terminal to the first occurrence of the character with ASCII code N. The most useful form of integer expression in this context is a single character in double quotes, for example, "a" which evaluates to 97, the ASCII code for the letter 'a'.

7-8-4. ttyput(+N)

Displays the ASCII character code of N on the terminal. N may be an integer expression. The most useful form of integer expression in this context is a single character in double quotes, for example, "a" which evaluates to 97, the ASCII code for the letter 'a'.

Note that the character is not necessarily displayed immediately; see ttyflush, section 7-8-7.

7-8-5. ttynl

Starts a new line on the terminal and flushes the terminal output buffer.

7-8-6. ttytab(+N)

Writes N spaces to the terminal. N may be an integer expression. It is an error if N is uninstantiated or if it contains an uninstantiated variable.

Note that the spaces are not necessarily output immediately; see ttyflush, section 7-8-7.

7-8-7. ttyflush

Flushes the terminal output buffer. Output to the terminal, using either ttyput or put, normally goes into an output buffer until a newline is output. Calling this predicate forces any characters in this buffer to be output immediately.

For flushing output that is not directed to the terminal, see flush_output/1 in section 7-3-8.

CHAPTER 8

ARITHMETIC

In Prolog, arithmetic is performed by certain built-in predicates which take arithmetic expressions as their arguments and evaluate them. Arithmetic expressions can evaluate to integers or floating-point numbers (floats).

Integers and floats each have advantages and disadvantages. The advantage of integers is that they precisely represent the numbers concerned. The disadvantages of integers are that they cannot represent every number you might wish to use (for example, integers cannot represent the number 3/2), and they may have a dynamic range that is more limited than that required by your application. Using floats eliminates these problems but does so at the expense of no longer representing precise mathematical quantities; floats are only approximations. As such, they should be used with caution.

Integers must be in the range -268435456 (-2^28) to 268435455 (2^28-1) inclusive. At the present time, there is NO range checking. (Integer arithmetic that yields numbers outside this range can be interpreted as modulo 2^29 in the above range.)

The exact range of floating point numbers is machine dependent. On all machines, 28 bits are used to represent a Prolog float. On the Sun, for example, this gives an approximate range of 1.0E-43 to 3.4E38, while on a VAX the range is 0.29*10^-38 to 1.7*10^38. The behavior on floating point overflow or underflow is also machine dependent.

8-1. Arithmetic Expressions

Arithmetic evaluation and testing is performed by predicates that take arithmetic expressions as arguments. An arithmetic expression is a term built from numbers, variables, and functors that represent arithmetic functions. These expressions are evaluated to yield an arithmetic result which may be either an integer or a float; the type is determined by the rules described below.

At the time of evaluation, each variable in an arithmetic expression must be bound to a number; if not, an error indication is given. In particular, note that a variable in an arithmetic expression may NOT be bound to another arithmetic expression.

The most common way to do arithmetic in Prolog is by using the built-in predicate is/2 (section 8-2).

Only certain functors are permitted in arithmetic expressions. These are listed below, together with a description of their arithmetical meaning. In the following, X and Y are considered to be arithmetic expressions.

X+Y — Results in the sum of X and Y. If both operands are integers, the result is an integer; otherwise, the result is a float.

X-Y — Results in the difference of X and Y. If both operands are integers, the result is an integer; otherwise, the result is a float.

X*Y — Results in the product of X and Y. If both operands are integers, the result is an integer; otherwise, the result is a float.

-X — Results in the negative of X. The type of the result, integer or float, is the same as the type of the operand.

X/Y — Results in the quotient of X and Y. The resultant number is always a float, regardless of the types of the operands X and Y.

X//Y — Results in the integer quotient of X and Y. X and Y must both be integers. The result is truncated to the nearest integer that is between it and 0.

integer(X) — Results in X if it is an integer. Otherwise, if it is a float, the result is the nearest integer that is between it and 0.

float(X) — Results in X if it is a float. Otherwise, if it is an integer, the result is the floating point equivalent.

The following bit-vector operations apply to integer arguments only. The operations will yield correct results on bit strings of 28 or fewer bits:

X/\Y — Results in the bitwise conjunction of X and Y.

X\/Y — Results in the bitwise disjunction of X and Y.

\(X) — Results in the complement of the bits in X.

X << Y — X is logically left-shifted Y places.

X >> Y — X is logically right-shifted Y places.

[X] — Results in X for numeric X. The reason for allowing this is that character strings in double quotes are just lists of ASCII codes; so that to use the ASCII code (97) for the letter "a" in an expression, you can just write "a".

8-2. -X is +Expression

Expression is evaluated as an arithmetic expression (see section 8-1), and the resulting number is unified with X. If Expression is not an arithmetic expression, or if it contains variables which are not bound to numbers, an error indication is given and the goal fails.

8-3. Arithmetic Comparison

Each of the following predicates evaluates both its arguments as arithmetic expressions and then compares the results. If one argument evaluates to an integer and the other to a float, then the integer is coerced to a float before the comparison is made.

Note that two floating point numbers are equal if and only if they have the same bit pattern.

8-3-1. +X =:= +Y

X and Y are each evaluated as arithmetic expressions. The goal succeeds if the results are equal.

8-3-2. +X =\= +Y

X and Y are each evaluated as arithmetic expressions. The goal succeeds if the results are not equal.

8-3-3. +X < +Y

X and Y are each evaluated as arithmetic expressions. The goal succeeds if the result of evaluating X is strictly less than the result of evaluating Y.

8-3-4. +X > +Y

X and Y are each evaluated as arithmetic expressions. The goal succeeds if the result of evaluating X is strictly greater than the result of evaluating Y.

8-3-5. +X =< +Y

X and Y are each evaluated as arithmetic expressions. The goal succeeds if the result of evaluating X is less than or equal to the result of evaluating Y.

8-3-6. +X >= +Y

X and Y are each evaluated as arithmetic expressions. The goal succeeds if the result of evaluating X is greater than or equal to the result of evaluating Y.

CHAPTER 9

LOOKING AT TERMS

This chapter describes predicates which allow you to examine the current instantiation state of a term.

9-1. var(?X)

Succeeds if X is currently uninstantiated ('var' is short for variable); otherwise fails. An uninstantiated variable is one which has not been bound to anything, except possibly another uninstantiated variable. Note that a structure with some components which are uninstantiated is not itself considered to be uninstantiated. Thus the command

```
:- var(foo(X,Y)).
```

always fails, despite the fact that X and Y are uninstantiated.

9-2. nonvar(?X)

Succeeds if X is currently instantiated; otherwise fails. This is the opposite of var.

9-3. atom(?X)

Succeeds if X is currently instantiated to an atom; otherwise fails.

9-4. integer(?X)

Succeeds if X is currently instantiated to an integer; otherwise fails.

9-5. float(?X)

Succeeds if X is currently instantiated to a float; otherwise fails.

9-6. number(?X)

Succeeds if X is currently instantiated to either an integer or a float; otherwise fails.

9-7. atomic(?X)

Succeeds if X is currently instantiated to an atom or number; otherwise fails.

9-8. functor(?Term,?Name,?Arity)

The principal functor of term Term has name Name and arity Arity.

There are two ways of using this predicate:

1. If Term is initially instantiated then

   - if Term is a compound term, then Name and Arity are unified with the name and arity of its principal functor.

   - if Term is atomic, then Name is unified with Term and Arity is unified with 0.

2. If Term is initially uninstantiated, Name and Arity must both be instantiated, and

   - if Arity is non-zero, then Name must be an atom and Term becomes instantiated to the most general term having the specified Name and Arity, that is, a term with distinct variables for all of its arguments.

   - if Arity is 0, then Name must be an atom or a number and it is unified with Term.

Some examples of calls to functor are:

```
| ?- functor(foo(a,b),N,A).

N = foo,
A = 2

| ?- functor(X,foo,2).

X = foo(_1,_2)
```

Note: "_1" and "_2" are variables. The term foo(_1,_2) is the "most general" term that has name foo and arity 2.

```
| ?- functor(X,foo,0).

X = foo
```

functor does not give error indications; it just fails if its arguments are inappropriate.

9-9. arg(+I,+T,-X)

Initially, I must be instantiated to a positive integer and T to a compound term. The result of the call is to unify X with the Ith argument of term T. (The arguments are numbered from 1 upwards.) If the initial conditions are not satisfied or I is out of range, the call just fails; it does not give an error indication.

9-10. ?X =.. ?Y

Y is a list whose head is the atom corresponding to the principal functor of X and whose tail is a list of the arguments of X. If X is uninstantiated, then Y must be instantiated either to a list of determinate length whose head is an atom, or to a list of length 1 whose head is an integer. Some examples of its use are:

```
| ?- foo(a,b) =.. X.

X = [foo,a,b]


| ?- X =.. [foo,a,b].

X = foo(a,b)


| ?- foo =.. X.

X = [foo]
```

=.. does not give any error indications; it just fails if its arguments are inappropriate.

=.. is sometimes pronounced "univ", after a predicate of that name and similar function in the original Marseilles implementation of Prolog.

9-11. name(?X,?L)

name(X,L) is a relation between an atomic object X and a list L which consists of the ASCII character codes for the printed representation of X. Initially, either X must be instantiated to an atomic object, or L must be instantiated to a list of character codes (containing no variables). Otherwise the call simply fails.

If X is initially instantiated to an atom or number, L will get bound to the list of character codes that make up its printed representation. Otherwise, if L is initially instantiated to a list of characters that corresponds to the correct syntax of a number (either integer or float), then X will get bound to that number; otherwise X will be instantiated to an atom containing exactly those characters. If neither argument is bound, the goal fails.

Here are some examples of its use:

```
| ?- name(foo,L).

L = [102,111,111]

| ?- name('Foo',L).

L = [70,111,111]

| ?- name(431,L).

L = [52,51,49]

| ?- name(X,[102,111,111]).

X = foo

| ?- name(X,[52,51,49]).

X = 431

| ?- name(X,"15.0e+12").

X = 1.5e+13
```

CHAPTER 10

COMPARISON OF TERMS

The following predicates are used to compare and order terms, rather than to evaluate or process them. For example, these predicates can be used to compare variables; however, they never instantiate those variables. These predicates should not be confused with arithmetic comparison (see section 8-3) or unification.

10-1. Standard Order on Terms

These predicates use a standard total order when comparing terms. The standard total order is: variables, numbers, atoms, complex terms. Within these categories, ordering is as follows.

- Variables are themselves put in a standard order. (Roughly, the oldest variable is put first; the order is not related to the names of variables.)
- Numbers are put in numeric order. Integers are considered to be less than their floating point equivalents.
- Atoms are put in alphabetical (ASCII) order.
- Complex terms are ordered first by arity, then by the name of the principal functor, then by the arguments (in left-to-right order).

For example, here is a list of terms in the standard order:

```
[ X, -9, 1, fie, foe, fum, X = Y, fie(0,2), fie(1,1) ]
```

The predicates for comparison of terms are described below.

10-2. ?T1 == ?T2

Succeeds if the terms currently instantiating T1 and T2 are literally identical (in particular, variables in equivalent positions in the two terms must be identical). For example, the question

```
| ?- T1 == T2.
```

fails (answers "no") because T1 and T2 are distinct uninstantiated variables. However, the question

```
| ?- T1 = T2, T1 == T2.
```

succeeds because the first goal unifies the two variables.

10-3. ?T1 \== ?T2

Succeeds if the terms currently instantiating T1 and T2 are not literally identical. For example, the question

```
| ?- T1 \== T2.
```

succeeds because T1 and T2 are distinct uninstantiated variables. However, the question

```
| ?- T1 = T2, T1 \== T2.
```

fails because the first goal unifies the two variables.

10-4. ?T1 @< ?T2

Succeeds if term T1 is before term T2 in the standard order.

10-5. ?T1 @> ?T2

Succeeds if term T1 is after term T2 in the standard order.

10-6. ?T1 @=< ?T2

Succeeds if term T1 is not after term T2 in the standard order.

10-7. ?T1 @>= ?T2

Succeeds if term T1 is not before term T2 in the standard order.

10-8. compare(?Op,?T1,?T2)

The result of comparing terms T1 and T2 is Op, where the possible values for Op are:

```
'='   if T1 is identical to T2,
'<'   if T1 is before T2 in the standard order,
'>'   if T1 is after T2 in the standard order.
```

Thus "compare(=,T1,T2)" is equivalent to "T1 == T2".

10-9. sort(+L1,-L2)

The elements of the list L1 are sorted into the standard order, and any identical (==) elements are merged, yielding the list L2. For example

```
| ?- sort([a,X,1,a(x),a,a(X)],L).
L = [X,1,a,a(X),a(x)]
```

(The time taken to do this is at worst order (N log N) where N is the length of the list.)

10-10. keysort(+L1,-L2)

The list L1 must consist of items of the form Key-Value. These items are sorted into order according to the value of Key, yielding the list L2. No merging takes place. For example

```
| ?- keysort([3-a,1-b,2-c,1-a,1-b],X).
X = [1-b,1-a,1-b,2-c,3-a]
```

(The time taken to do this is at worst order (N log N) where N is the length of the list.)

CHAPTER 11

LOOKING AT THE PROGRAM STATE

11-1. listing

Lists in the current output stream all the procedures in the current interpreted program. Procedures listed to a file can be consulted back. You could list the entire interpreted program to a file by the command

```
| ?- tell(file), listing, told.
```

Note that listing/0 does not work on compiled procedures unless they are declared to be dynamic.

11-2. listing(+Predicate)

If Predicate is just an atom, then the interpreted procedures for all predicates of that name are listed as for listing/0. The argument Predicate may also be a predicate specification of the form Name/Arity in which case only the clauses for the specified predicate are listed. Finally, it is possible for Predicate to be a list of predicate specifications of either type; for example,

```
| ?- listing([concatenate/3, reverse, go/0]).
```

Note that listing/1 does not work on compiled clauses unless they are declared to be dynamic.

If you are running under Emacs there is a facility for finding the source code definition for a specified compiled or interpreted procedure and reading it into an edit buffer. This is likely to be more helpful than listing/1 in most cases. See "^X .", section 22-2.

11-3. current_atom(?Atom)

If Atom is initially uninstantiated, this predicate generates (through backtracking) all currently known atoms. Otherwise it succeeds if and only if Atom is an atom.

11-4. current_predicate(?Name,?Term)

True if Name is the name of a user-defined predicate and Term is the most general term corresponding to that predicate. For example, if you have a foo/1 and a foo/3 in your program you might get:

```
| ?- current_predicate(foo,T).

T = foo(_116) ;

T = foo(_116,_117,_118) ;

no
```

The goal

```
| ?- current_predicate(Name,Term).
```

can be used to backtrack through every predicate in your program.

CHAPTER 12

LOOKING AT AND CONTROLLING THE EXECUTION STATE

12-1. Control C interrupts

At any time, Prolog's execution can be interrupted by typing ^C. The following prompt is then displayed:

```
Prolog interruption (h for help)?
```

If you then type 'h', followed by Return, you will get a list of the possible responses to this prompt:

```
Prolog interrupt options:
    c    continue - do nothing
    t    trace    - debugger will start creeping
    d    debug    - debugger will start leaping
    a    abort    - cause a Prolog abort
    e    exit     - irreversible exit from Prolog
    h    help     - this list
```

The 'trace' option will cause you to enter the debugger the next time control passes to an interpreted procedure. (You can then use the 'g' option of the debugger to see what Prolog is doing.) This will not help if you are stuck in a loop in compiled code; in this case the 'd' option can be useful if you have spypoints set in your compiled code and debugging is currently off.

12-2. halt

Causes an irreversible exit from Prolog.

12-3. break

Causes the current execution to be interrupted and the message "[Break (level 1)]" to be displayed, followed by a top level prompt ('| ?- '). The system is then ready to accept input as though it were at top level. If another call of break is encountered, it moves up to level 2, and so on. To close a break level and resume the execution which was suspended, type the end-of-file character ^D (or ^X^D if running under Emacs). Execution of the interrupted program is then resumed with the call to break/0 succeeding. Alternatively, the suspended execution can be abandoned by calling the built-in predicate abort/0.

The break facility is often used via the debugging option 'b'.

12-4. abort

Abandons (aborts) the current execution and returns to top level. All break levels (suspended execution states created by break/0) are destroyed. This is a fairly drastic predicate and is normally only used when some error condition has occurred and there is no way of carrying on, or when debugging.

The abort facility is often used via the debugging option 'a' and also via the ^C interrupt option 'a'.

12-5. ancestors(-L)

Unifies L with a list of ancestor goals for the current clause. The list starts with the parent goal and ends with a goal which was typed at top level. If a goal has a succession of compiled (and not dynamic) ancestors, then only one of these (the highest in the calling hierarchy) will appear in the list.

ancestors/1 is most often used in debugging via the debugger option 'g'.

12-6. subgoal_of(?S)

Equivalent to the sequence of goals:

```
ancestors(L), member(S,L).
```

where the predicate **member** (not a built-in predicate) successively matches its first argument with each of the elements of its second argument. It may be defined by

```
member(X,[X|L]).
member(X,[_|L]) :- member(X,L).
```

12-7. maxdepth(+D)

D should be a positive integer which specifies the maximum number of nested INTERPRETED procedure calls, beyond which the interpreter will cause a trap to the debugger. The usual debugger options can be used when the interpreter traps. Top level has depth zero. maxdepth is useful for guarding against loops in untested programs. The default setting is maxdepth(100000).

NOTE: Calls to compiled procedures are not included in the depth and calls to interpreted code from compiled code will start again from a depth of zero.

12-8. depth(-D)

Unifies D with the current depth, that is, the number of currently active interpreted procedure calls.

NOTE: This predicate is only applicable in interpreted procedures. Calls to compiled procedures are not included in the depth and calls to interpreted code from compiled code will start again from a depth of zero.

CHAPTER 13

MEMORY USE

Quintus Prolog uses three data areas: program space, local stack space and global data space. Each of these areas is automatically expanded if it overflows, with the other areas being shifted if necessary to allow this.

The program space contains compiled and interpreted code, recorded terms and atoms. The space occupied by interpreted code and recorded terms is recovered when it is no longer needed, but currently the space occupied by compiled code and atoms is not recovered.

The global stack space contains the global stack and the trail which grow inward toward one another. The global stack contains all the data structures constructed in an execution of the program, and the trail contains references to all the variables which need to be reset when backtracking occurs. Both of these areas grow with forwards execution and shrink on backtracking.

The local stack contains all the control information and variable bindings needed in a Prolog execution. Space on the local stack is reclaimed on determinate success of predicates and by tail recursion optimization, as well as on backtracking.

13-1. trimcore

Reduces free space on all the working areas as much as possible and releases space no longer needed. During a computation, Prolog automatically expands its working storage as needed, and it keeps the space it grabs until trimcore is called. The interpreter automatically calls trimcore after each directive at top-level.

13-2. statistics

Displays on the terminal statistics relating to memory usage and run time, including information about which areas of memory have overflowed and how much time has been spent expanding them.

The output from statistics/0 looks like this:

```
memory (total)        426712 bytes
   program space      287280 bytes:   261848 in use,    25432 free
   global space         6664 bytes:     3036 in use,     3628 free
      global stack                     3020 bytes
      trail                              16 bytes
   local stack          5532 bytes:     1592 in use,     3940 free

 0.0 sec. for 0 program, 0 global and 0 local space overflows
 0.583 sec. runtime
```

Note the use of indentation to indicate sub-areas. That is, memory contains the program space, global space and local stack, and the global space contains the global stack and trail.

See section 13-3 for how to obtain individual statistics.

13-3. statistics(?Keyword,-List)

statistics(Keyword,List) is usually used with Keyword instantiated to a keyword such as 'runtime' and List unbound. The predicate then binds List to a list of statistics related to the keyword. This predicate is used in programs which depend on current runtime statistical information for their control strategy, and in programs which choose to format and output their own statistical summary.

The keys and values for statistics(Keyword,List) are summarized below. The keywords 'core' and 'heap' are included to retain compatibility with Prolog-20. Times are given in milliseconds and sizes are given in bytes.

Keyword	List
runtime	cpu time used by Prolog, cpu time since last call to statistics
memory	total virtual memory (,0)
core	(same as memory)
program	program size (in use, free)
heap	(same as program)
global_stack	size of global stack (in use, free)
local_stack	size of local stack (in use, free)
trail	size of trail (,0)
garbage_collection	number of GCs, freed bytes, time spent
stack_shifts	number of global stack area shifts, number of local stack area shifts, time spent shifting

For the keywords 'memory' and 'trail' the second element of the returned list is always 0. This is for Prolog-20 compatibility only, 0 being the most appropriate value in this system for the quantities which would be returned here in Prolog-20.

Currently, the garbage_collection keyword always returns a list of 0's because garbage collection of the global stack is not yet available.

To see an example of the use of each of these keywords, type

```
| ?- statistics(K,L).
```

and then repeatedly type ';' to backtrack through all the possible keywords.

CHAPTER 14

SAVING THE PROGRAM OR THE EXECUTION STATE

14-1. save_program(+File)

Saves the current program state, that is, all compiled and interpreted procedures, into the specified File. File should be an atom representing a file name.

A saved program state can be activated either by running it directly from the operating system or else by using the predicate restore/1 (section 14-4). In either case, after the reactivation of the program you will be at Prolog's top level. The only difference is that when the saved state is run directly from the operating system, a search is made in your home directory for a file named 'prolog.ini', and if one is found it is consulted. This does not happen after a restore.

If File is not an atom, the goal simply fails. If it is not possible to perform the save operation for some external reason, such as the file having the wrong protection, an error indication is given.

14-2. save(+File)

Saves the current execution state, that is, all compiled and interpreted procedures and all data areas, into the specified File. File should be an atom representing a file name.

A saved execution state can be activated either by running it directly from the operating system or else by using the predicate restore/1 (section 14-4). In either case, after the reactivation of the program, execution continues exactly as if the save goal had just succeeded.

For example

```
| ?- save(foo), write('FOO HERE').
[ Prolog state saved into foo ]
FOO HERE
yes
| ?- restore(foo).

Quintus Prolog Release 1.0
Copyright (C) 1985, Quintus Computer Systems, Inc.
FOO HERE
yes
| ?-
```

If File is not an atom, the save(File) simply fails. If it is not possible to perform the save operation for some external reason, such as the file having the wrong protection, an error indication is given.

14-3. save(+File,-Return)

Saves the current execution state in File just as save(File), but in addition unifies Return to 0 or 1 depending on whether the return from the call occurs in the original save operation or through a reactivation of the saved state (either by running the saved state as a program or else via restore/1; see section 14-4).

For example

```
| ?- save(foo,X).
[ Prolog state saved into foo ]

X = 0

| ?- restore(foo).

Quintus Prolog Release 1.0
Copyright (C) 1985, Quintus Computer Systems, Inc.

X = 1

| ?-
```

The purpose of this predicate is to allow you to specify different behavior after a restore than after the save itself. For example, if you have a program which you want to run immediately when a saved state is reactivated, but which you do not want to run when you create the saved state, then you might type

```
| ?- save(prog,1), go.
```

where go/0 is the entry point to your program. This will create the saved state and then fail, without entering your program.

If File is not an atom, save(File,_) simply fails. If it is not possible to perform the save operation for some external reason, such as the file having the wrong protection, an error indication is given.

14-4. restore(+File)

This predicate is used to reactivate the saved state in the file named File which must have been created by save_program/1 (section 14-1), save/1 (section 14-2) or save/2 (section 14-3). The effect of the restore depends on which of these predicates was used to create the saved state; refer to the definitions of those predicates for details.

The effect of a restore can also be achieved by running a saved state directly from the operating system. One situation in which you might find restore preferable is in debugging a program which uses asserts and retracts; the combination of save and restore enables you to save interesting states of the execution and then go back to exactly that state a bit later (without having

to exit from Prolog).

If File is not an atom, restore(File) simply fails. If it is not possible to perform the restore operation because the specified file is not a saved state, or does not exist, or is protected, or for some similar reason, a fatal error indication is given and an exit from Prolog occurs.

14-5. reinitialise

This predicate can be used at any time to force Prolog's normal initialization behavior. It aborts any current computation (like abort/0, see section 12-4), and then looks for a file called 'prolog.ini' in your home directory. If such a file is found, it is consulted.

This predicate is not particularly useful now that there is save_program (section 14-1). It is provided for Prolog-20 compatibility only. If you do use it, note that it is spelled with an 's'.

CHAPTER 15

DEBUGGING

This chapter describes the built-in predicates which are concerned with debugging. The debugger is more fully described in the Quintus Prolog User's Guide.

15-1. debug

Turns the debugger on and sets the top-level state to 'debug'. Turning the debugger on means that it will stop at the next spypoint encountered in the current execution. Setting the top-level state to 'debug' means that debugging will be on whenever you type a question. That is, the debugger will start by not showing any goals and not stopping until it reaches a spypoint, that is, it will "leap". Once you reach a spypoint you will have a number of options including those of "creeping" forwards, or "leaping" once again.

The effect of this predicate can also be achieved by typing the letter 'd' after a ^C interrupt (see section 12-1).

Note that the top-level state of the debugger is ignored if you precede a goal with a ':-', and also for goals appearing in files being consulted or compiled. In these cases debugging is (initially) off.

15-2. trace

Turns the debugger on and starts it "creeping", and sets the top-level state to 'trace'. The debugger will start showing goals as soon as an interpreted procedure is reached, and it will stop to allow you to interact as soon as it reaches a leashed port (see leash/1, section 15-10). Setting the top-level state to 'trace' means that every time you type a question, the debugger will start by creeping.

The effect of this predicate can also be achieved by typing the letter 't' after a ^C interrupt (see section 12-1).

Note that the top-level state of the debugger is ignored if you precede a goal with a ':-', and also for goals appearing in files being consulted or compiled. In these cases debugging is (initially) off.

15-3. nodebug

Turns the debugger off (nodebug and notrace are equivalent). Turning the debugger off does NOT remove any spypoints. Spypoints will remain where they were set, although they will have no effect while the debugger is off. When the debugger is turned on again, the spypoints will again take effect. If you want to remove all your spypoints, use nospyall (section 15-8).

15-4. notrace

Turns the debugger off (nodebug and notrace are equivalent). Turning the debugger off does NOT remove any spypoints. Spypoints will remain where they were set, although they will have no effect while the debugger is off. When the debugger is turned on again, the spypoints will again take effect. If you want to remove all your spypoints, use nospyall (section 15-8).

15-5. debugging

Displays on the terminal information about the current debugging state. It shows

- the top-level state of the debugger which is one of

  debug — The debugger is on but will not show anything or stop for user interaction until a spypoint is reached.

  trace — The debugger is on and will show everything. As soon as you type a goal, you will start seeing a debugging trace. After printing each trace message, the debugger may or may not stop for user interaction: this depends on the type of leashing in force (see below).

  off — The debugger is off.

  The top-level state can be controlled by the predicates debug/0, nodebug/0, trace/0 and notrace/0.

- the type of leashing in force; when the debugger prints a message saying that it is passing through a particular port (one of 'call', 'exit', 'redo' and 'fail') of a particular procedure, it stops for user interaction only if that port is leashed. The predicate leash/1 can be used to select which of the four ports you want to be leashed.

- the action to be taken on undefined predicates. This is either 'trace' which means that calling an undefined predicate traps to the debugger, or 'fail' meaning that such calls just fail. This option can be controlled using the predicate unknown/2.

- all the current spypoints. Spypoints are controlled by the predicates spy/1, nospy/1 and nospyall/0.

15-6. spy +X

Sets spypoints on all the predicates represented by X, which is either a single predicate specification, or a list of such specifications. A predicate specification is either of the form Name/Arity, (for example, member/2, foo/0, hello/27); or it is of the form Name, (for example, member, foo, hello) which represents all the currently defined predicates with the name Name (there may

be several predicates with the same name but different arities).

If you use the form Name but there are no clauses for this predicate (of any arity), then a warning message will be displayed and no spypoint will be set. If you really want to place a spypoint on a currently undefined procedure, then you must use the full form Name/Arity; you will still get a warning message but the spypoint will be set in this case.

If debugging is off, calling spy/1 will set it to 'debug', so that the debugger will stop as soon as it reaches a spypoint.

You can put spypoints on compiled procedures, as well as interpreted ones, but there is a space and time penalty incurred. This penalty does not go away when you simply turn off debugging (using nodebug); you have to remove the spypoints explicitly (using nospy or nospyall).

15-7. nospy +X

Removes spypoints on all the predicates represented by X. The possible forms for X are the same as for spy/1.

To remove all spypoints, see nospyall/0, section 15-8.

15-8. nospyall

All spypoints are removed. This is the only way to remove all your spypoints at once, since turning off debugging with nodebug does NOT remove spypoints; they remain in place and are reactivated if the debugger is turned back on with trace or debug.

15-9. unknown(-OldAction,+NewAction)

Unifies OldAction with the current action on unknown procedures, and then sets the current action to NewAction. This action determines whether or not the interpreter will catch calls to undefined procedures and is one of:

trace	Undefined procedures will trap to the debugger
fail	Undefined procedures will just fail

The default action is 'trace'. Note that

```
| ?- unknown(Action,Action).
```

just returns the current Action without changing it.

Procedures which are known to be dynamic are never trapped by the debugger; it is assumed that these procedures are meant to fail when there are no clauses for them. For more information on dynamic procedures, see section 16-1.

15-10. leash(+Mode)

Sets the leashing mode to Mode. The purpose of leash is to let you speed up single-stepping (creeping) through a program by telling the debugger that it does not always need to wait for user input after printing a trace message.

The leashing mode only applies to procedures which do not have spypoints on them, and it determines which ports of such procedures are leashed. By default, all four ports are leashed. On arrival at a leashed port the debugger will stop and allow you to look at the execution state and decide what to do next. At unleashed ports, the goal is displayed but program execution does not stop to allow user interaction.

Mode should either be the atom 'all', for leashing on all four ports, or else a list containing zero or more of the atoms 'call', 'exit', 'redo' and 'fail'. Thus, for example

```
| ?- leash([]).
```

turns off all leashing; now when you creep you will get an exhaustive trace but no opportunity to interact with the debugger. And

```
| ?- leash([call,redo]).
```

puts leashing on the Call and Redo ports. When creeping, the debugger will now stop at every Call and Redo port to allow you to interact.

In Prolog-20, a different form of argument was used for leash/1. This form is also supported by Quintus Prolog, but it is not recommended.

CHAPTER 16

MODIFICATION OF THE DATABASE

The assert and retract family of predicates described below enable you to modify a Prolog program by adding or deleting clauses while it is running. These predicates should not be over-used. Often people who are experienced with other programming languages have a tendency to think in terms of global data structures, as opposed to data structures which are passed as procedure arguments, and hence they make too much use of assert and retract. This leads to less readable and less efficient programs.

An interesting question in Prolog is what happens if a procedure modifies itself, by asserting or retracting a clause, and then fails. On backtracking, do new clauses added to the bottom of the procedure get used or not? Different Prolog implementations do different things. In Quintus Prolog, as in the Prolog-20 interpreter, changes to a procedure take effect immediately: any clauses which you assertz (add to the end of the procedure) will be encountered on backtracking, and any clauses which you retract will not be seen on backtracking.

16-1. Dynamic and Static Procedures

All procedures in Prolog fall into one of two categories: static or dynamic. Dynamic procedures can be modified by adding or deleting individual clauses. In contrast, static procedures can only be changed by completely redefining them using consult or compile.

If a procedure is first defined by being consulted or compiled, it is static by default. If you need to be able to add, delete or inspect individual clauses of a procedure you must make the procedure dynamic. There are two ways of making a procedure dynamic:

- If the procedure is to be compiled or consulted, then its definition must be preceded, in the same file or editor buffer, by a dynamic declaration.

- If the procedure is to be created by assertions only, then the first assert, retract or clause operation on the procedure will make it dynamic.

A dynamic declaration is a goal clause of the form

```
:- dynamic Pred.
```

appearing in a file to be consulted or compiled. Pred must be a predicate specification of the form Name/Arity or else a sequence of such specifications separated by commas. For example,

```
:- dynamic exchange_rate/3, spouse_of/2, gravitational_constant/1.
```

(dynamic is a built-in prefix operator.) Note that the ':-' preceding the word 'dynamic' is essential; if it were omitted you would get an error because it would appear that you were trying to define a clause for the predicate dynamic/1. However, dynamic/1 is not really a built-in predicate, and it may only be used in declarations.

When a dynamic declaration is encountered in a file being consulted or compiled, it is considered to be a part of the redefinition of the predicates in its argument. Thus if you consult a file containing only the following

```
:- dynamic hello/0.
```

the effect will be to remove any previous definition of hello/0 from the database, as well as making the predicate dynamic.

Although you can simultaneously declare several procedures to be dynamic, as shown above, it is recommended that you put a separate dynamic declaration for each procedure immediately before the clauses for that procedure. This way, if you use the editor interface to reconsult or recompile that procedure you won't forget to include its dynamic declaration. (Reconsulting or recompiling a procedure without its dynamic declaration would make it static.)

It is often useful to have a dynamic declaration for a procedure even if it is to be created only by assertions. This helps another person to understand your program, since it makes clear the fact that there are no pre-existing clauses for this procedure, and it also avoids the possibility of Prolog stopping to tell you there are no clauses for this procedure if you should happen to call it before any clauses have been asserted. This is because unknown procedure catching (see section 15-9) does not apply to dynamic procedures; it is presumed that a call to a dynamic procedure should just fail if there are no clauses for it.

Dynamic procedures are actually implemented by interpretation, even if they are included in a file which is compiled. This means that they are executed slower than if they were static, and also that they are visible to the debugger and to listing/0.

16-2. assert(+Clause)

Adds the specified Clause to the database. The relative position of this clause with respect to other clauses for the same predicate is arbitrary. If you want to insert this clause in a particular position, use asserta or assertz instead.

Clause must be instantiated to a non-numeric value. If it is of the form Head :- Body then Head must also be instantiated to a non-numeric value. An error indication is given if these conditions are not satisfied, or if there is a static definition for the specified predicate.

Syntactic note: If you want to write a term of the form Head :- Body as the argument to assert, you must put it in parentheses because the operator precedence of the functor :- /2 is greater than 1000 (see section 2-4). For

example,

```
| ?- assert(foo:-bar).
```

will cause a syntax error; instead you should type

```
| ?- assert((foo:-bar)).
```

16-3. asserta(+Clause)

Adds the specified Clause to the database as the first clause of its procedure.

Clause must be instantiated to a non-numeric value. If it is of the form Head :- Body then Head must also be instantiated to a non-numeric value. An error indication is given if these conditions are not satisfied, or if there is a static definition for the specified predicate.

16-4. assertz(+Clause)

Adds the specified Clause to the database as the last clause of its procedure.

Clause must be instantiated to a non-numeric value. If it is of the form Head :- Body then Head must also be instantiated to a non-numeric value. An error indication is given if these conditions are not satisfied, or if there is a static definition for the specified predicate.

16-5. clause(+Head,?Body)

Searches the database for a clause whose head matches Head and whose body matches Body. This predicate is non-determinate; it can be used to backtrack through all the clauses matching a given Head and Body. It fails when there are no (further) matching clauses in the database.

For the purpose of this matching, unit clauses (clauses with no body) are treated as if they had a body consisting of the single goal true. For example,

```
| ?- asserta(foo(a)).

yes
| ?- clause(foo(X),Body).

X = a,
Body = true
```

Head must be instantiated to a term of which the principal functor is a dynamic predicate. An error indication is given if the principal functor of Head is a static predicate. If there are no clauses for the predicate, then

it is made dynamic and the goal fails.

16-6. retract(+Clause)

Erases the first clause in the database that matches Clause. Clause must be instantiated to a term corresponding with a dynamic procedure. retract is non-determinate. If control backtracks into the call to retract, successive clauses matching Clause are erased. If no clauses match, the call to retract fails.

retract treats unit clauses as if they each had the single goal true as their body. Thus

```
| ?- retract((foo(X) :- Body)), fail.
```

is guaranteed to retract all the clauses for foo/1, including any unit clauses, providing of course that foo/1 is dynamic.

Since retract is non-determinate, it can be a good idea, if you only want to retract a single clause, to use a cut to eliminate the choice point so generated.

The space occupied by a clause which is retracted is reclaimed. The reclamation does not necessarily happen immediately, but it is not delayed until backtracking past the call of retract, as in some implementations.

16-7. abolish(+Name,+Arity)

Removes the procedure specified by Name and Arity from the database. The procedure may be either static or dynamic.

This predicate gives no error indications, and it always succeeds; it has no effect when its arguments do not in fact specify an existing procedure in the database.

Note that the space occupied by interpreted (and compiled dynamic) clauses is reclaimed after the abolish, but space occupied by static compiled clauses is not reclaimed.

CHAPTER 17

DATABASE REFERENCES

17-1. Definition

A database reference is a term which uniquely identifies a clause or recorded term (see chapter 18) in the database. The purpose of providing database references is only to increase efficiency in programs which have to do a lot of manipulation of the database. Using a database reference to a clause can save repeated searches using clause/2.

The internal form of a database reference is irrelevant, but for the benefit of the curious it is

```
'$ref'(Pointer,Counter)
```

where Pointer is a machine pointer to the clause representation and Counter is a unique integer. This representation of database references may change in the future; programs should not rely on it.

Consistency checking is done whenever a reference is used; any attempt to use a reference to a clause which has been retracted will simply fail.

There is no restriction on the use of references. References may be included in asserted clauses.

17-2. assert(+Clause,-Ref)

Equivalent to assert/1 but returns Ref which is the database reference which uniquely identifies the newly asserted clause.

17-3. asserta(+Clause,-Ref)

Equivalent to asserta/1 but returns Ref which is the database reference which uniquely identifies the newly asserted clause.

17-4. assertz(+Clause,-Ref)

Equivalent to assertz/1 but returns Ref which is the database reference which uniquely identifies the newly asserted clause.

17-5. clause(?Head,?Body,?Ref)

This predicate has two different modes of use, depending on whether or not the database reference Ref of the clause is given. If Ref is specified, then Head:-Body is unified with the clause identified by Ref. (If this clause is a

unit clause, Body is unified with true.)

If Ref is not given, then this is exactly like clause/2 except that the database reference is returned.

17-6. erase(+Ref)

The dynamic clause (or recorded term - see chapter 18) which is referenced by Ref is erased from the database. Any subsequent attempt to use the specified reference will fail.

If Ref is not a database reference to an existing clause or recorded term, then erase simply fails.

17-7. instance(+Ref,-Term)

A (most general) instance of the clause or recorded term indicated by the database reference Ref is unified with Term. If the reference is to a unit clause C then Term is unified with C:-true.

If Ref is not a database reference to an existing clause or recorded term, then instance simply fails.

CHAPTER 18

THE INTERNAL DATABASE

The following predicates are provided purely for efficiency. Their semantics can be understood by imagining that they are defined by the following clauses:

```
recorda(Key,Term,Ref) :- asserta('$recorded'(Key,Term),Ref).
recordz(Key,Term,Ref) :- assertz('$recorded'(Key,Term),Ref).
recorded(Key,Term,Ref) :- clause('$recorded'(Key,Term),_,Ref).
```

The only reason that these predicates are not implemented in something like the above manner is that recorded would have to do a linear search through all the clauses for '$recorded'/2. In the actual implementation there is a fast look-up using the Key.

18-1. recorda(+Key,?Term,-Ref)

The term Term is recorded in the internal database as the first item for the key Key, and a database reference to the newly recorded term is returned as Ref. The Key must be instantiated (not to a float); otherwise the goal fails. If Key is a compound term, only its principal functor is significant. That is, foo(1) would be considered to represent the same Key as foo(n).

18-2. recordz(+Key,?Term,-Ref)

The term Term is recorded in the internal database as the last item for the key Key, and a database reference to the newly recorded term is returned as Ref. The Key must be instantiated (not to a float); otherwise the goal fails. If Key is a compound term, only its principal functor is significant. That is, foo(1) would be considered to represent the same Key as foo(n).

18-3. recorded(+Key,?Term,?Ref)

The internal database is searched for a term recorded under the key Key which unifies with Term and whose database reference unifies with Ref.

This predicate is non-determinate, meaning that it can be used to backtrack through all the matching terms recorded under the specified key. Thus, if you only want to match a single term you should use a cut to prevent any of this backtracking.

Key must be instantiated (not to a float); otherwise the call fails. If Key is a compound term, only its principal functor is significant. That is, foo(1) would be considered to represent the same Key as foo(n).

CHAPTER 19

SETS: COLLECTING ALL THE SOLUTIONS TO A GOAL

When there are many solutions to a goal, and when you want a list of all those solutions, you can write a program which repeatedly backtracks into that goal to get another solution. In order to collect all the solutions together, it is necessary to use the database (via assert) to hold the solutions as they are generated, because backtracking to redo the goal would undo any list construction that had been done after satisfying the goal.

Instead of writing this backtracking loop yourself, it is convenient to use one of the built-in predicates setof/3 and bagof/3 which are described below. These provide a nice logical abstraction, whereas if you write the backtrack loop yourself the need for explicit side-effects (asserts) destroys the declarative interpretation of the code.

19-1. setof(?Template,+Goal,-Set)

Read this as "Set is the set of all instances of Template such that Goal is provable, where that set is non-empty". The term Goal specifies a goal to be called as if by call/1. Set is a set of terms represented as a list of those terms, without duplicates, in the standard order for terms (see section 10). If there are no instances of Template such that Goal is satisfied then the predicate fails.

The variables appearing in the term Template should not appear anywhere else in the clause except within the term Goal. Obviously, the set to be enumerated should be finite, and should be enumerable by Prolog in finite time. It is possible for the provable instances to contain variables, but in this case the list Set will only provide an imperfect representation of what is in reality an infinite set.

If there are uninstantiated variables in Goal which do not also appear in Template, then setof can succeed non-determinately, generating alternative values for Set corresponding to different instantiations of the free variables of Goal. (It is to allow for such usage that the set Set is constrained to be non-empty.) For example, the call:

```
| ?- setof(X, X likes Y, S).
```

might produce two alternative solutions via backtracking:

```
Y = beer,   S = [dick, harry, tom]
Y = cider,  S = [bill, jan, tom ]
```

The call:

```
| ?- setof((Y,S), setof(X, X likes Y, S), SS).
```

would then produce:

```
SS = [ (beer,[dick,harry,tom]), (cider,[bill,jan,tom]) ]
```

Variables occurring in Goal will not be treated as free if they are explicitly bound within Goal by an existential quantifier. An existential quantification is written:

```
Y^Q
```

meaning "there exists a Y such that Q is true", where Y is some Prolog variable. For example:

```
| ?- setof(X, Y^(X likes Y), S).
```

would produce the single result:

```
X = [bill, dick, harry, jan, tom]
```

in contrast to the earlier example.

19-2. bagof(?Template,+Goal,-Bag)

This is exactly the same as setof except that the list (or alternative lists) returned will not be ordered, and may contain duplicates. The effect of this relaxation is to save considerable time and space in execution.

19-3. X^P

This is recognized as meaning "there exists an X such that P is true", and is treated as equivalent to simply calling P. The use of this explicit existential quantifier outside the setof and bagof constructs is superfluous.

CHAPTER 20

GRAMMAR RULES

20-1. Definite Clause Grammars

Prolog's grammar rules provide a convenient notation for expressing definite clause grammars, which are useful for the analysis of both artificial and natural languages.

Definite clause grammars are an extension of the well-known context-free grammars. A grammar rule in Prolog takes the general form

head —> body.

meaning "a possible form for head is body". Both body and head are sequences of one or more items linked by the standard Prolog conjunction operator ','.

Definite clause grammars extend context-free grammars in the following ways:

- A non-terminal symbol may be any Prolog term (other than a variable or integer).

- A terminal symbol may be any Prolog term. To distinguish terminals from non-terminals, a sequence of one or more terminal symbols is written within a grammar rule as a Prolog list. An empty sequence is written as the empty list '[]'. If the terminal symbols are ASCII character codes, such lists can be written (as elsewhere) as strings. An empty sequence is written as the empty list, '[]' or '""'.

- Extra conditions, in the form of Prolog procedure calls, may be included in the right-hand side of a grammar rule. Such procedure calls are written enclosed in curly brackets ('{' and '}').

- The left-hand side of a grammar rule consists of a non-terminal, optionally followed by a sequence of terminals (again written as a Prolog list).

- Alternatives may be stated explicitly in the right-hand side of a grammar rule, using the disjunction operator ';' as in Prolog.

- The cut symbol may be included in the right-hand side of a grammar rule, as in a Prolog clause. The cut symbol does not need to be enclosed in curly brackets.

20-2. An Example

As an example, here is a simple grammar which parses an arithmetic expression (made up of digits and operators) and computes its value.

```
expr(Z) --> term(X), "+", expr(Y), {Z is X + Y}.
expr(Z) --> term(X), "-", expr(Y), {Z is X - Y}.
expr(X) --> term(X).

term(Z) --> number(X), "*", term(Y), {Z is X * Y}.
term(Z) --> number(X), "/", term(Y), {Z is X / Y}.
term(Z) --> number(Z).

number(C) --> "+", number(C).
number(C) --> "-", number(X), {C is -X}.
number(X) --> [C], {"0"=<C, C=<"9", X is C - "0"}.
```

In the last rule, C is the ASCII code of some digit.

This grammar can now be used to parse and evaluate an expression by means of the built-in predicate **phrase/2**. For example,

```
| ?- phrase(expr(Z),"-2+3*5+1").

Z = 14
```

20-3. Translation of Grammar Rules into Prolog Clauses

Now, in fact, grammar rules are merely a convenient abbreviation for ordinary Prolog clauses. Each grammar rule is translated into a Prolog clause as it is consulted or compiled. This translation is described below.

The procedural interpretation of a grammar rule is that it takes an input list of symbols or character codes, analyzes some initial portion of that list, and produces the remaining portion (possibly enlarged) as output for further analysis. The arguments required for the input and output lists are not written explicitly in a grammar rule, but are added when the rule is translated into an ordinary Prolog clause. For example, a rule such as

```
p(X) --> q(X).
```

translates into

```
p(X,S0,S) :- q(X,S0,S).
```

If there is more than one non-terminal on the right-hand side, as in

```
p(X,Y) --> q(X), r(X,Y), s(Y).
```

then corresponding input and output arguments are identified, as in

```
p(X,Y,S0,S) :-
    q(X,S0,S1),
    r(X,Y,S1,S2),
    s(Y,S2,S).
```

Terminals are translated using the built-in predicate 'C'(S1,X,S2), read as

"point S1 is connected by terminal X to point S2", and defined by the single clause

```
'C'([X|S],X,S).
```

(This predicate is not normally useful in itself; it has been given the name upper-case 'c' simply to avoid using up a more useful name.) Then, for instance

```
p(X) --> [go,to], q(X), [stop].
```

is translated into

```
p(X,S0,S) :-
    'C'(S0,go,S1),
    'C'(S1,to,S2),
    q(X,S2,S3),
    'C'(S3,stop,S).
```

Extra conditions expressed as explicit procedure calls naturally translate as themselves. For example,

```
p(X) --> [X], {integer(X), X>0}, q(X).
```

translates to

```
p(X,S0,S) :-
    'C'(S0,X,S1),
    integer(X),
    X>0,
    q(X,S1,S).
```

Similarly, a cut is translated literally.

Terminals on the left-hand side of a rule translate into 'C'/3 goals with the first and third arguments reversed. For example,

```
is(N), [not] --> [aint].
```

becomes

```
is(N,S0,S) :-
    'C'(S0,aint,S1),
    'C'(S,not,S1).
```

Disjunction has a fairly obvious translation. For example,

```
args(X,Y) -->
        dir(X), [to], indir(Y) |
        indir(Y), dir(X).
```

translates to

```
args(X,Y,S0,S) :-
    (   dir(X,S0,S1),
        'C'(S1,to,S2),
        indir(Y,S2,S)
    ;   indir(Y,S0,S1),
        dir(X,S1,S)
    ).
```

20-4. Grammar-Related Built-in Predicates

20-4-1. expand_term(+T1,-T2)

When a program is compiled (or consulted), some of the terms read are transformed before being compiled (or stored as interpreted clauses). This transformation is done by calling expand_term/2. Thus expand_term is usually called by the built-in predicates compile and consult and not directly by user programs.

The normal use of expand_term(T1,T2) is to translate grammar rules into ordinary Prolog clauses. If T1 is a grammar rule, then T2 is the corresponding clause. Otherwise T2 is just T1 unchanged. For example,

```
| ?- expand_term((a-->b),T).

T = a(_154,_152):-b(_154,_152)
```

The user may define other transformations to be done by expand_term by defining clauses for term_expansion/2. expand_term calls term_expansion first; if it succeeds the grammar rule expansion is not tried.

20-4-2. phrase(+Phrase,?List)

This predicate is the normal way to commence execution of grammar rules. Viewed declaratively, the list List is a phrase of type Phrase (according to the current grammar rules), where Phrase is either a non-terminal or, more generally, a grammar rule body.

Phrase must be non-variable. List may be bound to a list of symbols or character codes, in which case this goal corresponds to using the grammar for parsing. Or it may be unbound, in which case the grammar is being used for generation. For example, suppose we have the following simple grammar.

```
s --> [] | [a], s.
```

Then we can generate phrases of type 's' as follows.

```
| ?- phrase(s,L).

L = [] ;

L = [a] ;

L = [a,a] ;

L = [a,a,a]
```

and so on indefinitely.

20-4-3. 'C'(?S1,?Terminal,?S2)

Not normally of direct use to the user, this built-in predicate is used in the expansion of grammar rules (see above). It is defined by the clause:

```
'C'([X|S],X,S).
```

CHAPTER 21

ACCESS TO UNIX

The predicate described here provides the most commonly needed access to Unix. You can also extend the Prolog system with additional C code, including system calls, using the C interface (see chapter 23).

The reason for channeling all the interaction with Unix through a single built-in predicate, rather than having separate predicates for each function, is simply to localize the system dependencies. Admittedly, this makes for more cumbersome commands, so you may wish to put some clauses such as

```
cd :- unix(cd).
cd(X) :- unix(cd(X)).
```

in your 'prolog.ini' file ('prolog.ini' files are discussed in section 1-3).

21-1. unix(cd(+Path))

Changes the working directory of Prolog (and Emacs if running under the editor interface) to that specified by its argument which should be an atom corresponding to a legal directory. If the argument is not of this form, the goal just fails.

Note that the "Escape x cd Path" command under Emacs has the same effect as this, except that Emacs allows a slightly more general form of Path in that it accepts the '~' prefix (see csh(1) in the Unix manual).

21-2. unix(cd)

Changes the working directory of Prolog (and Emacs if running under the editor interface) to your home directory.

Note that the "Escape x cd" command of Emacs is exactly the same as this.

21-3. unix(shell(+Command))

Command must be an atom and it is passed to a newly created shell process for execution as a shell command. The goal fails if Command is not an atom, or if the shell returns with a non-zero result.

21-4. unix(shell)

Starts up an interactive shell. The shell run depends on your SHELL environment variable. You can exit from the shell by typing ^D (your end-of-file character) unless under Emacs in which case you should type "exit"

if using the C shell (csh) or "kill -9 $$" if using the Bourne shell (sh). (Under Emacs you should be able to simply type ^X^D to exit from any shell, but currently this does not work.) The call to shell/0 fails if a non-zero result is returned by the shell.

CHAPTER 22

THE EMACS INTERFACE

22-1. Overview

This chapter presupposes some knowledge of the Emacs editor. An introduction to the Emacs editor is included in the Quintus Prolog User's Guide. This chapter summarizes the features which have been added to Emacs specifically to support Quintus Prolog. The supplied version of Emacs was written by Unipress Inc., and it is documented separately.

To run Prolog under the Emacs interface, type

```
        prolog +
or
        prolog + file-to-be-edited
```

at the Unix prompt. You then get Emacs running with two windows: file-to-be-edited, if given, appears in the upper window while Prolog is running in the lower window.

You can talk to Prolog very much as you would without the editor interface. The only difference is that control characters generally have their Emacs meaning rather than any meaning they might have outside of Emacs. The reason for this is that the Prolog window is still an edit buffer and you are free to move up and down it using the full range of editing commands. Thus ^D means delete the next character, and ^U may be used to specify an argument for the next command.

The general philosophy of the Prolog/Emacs interface is that you should not be able to lose your Prolog window, for example, by changing buffers. For this reason, a few commands have been slightly amended in a way which, hopefully, will seem very natural to you. There are also a number of extra key bindings which are described in section 22-2.

Emacs is a customizable editor. You can use a language called MockLisp to extend or change the way it behaves, and in fact this is the way that the Prolog/Emacs interface has been built. If you want to make your own extensions you may need to know something about the way this interface works; some notes to assist you are provided in section 22-5.

22-2. Key Bindings

This section describes the key bindings connected with the Prolog/Emacs interface. For a complete listing of all the key bindings applicable in a particular window type Escape x describe-bindings.

The following apply only in the Prolog window, not in the upper window(s):

^X ^D sends an end-of-file to Prolog. Since ^D (which is the default

end-of-file character in Unix) is taken up by the editor you need this in order to exit from a break or to exit from Prolog altogether. Having exited from Prolog using this command you can start up a new Prolog by typing Escape x followed by "restart-prolog".

^X ^E allows you to edit the previous question you typed to the Prolog prompt and resubmit it. Effectively, it grabs the last question and brings it down to the bottom of the buffer. There you can edit it if necessary, then move your cursor to the end of the line and type Return. You can also grab questions other than the most recent one by specifying an argument to this command, 2 to get the second last, 3 to get the third last, and so on. Another way of doing this is to move the cursor back to the question you want and type ^X ^E.

The following key bindings apply in any window:

^C sends an interrupt to the Prolog process (exactly as if you were not running under Emacs).

<esc>^C causes an irreversible exit from Emacs and Prolog. You will be prompted to make sure (1) you don't want any buffers you have failed to save, and (2) you don't mind your Prolog process being killed.

^X ^C stops the Emacs process and returns you to the Unix shell. If a Prolog program is running it will continue to run but you will not see any output from it. You can get your Emacs back by typing "fg" - see the documentation for csh in the Unix manual for further details.

<esc> e (for enlarge window) enlarges the current window as much as it can without destroying the other window (the current window should be either the Prolog window or the one above it). Typed a second time, this command undoes what it did the first time. If an argument is given, the other window is reduced only to the specified number of lines.

^X . Find the source code for a particular procedure. If the cursor is positioned on or before the predicate name part of a goal, then you don't need to type that name. Otherwise, in response to an explanatory prompt, you should type the name of the procedure optionally followed by a '/' and its arity. The file containing the specified procedure is then visited and the cursor is positioned at the beginning of the procedure. There are some layout conventions which must be followed for this facility to work; see section 22-4. The facility is also available via the '.' option to the debugger (see the Quintus Prolog User's Guide).

<esc> x restart-prolog
Restart Prolog, optionally using a different saved state. This is useful if you have exited from Prolog for some reason (for example, by typing ^C followed by 'e') and want to start again, or

if you want to run a different saved state.

<esc> x prolog-mode
Changes the current buffer to Prolog mode. See section 22-3.

The following key bindings apply except in the Prolog window:

<esc> i (for interpret) is for loading procedures from the edit buffer into the interpreter (that is, consulting them). You then are prompted to choose one of three options; you can consult

1. the procedure in which the cursor is currently positioned (see section 22-4 for restrictions on program layout necessary for this to work);

2. the region between the cursor and the mark;

3. the whole buffer.

<esc> k (for kompile?) is for compiling procedures from the edit buffer, and the options are the same as for <esc> i.

22-3. Prolog Mode

Prolog mode applies automatically whenever you are editing a file which ends with the characters '.pl'. It is useful when you are editing Prolog source code. In Prolog mode:

- whenever you type a closing parenthesis or bracket the corresponding opening one is flashed. This bracket matching attempts to be clever about strings in quotes, because normally you would not want a bracket written within quotes to count for matching purposes. Unfortunately, this means that the bracket matching does not work properly when radix notation (for example, 16'100 is hexadecimal 100, or 256 decimal) is used.

- the definition of linefeed is modified: immediately after a ':-' it is equivalent to a Return followed by 3 spaces. Otherwise it is equivalent to a Return followed by enough tabs and spaces to put the cursor underneath the first non-space character in the current line.

22-4. Layout Restrictions

There are some restrictions on program layout which are necessary for Escape i p (consulting a procedure), Escape k p (compiling a procedure) and ^X . (find definition) to work properly. They are:

1. Group clauses of the same name and arity together.

2. Start the head of each clause at the beginning of a line, that is, right up against the left-hand side of your window or screen.

Indent any continuation lines for that clause.

3. If you use multi-line comments, indent all the continuation lines.

4. If you are defining clauses for a predicate which is an operator, do not use the operator property of the predicate when writing the head of the clause. For example, if you want to define clauses for '+'/2, write the head of the clause in the form "+(A,B") and not "A + B".

22-5. Emacs Customization Notes

This section is only for those who wish to customize their Emacs environment. All of the MockLisp code written by Quintus is supplied with the system. This code can be found in /usr/local/quintus/ml, assuming that your system manager has installed the quintus directory in /usr/local.

Please note that future releases of Quintus Prolog may not be compatible with all modifications you may make.

22-5-1. Initialization Files

Your Emacs initialization file (.emacs_pro in your home directory), if you have one, is loaded before the MockLisp files defining the editor interface. It MUST NOT call argv or argc or the package will break. There are two "hook functions" you may define to customize the various modes used by the system.

split-screen-mode-hook
: If a function by this name is defined, it will be called after all the initializations done on invoking Prolog through the editor interface are completed, and before the screen is displayed.

prolog-mode-hook
: If a function by this name is defined, it will be called every time Prolog mode is entered. Prolog mode is entered every time you load a file with a ".pl" extension.

22-5-2. Rebinding Keys

The file splitscreen.ml holds many of the editor interface functions. The function split-screen, defined inside of splitscreen.ml, is always executed upon startup. This function is responsible for setting up the customized Emacs environment that is the Quintus Prolog editor interface. It creates key bindings and sets up a number of variables. All key bindings are made in the file keys.ml.

The Quintus Prolog editor interface environment is the result of rebinding the keys of many of the most commonly used functions to analogous functions that know about the Prolog window and automatically perform extra "housekeeping".

The general naming scheme for these new functions is to add the prefix "split-screen" to the old function name.

In addition to rebinding keys, the function **meta-x-trap**, also defined in the file splitscreen.ml, is used to catch any of the old function names that you might explicitly invoke. It is bound to the key Escape x. **meta-x-trap** works by "pushing" a string representing its substituted command sequence. For this to work properly, the OLD Meta-x function, execute-extended-command, must be bound to the key Escape ^~ (also known as \e\036). This key binding was picked to be out of the way, and should NEVER be rebound, or **meta-x-trap** will break.

If you reset variables, be particularly careful of any that affect the handling of the window environment. Resetting some of these may break the editor interface.

If you think you have found a bug with the Emacs interface, please check that the bug occurs when you have no initialization file.

CHAPTER 23

INTERFACE TO C FUNCTIONS

23-1. Overview

Quintus Prolog provides tools for loading, and then calling, C programs from within Prolog. This may be desirable for several reasons:

1. To speed up certain critical operations by writing them in the lower level language C.

2. To interface with the operating system and other libraries and programs.

3. To integrate already existing C programs to form a composite system.

C functions are loaded directly into the running Prolog system using the built-in predicate load_foreign_files/2 (see below). Prior to calling this predicate, you need to prepare facts in the database which specify which C functions should be callable from Prolog and how their arguments should be passed. Using this information, the load_foreign_files command automatically sets up the linkage from Prolog procedures to C functions.

The interface allows the passing and returning of Prolog's simple (atomic) data types: integers, floats and atoms. When data is passed between Prolog and C, it is automatically converted between its Prolog representation and its C representation. In this way, your C program does not have to understand Prolog's internal data structures, and thus is portable between different releases of the Quintus Prolog system. Complex data structures, such as lists and trees, cannot be passed directly between Prolog and C. However, such data structures can be passed by unpacking them in Prolog and passing their atomic components.

The Prolog system manages its own working storage in sophisticated ways. You can allocate space for C programs either statically in the programs themselves, or dynamically using the malloc() family of routines. C programs CANNOT control the overall storage allocation (that is, at the brk(), sbrk() level). You will get an error message if you try to do this.

For applications which involve integrating large and complex C programs with Prolog programs, Quintus recommends investigating the possibility of running the C and Prolog programs as separate Unix processes communicating through an inter-process communication channel. In this approach, Prolog's C interface is used to load C code to support the Prolog side of this communication. The Prolog/Emacs interface is an example of a multi-process system.

Some examples of using the C interface are supplied with the Quintus Prolog system. You should be able to find these in the "quintus" directory (by default this is /usr/local/quintus).

23-2. load_foreign_files(+ListOfFiles,+ListOfLibraries)

Loads C functions directly into the running Prolog system. For example,

```
| ?- load_foreign_files( ['math.o','other.o'], ['-lm'] ).
```

Each file in the ListOfFiles is a normal Unix object file. You need to produce these files from your source files using the normal C compiler. This will produce object files that can then be loaded into Prolog. The object files will normally have names ending in '.o'. For example:

```
% cc -c math.c
% cc -c other.c
```

Any libraries that need to be searched when linking these object files together are listed in ListOfLibraries. This is a list of atoms which will be used to provide options to the Unix linker "ld". The load_foreign_files command automatically generates a call to "ld" of the following form (see the Unix documentation for "ld" for more details):

```
ld -N -x -A ?? -T ?? -o ?? LinkFile ListOfFiles ListOfLibraries -lc
```

The ?? fields are filled in by Prolog. LinkFile will be an internally generated C file used by Prolog when loading the C program. The ListOfFiles and ListOfLibraries fields are filled in from the corresponding lists passed to load_foreign_files. Details of how libraries are searched are given in the Unix documentation for "ld". In general, ListOfLibraries will contain special '-l' options and/or names of library files. In many cases there may be no additional libraries required, in which case ListOfLibraries = [].

The result of linking these object files and libraries is loaded into the Prolog system, and C functions can then be called from Prolog. This loading process may fail if:

- the facts in the database (see below), describing how to link C functions to Prolog procedures, are incomplete;
- the C functions specified have already been loaded;
- or the call to the Unix linker "ld" fails.

If the load does not complete successfully then no change is made to the Prolog state. The load can be subsequently retried once the problem has been corrected.

23-3. Linking C functions to Prolog procedures

When load_foreign_files/2 is called, the system looks in the database for facts of the form:

```
foreign_file( FileName, [ CFunction1, CFunction2, ..., CFunctionN ] ).
```

```
foreign( CFunction1, PredicateSpecification1 ).
foreign( CFunction2, PredicateSpecification2 ).
....
foreign( CFunctionN, PredicateSpecificationN ).
```

Example:

```
foreign_file( 'math.o', [ sin, cos, tan ] ).

foreign( sin, sin(+float,[-float]) ).
foreign( cos, cos(+float,[-float]) ).
foreign( tan, tan(+float,[-float]) ).
```

Each foreign_file/2 fact lists the C functions that will be provided by each object file. A fact of this form needs to be provided for every object file listed in the ListOfFiles argument to load_foreign_files/2. The C functions listed should only be the ones that are to be attached to Prolog procedures. Supporting functions in the file, which are called by C and are not to be directly attached, should not be listed.

Each foreign/2 fact describes how a C function is to be attached to a Prolog procedure. The PredicateSpecification specifies the Prolog procedure and also the argument passing interface (described below). A fact of this form needs to be provided for every C function that is to be attached to a Prolog procedure.

Once the ListOfFiles and ListOfLibraries have been successfuly linked together and loaded into the Prolog state, then all the specified Prolog procedures are abolished and have their definitions replaced by links to the C functions. Calling the Prolog procedures will now result in the C functions being called.

You can link a Prolog predicate directly to a C library function. However, note that the functions shown in the library documentation are sometimes actually C macros (see the .h files). In this case, the simplest thing to do is write a small C function which simply calls the macro, and then link to that.

You may abolish or redefine (using consult/1 or compile/1) any procedure which has been attached to a C function. If such a procedure is redefined, then it ceases to be attached to the C function. The link is replaced by the new definition. It is not possible to re-establish this link.

The foreign_file/2 and foreign/2 facts must be consistent whenever load_foreign_files/2 is called. However, they are not used after this point and they may be abolished if you wish (perhaps to reclaim some space).

The command load_foreign_files/2 can be used any number of times in a Prolog session to keep loading further C programs. All such loaded C programs are saved in the Prolog saved state when save/1 is used, and will be available after any subsequent use of restore/1 (or when the saved state is run directly from Unix).

Note that, if you want to load more than one foreign module and thus have foreign_file and foreign facts in more than one file, then you need to call load_foreign_files after consulting/compiling each one of these files. The

reason for this is that foreign_file and foreign are treated just like any other predicates, so that each consult/compile throws away any previous definition. In case you forget, you will get a "multiple definition" style warning in this case; to avoid this warning, abolish all the facts after doing the load_foreign_files.

Once a C program is loaded, it cannot be unloaded or replaced. If a load operation duplicates any symbols from a previous load, it will NOT be completed. If you wish to be able to return to a particular state, you should use save/1 to create a saved state before loading any more C code. Then you can use restore at any time to get back to that state.

23-4. Specifying the argument passing interface

C functions are linked to Prolog procedures when the load_foreign_files/2 command is called, according to the foreign/2 facts you have provided.

These are of the form:

```
foreign( CFunction, PredicateSpecification ).
```

where CFunction is the name of the C function (an atom),

and PredicateSpecification is:

```
PredicateName(ArgSpec,ArgSpec,...)
```

and PredicateName is the name of the Prolog predicate (an atom),

and there is an ArgSpec for each argument of the predicate,

and ArgSpec is one of:

```
+integer     +float     +atom     +string
-integer     -float     -atom     -string
[-integer]  [-float]  [-atom]  [-string]
```

Examples:

```
foreign( sin,       sin(+float,[-float]) ).
foreign( rename,    rename(+string,+string) ).
foreign( cxyz,      foo(+integer,-integer,+atom,-atom) ).
```

Note that the CFunction name does NOT have to be the same as the PredicateName.

The interface allows the simple Prolog data types, atoms, integers and floating point numbers, to be passed to C functions and returned from C functions. Prolog checks the types of the arguments it passes to C, and the call will fail if any argument is not the right type. Prolog assumes that C will return results of the specified type.

The interface is responsible for all the data conversions between Prolog's internal representation and C's internal representation. The C program does not need to know how Prolog represents atoms, integers and floats in order to interface with Prolog. This feature simplifies the integration of C and Prolog, and allows for compatibility across later versions of Quintus Prolog and versions of Quintus Prolog running on other hardware. In particular, this feature makes it easier to interface directly with already written C functions in libraries and other programs.

The ArgSpec specifications have the following meanings:

```
Prolog: +integer
C:      long int
```

The argument must be instantiated to an integer. The Prolog integer is converted to a C integer and passed to the C function.

```
Prolog: +float
C:      float
```

The argument must be instantiated to a float. The Prolog float is converted to a C float and passed to the C function. The float is passed following the C language convention that single precision floats are passed as double precision. The C parameter declaration should still be 'float', however, to avoid confusion.

```
Prolog: +atom
C:      long unsigned
```

The argument must be instantiated to an atom, otherwise the call fails. A canonical representation (see below) of the Prolog atom is passed to the C function as an unsigned integer.

```
Prolog: +string
C:      char *
```

The argument must be instantiated to an atom, otherwise the call fails. A pointer to a null terminated string of characters containing the printed representation of the atom is passed to the C function. This string must NOT be overwritten by C.

```
Prolog: -integer
C:      long int *
```

A pointer to an integer is passed to the C function. It is assumed that C will overwrite this integer with the result it wishes to return. When the C function returns, the pointed to integer is converted to a Prolog integer and unified with the corresponding argument of the Prolog call. The argument can be of any type; if it cannot be unified with the returned integer, the call will fail. If the C function does not overwrite the integer, then the result is undefined.

```
Prolog: -float
C:      float *
```

A pointer to a single precision float is passed to the C function. It is assumed that C will overwrite this float with the result it wishes to return. When the C function returns, the float is converted to a Prolog float and unified with the corresponding argument of the Prolog call. The argument can be of any type; if it cannot be unified with the returned float then the call will fail. If the C function does not overwrite the float then the result is undefined.

```
Prolog: -atom
C:      long unsigned *
```

A pointer to an unsigned integer is passed to the C function. It is assumed that C will overwrite this unsigned integer with the result it wishes to return. This result should be a canonical representation of an atom already obtained by C from Prolog. Returning an arbitrary integer will have undefined results. When the C function returns, the atom represented by the pointed to unsigned integer is unified with the corresponding argument of the Prolog call. The argument can be of any type; if it cannot be unified with the returned atom, then the call will fail. If the C function does not overwrite the unsigned integer, then the result is undefined.

```
Prolog: -string
C       char * *
```

A pointer to a character pointer is passed to the C function. It is assumed that C will overwrite this character pointer with the result it wishes to return. This result should be a pointer to a null terminated string of characters. When the C function returns, the atom which has the printed representation specified by the string is unified with the corresponding argument of the Prolog call. The argument can be of any type; if it cannot be unified with the returned atom, then the call will fail. If the C function does not overwrite the character pointer, then the result is undefined. Note: Prolog copies the string if required, so that it is not necessary for the C program to worry about retaining it.

```
Prolog: [-integer]
C:      return (long int)
```

No argument is passed to C. The return value from the C function is assumed to be an integer. It is converted to a Prolog integer and unified with the corresponding argument of the Prolog call. The argument can be of any type; if it cannot be unified with the returned integer, then the call will fail.

```
Prolog: [-float]
C:      return (float)
```

No argument is passed to C. The return value from the C function is assumed to be a float. This returned value is assumed to follow the C language convention that single precision floats are returned as double precision. The result is converted to a Prolog float and unified with the corresponding argument of the Prolog call. The argument can be of any type; if it cannot be unified with the returned integer, then the

call will fail.

```
Prolog: [-atom]
C:      return (long unsigned)
```

No argument is passed to C. The return value from the C function is assumed to be an unsigned integer which should be a canonical representation of an atom already obtained by C from Prolog. Returning an arbitrary integer will have undefined results. The atom represented by the unsigned integer is unified with the corresponding argument of the Prolog call. The argument can be of any type; if it cannot be unified with the returned atom, then the call will fail.

```
Prolog: [-string]
C       return (char *)
```

No argument is passed to C. The return value from the C function is assumed to be a character pointer pointing to a null terminated string of characters. The atom which has the printed representation specified by the string is unified with the corresponding argument of the Prolog call. The argument can be of any type; if it cannot be unified with the returned atom then the call will fail. Prolog copies the string if required, so that it is not necessary for the C program to worry about retaining it.

Arguments are passed to C in the same order as they appear in the Prolog call. Only one "return value" argument can be specified; that is, there can be only one [-integer], [-float], [-atom] or [-string] specification. There need not be any "return value" argument in which case the value returned by the C function is ignored. Both input and output specifications cause data to be passed to the C function (except of course for the "return value" argument, if present). Each input argument is appropriately converted and passed, and each output argument is passed as a pointer through which the C function will send back the result.

Note that Prolog procedures attached to C functions are always determinate. However, they will fail if their input arguments are of the wrong type, or if an output returned from C cannot be unified with the corresponding argument of the Prolog call.

Prolog integers have a different precision from C integers. In the current Sun and VAX implementations, Prolog integers are 29-bit integers, whereas C long ints are 32-bit integers. When integers are returned to Prolog from C they will be reduced modulo 2^29 to a number in the range -2^28..2^28-1, that is, the three most significant bits are lost.

Prolog floats also have a different precision from C (single precision) floats. In the current Sun and VAX implementations, Prolog floats have a 20-bit signed mantissa, whereas C floats have a 24-bit signed mantissa. (Both have an 8-bit exponent.) When floats are returned to Prolog from C, there will be a corresponding loss in precision.

23-5. Access to Prolog atoms from C

The C interface allows Prolog atoms to be passed to C either in a canonical form as unsigned integers, or as pointers to character strings.

For each Prolog atom there is a single canonical representation. C programs can rely on the property that identical atoms have identical canonical representations. Note, however, that the canonical form of an atoms is NOT necessarily identical across different invocations of the program. This means that canonical atom representations should not be used in files or inter-program communication. For these purposes use strings. C programs can store canonical atoms in data structures and pass them around and back to Prolog, but they should not attempt to construct or decompose them.

Strings passed from Prolog to C should NOT be overwritten. Strings passed back from C to Prolog are automatically copied by Prolog if necessary. Thus the C program does not have to retain them and can reuse their storage space as desired.

In addition to obtaining and returning atoms through the interface Prolog provides two C functions for converting back and forth between canonical atoms and strings.

```
long unsigned QP_atom_from_string(string)
        char *string;
```

Returns the canonical representation of the atom whose printed representation is string. The string is copied and the C routine can reuse the string and its space.

```
char * QP_string_from_atom(atom)
        long unsigned atom;
```

Returns the string of characters for the atom with canonical representation atom. This string should NOT be overwritten by C.

Canonical atoms are particularly useful as constants to be used in passing back results from C functions. The above routines can be used to initialize tables of such constants.

23-6. Important Prolog assumptions

23-6-1. Storage management assumptions

The Prolog system has a sophisticated storage management strategy. The system automatically expands and contracts space, allocating from Unix and returning space to Unix as required. The standard malloc family of library functions are provided by the Prolog system and they allocate memory from the Prolog codespace. These can be used by a C program to allocate memory for its own purposes. C programs CANNOT use brk() and sbrk(). The Prolog system provides

dummy versions of these which will cause a Prolog error if called.

Note that C programs reside in the same address space as the Prolog system. C programs which mistakenly write to areas of memory used by the Prolog system can cause unpredictable results.

23-6-2. Input/output assumptions

Prolog uses the standard C I/O library for input and output, except for saving and restoring saved states when the Unix system calls open(2), read(2) and write(2) are used directly.

C programs can use any of the standard C I/O and/or Unix I/O libraries.

23-7. Debugging loaded C functions

C programs should be independently tested before they are loaded into a running Prolog program. No tools are currently provided for debugging C programs once they have been loaded into Prolog.

CHAPTER 24

MISCELLANEOUS BUILT-IN PREDICATES

24-1. numbervars(?X,+N0,-N1)

Instantiates each of the variables in X to a term of the form '$VAR'(N). For example,

```
| ?- Term=foo(A,A,B), numbervars(Term,22,_), display(Term).
foo($VAR(22),$VAR(22),$VAR(23))
```

N0 must be an integer. That integer is used as the value of N for the first variable in X (starting from the left), as shown in the example above. The second distinct variable in X is given a value of N satisfying "N is N0+1"; the third distinct variable gets the value N0+2, and so on. The last variable in X has the value N1-1.

Notice that in the example above, display is used rather than write. This is because write treats terms of the form $VAR(N) specially; it writes 'A' if N=0, 'B' if N=1, ...'Z' if N=25, 'A1' if N=26, etc. That is why, if you type the goal in the example above, you will get the variable bindings printed out as follows.

```
Term = foo(W,W,X),
A = W,
B = X,
```

24-2. ?X = ?Y

Defined as if by the clause "Z=Z."; that is, X and Y are unified.

24-3. length(?L,?N)

If L is instantiated to a list of determinate length, this length is unified with N. For example,

```
| ?- length([a,B,1,f(x)],N).

N = 4
```

Otherwise, if L is uninstantiated and N is a non-negative integer, then L is bound to a list of N distinct variables. For example,

```
| ?- length(L,2).

L = [_117,_119]
```

If the arguments to length satisfy neither of these conditions, the call

simply fails.

24-4. prompt(-Old,+New)

The sequence of characters (prompt) which indicates that the system is waiting for user input is represented as an atom, and unified with Old. New must be an atom and specifies the prompt to be used henceforth. In particular, the goal

```
prompt(X,X)
```

matches the current prompt to X, without changing it.

Prompts are not shown when you are running under the Emacs interface, apart from the top-level prompt "| ?- " which cannot be changed. This predicate only affects the prompt given when a user's program is trying to read from the terminal (for example, by calling read). Note also that the prompt is reset to the default "|: " on return to top-level.

APPENDIX I

COMPARISON OF QUINTUS PROLOG AND PROLOG-20

Quintus Prolog is very similar to Prolog-20, DEC10 Prolog and C-Prolog. The purpose of this document is to describe the differences, and in doing so to provide an introduction to Quintus Prolog for those who are familiar with one of these systems. A guide to porting programs to Quintus Prolog is given in the User's Guide.

I.1. The Emacs Editor Interface

Perhaps the most interesting new feature of Quintus Prolog is the interface to the Emacs editor. You get this by typing

```
          prolog +
or        prolog + file-to-be-edited
```

to the Unix prompt. You then get Emacs running with two windows: file-to-be-edited (if given) appears in the upper window while Prolog is running in the lower window.

You can talk to Prolog very much as you would without the editor interface. The only difference is that control characters generally have their Emacs meaning rather than any meaning they might have outside of Emacs. The reason for this is that the Prolog window is still an edit buffer and you are free to move up and down it using the full range of editing commands. Thus ^D means delete the next character, and ^U may be used to specify an argument for the next command.

There is also an Emacs command which allows you to conveniently recall a previous Prolog command so that you can then modify it and resubmit it to Prolog.

It is possible to consult or compile Prolog code directly from an Emacs buffer; you can choose between consulting or compiling the whole buffer, a specified region of that buffer or just a single procedure.

See chapter 22 for a full description of the Emacs interface.

I.2. The Help System

Both the User's Guide to Quintus Prolog and this Reference Manual are available on-line. The help system is most conveniently used in conjunction with the Emacs interface, but can also be used without it.

The help system is menu-driven, with menus corresponding to the chapters within a manual, the sections within a chapter and so on. There are also two methods of topic-oriented access: help(Topic) is analogous to looking up a Topic in an index, and manual(Topic) is analogous to looking it up in a table

of contents. Topics can be abbreviated, and menus are generated in ambiguous situations.

I.3. The C Interface

Quintus Prolog provides tools for loading, and then calling, C programs from within Prolog. C functions are loaded directly into the running Prolog system using the built-in predicate load_foreign_files/2. Prior to calling this predicate, the user prepares facts in the database which specify which C functions should be callable from Prolog and how their arguments should be passed. Using this information the load_foreign_files command automatically sets up the linkage from Prolog procedures to C functions.

The interface allows the passing and returning of Prolog's simple (atomic) data types: integers, floats and atoms. When data is passed between Prolog and C it is automatically converted between its Prolog representation and its C representation. In this way the user's C program does not have to understand Prolog's internal data structures, thus making the C code portable between different releases of the Quintus Prolog system. Complex data structures, such as lists and trees, cannot be passed directly between Prolog and C. However, such data structures can be passed by unpacking them in Prolog and passing their atomic components.

See chapter 23 for a full description of the C interface.

I.4. Floating Point

Quintus Prolog provides floating point numbers (floats). Examples of acceptable syntax for floats are

```
0.0   -1.0   0.54   1000.0   1.0E6   12.345678e-12
```

A float has approximately 6 decimal digits of precision. Its magnitude, if nonzero, must be in the approximate range of 0.29e-38 to 1.7e+38 on the VAX, or 2.2e-44 to 3.4e+38 on the Sun.

In Quintus Prolog, the symbol '/' is used for floating point division rather than integer division, that is, its result is always a float. The symbol for integer division (which truncates fractional part of the answer) is '//'. This is consistent with C-Prolog.

There are two new arithmetic functors for coercing integers to floating point and vice versa. For example,

```
| ?- X is 2.5, Y is integer(X), Z is float(Y).

X = 2.5,
Y = 2,
Z = 2.0
```

Note that the effect of the arithmetic functor integer/1 in an expression is

to truncate its argument - the fractional part is thrown away. It should not be confused with the predicate integer/1 which, as in Prolog-20, succeeds if and only if its argument is instantiated to an integer. There are two new similar predicates:

float(N) - true if N is instantiated to a floating point number
number(N) - true if N is instantiated to an integer or a float

I.5. Improved Compiler/Interpreter Interface

Like Prolog-20, Quintus Prolog provides both an interpreter and a compiler. There are a number of important differences however:

- Compiled and interpreted code can be freely intermixed, with no need for public declarations. The advantages of using the interpreter are that there are more extensive debugging facilities, and space is reclaimed when procedures are redefined. The advantage of using the compiler is that compiled code is very fast and space-efficient.

- consult has been changed to behave like reconsult in Prolog-20. That is, the procedures defined in a consult operation replace (rather than extend) any previous definitions for those procedures. This makes consult much more like compile so that alternating between consulting and compiling a piece of code is much simpler. It also means that spreading the definition of a procedure across more than one file will NO LONGER WORK. reconsult is just treated as a synonym for consult.

- In Prolog-20 you can use assert, retract and clause on interpreted code but not on compiled code. In Quintus Prolog you can use these procedures only on DYNAMIC procedures. Regardless of whether a procedure is compiled or consulted it may be preceded by a declaration of the form

  ```
  :- dynamic foo/1.
  ```

  which makes foo/1 dynamic. (This declaration must appear in the same file as the procedure itself.) Note that compiled dynamic predicates are actually implemented like interpreted predicates; that is, they are shown by listing and in debugging, and also they are slow in comparison with non-dynamic compiled code.

  You do not need to have a dynamic declaration for a procedure which is only used by assert, retract and/or clause. However, it is recommended that you do have one if the procedure is to be called (for example, :- foo(x).) since if this happens before foo/1 is known to be dynamic you may get an "unknown procedure" trap (see unknown/2, section 15-9).

I.6. Improved Debugger

There are a number of improvements to the debugger:

- trace/0 now sets a permanent state in which the debugger will start creeping (single-stepping) on every goal typed at the top-level. debug/0 sets the state so that the debugger starts leaping on every goal typed at the top-level. notrace/0 is a new predicate identical to nodebug/0 - which turns the debugger off. These changes rationalize the top-level use of the debugger.

- Turning the debugger off does not remove spypoints, although no information will be shown about these spypoints while debugging is off. A new predicate, nospyall/0, is provided for removing all spypoints.

- Spypoints can be set on compiled procedures, although ancestor and depth information will not be available at such spypoints.

- Debugger messages now show if a procedure is compiled or is a built-in procedure.

- The 'r' (retry) option is more robust; in Prolog-20 it sometimes behaves strangely when interpreted and compiled code are mixed.

- By default the debugger only prints structures down to 10 levels deep - you just get '...' in place of any deeper structure. This makes it much easier to debug programs which manipulate large data structures. The print depth limit can be reset or removed using the '<' option.

- There are some additional debugging options:

    + put a spypoint on procedure currently being shown

    - remove any spypoint on procedure currently being shown

    = show the current state of the debugger

- Unknown procedure catching (see unknown/2, section 15-9) works consistently. (In Prolog-20 its behavior depends on whether or not debugging is switched on, and it does not work for compiled code). Also, there is no loss of efficiency incurred by using this facility. It is on by default.

- leash/1 now takes a more convenient form of argument (as well as continuing to support the Prolog-20 argument forms). You give it a list of the ports you want leashed; for example,

```
| ?- leash([call,redo]).
```

  will set leashing to its default under Prolog-20. The default under Quintus Prolog is leashing on all four ports.

I.7. Style Checking

By default, Quintus Prolog prints warnings, as it consults or compiles, about

1. single occurrences of a "named" variable in a clause, where "named" in this context means that the variable name does not begin with an underscore ('_');

2. procedures for which all the clauses are not adjacent (contiguous) in the source file;

3. procedures for which clauses are encountered in more than one file (multiple definitions). In Prolog-20 it is possible to spread the definition of a procedure across more than one file if you use consult, but not if you use compile or reconsult. This style warning can be useful in catching any attempt to do this.

These warnings can be very helpful in catching typing mistakes, and it is highly recommended that you adapt your programming style to make the best use of them. Style checking can be fully controlled by the predicates style_check/1 and no_style_check/1 which each take as argument any of the four atoms below.

all	turn on/off all style checking
single_var	turn on/off checking for single variable occurrences
discontiguous	turn on/off checking for discontiguous procedures
multiple	turn on/off checking for multiple definitions

I.8. Stream-Based Input and Output

In addition to fully supporting the Prolog-20 I/O predicates, Quintus Prolog supports the more powerful concept of streams. A stream is a special Prolog term, and there is a version of each of the Prolog-20 I/O predicates which takes an extra stream argument.

A new predicate, open/3, opens a file in one of three modes, 'read', 'write' or 'append'.

Streams are described in section 7-3.

I.9. Improved Handling of Database References

When a clause is retracted (or erased), or when a recorded term is erased, the clause (or term) disappears immediately from the database. Safety is maintained in that any attempt to access it via a database reference (a "dangling pointer") will just fail. Thus instance/2 and clause/3 cannot be used to get at clauses or recorded terms which have been erased.

There is no restriction against asserting clauses containing database references if you really want to.

I.10. Runnable Saved States

A saved state produced by save/1, save/2 or save_program/1 (which is the same as save/1 except that only the program and not the execution state is saved) can be run directly from Unix by typing the name of the file into which it was saved. You can also run it with the Emacs interface by typing, for example,

```
        savedstate +
or      savedstate + file-to-be-edited
```

I.11. Memory Management

The space occupied by interpreted code is reclaimed when clauses are retracted or procedures abolished. This reclamation is not delayed until backtracking as in Prolog-20. The space occupied by compiled code is not reclaimed.

A stack shifter is incorporated which expands data areas when necessary. The memory space is contracted by the built-in predicate trimcore/0. This predicate can be explicitly called by a user program, but it is automatically called anyway on completion of every goal typed at the top-level.

There is currently no garbage collection of constructed terms. However, this space is recovered on backtracking.

The predicate statistics/2 is fully upward compatible with Prolog-20. The data areas in Quintus Prolog are slightly differently organized, but the Prolog-20 statistics keywords (for example, ?- statistics(global_stack,X)) will still give meaningful results. See section 13-3 for further details.

I.12. Miscellaneous

If a file name specified in a compile or consult command does not end with the characters ".pl", Quintus Prolog adds a ".pl" extension to the file name before searching for it. If there is no file with the extended name, it tries again without the ".pl" extension. Thus you can type

```
| ?- [myfile].
```

instead of

```
| ?- ['myfile.pl'].
```

The printing of answers to questions has been slightly enhanced: variables which have been bound to the same value are grouped together, thus

```
| ?- X = 1, Y = 1.

X = Y = 1
```

Note that if a variable is bound to another uninstantiated variable then these variables are grouped together too

```
| ?- U=V, W=X.

U = V = _27,
W = X = _60
```

Also, when a named variable in a question occurs in a structure but remains unbound, its name is printed in that structure rather than the arbitrary identifier of the form _<integer> which is more normally used. For example,

```
| ?- X = f(Y).

X = f(Y),
Y = _51
```

APPENDIX II

CURRENT LIMITS IN QUINTUS PROLOG

Integers are in the range -2^28 to 2^28-1 (-268,435,456 to 268,435,455).

Floating point numbers are in the approximate range of 0.29e-38 to 1.7e+38 on the VAX, or 2.2e-44 to 3.4e+38 on the Sun. Floats have an 8 bit (base 2) exponent and a 19 + 1 sign bit mantissa giving a precision of approximately 6 decimal digits.

Atoms cannot have more than 512 characters.

Functors and predicates cannot have arities greater than 255.

There are no limits (apart from memory space) on the number of procedures and clauses allowed.

The size of a compiled clause is limited to 2^15 (32,768) bytes of compiled code.

There are various internal limits on the size of compiled clauses which are difficult to relate to user-understandable properties. These are 255 "symbols" (variables, atoms, numbers or functors) per clause head or body goal, 255 "temporary variables" (only occur in head or first goal), and 255 "permament variables" (non temporaries - which have occurrences in goals in the body). The compiler will generate warnings if these limits are exceeded.

There are no restrictions on the size of dynamic or interpreted clauses.

Quintus Prolog's memory space is automatically expanded as necessary up to the "datasize" limit for the process (this can be examined and set from the csh using "limit" before Prolog is run). The memory space is contracted using the built-in predicate trimcore/0. trimcore is automatically called on completion of each goal typed at the top-level.

A maximum of 20 input/output streams can be open simultaneously. 3 of these streams (for user input, user output and error output) are always open and cannot be changed.

APPENDIX III

BUILT-IN PREDICATES

Following is a complete list of Quintus Prolog built-in predicates.

abolish(F,N) abolish the procedure named F arity N
abort abort execution of the program; return to toplevel
ancestors(L) the list of interpreted ancestors of the current clause is L
arg(N,T,A) the Nth argument of term T is A
assert(C) clause C (dynamic predicate) is added to database
assert(C,R) clause C (dynamic predicate) is added to database: reference R
asserta(C) clause C (dynamic predicate) is added first in database.
asserta(C,R) clause C (dynamic) is added first in database: reference R
assertz(C) clause C (dynamic predicate) is added last in database
assertz(C,R) clause C (dynamic) is added last in database: reference R
atom(T) term T is an atom
atomic(T) term T is an atom or number
bagof(X,P,B) the bag of instances of X such that P is provable is B
break break at the next procedure call
'C'(S1,T,S2) (grammar rules) S1 is connected by the terminal T to S2
call(P) prove (execute) P
character_count(S,N) N is number of chars read/written on stream S
clause(P,Q) there is a clause for dynamic predicate, head P, body Q
clause(P,Q,R) clause for dynamic predicate, head P, body Q, ref R
close(F) close file F
compare(C,X,Y) C is the result of comparing terms X and Y
compile(F) add compiled procedures from file F to the database
consult(F) add interpreted procedures from file F to the database
current_atom(A) A is a currently available atom (nondeterminate)
current_input(S) S is the current input stream
current_op(P,T,A) atom A is an operator type T precedence P
current_output(S) S is the current output stream
current_predicate(A,P) A is name of a predicate, m. g. goal P
current_stream(F,M,S) S is a stream open on file F in Mode M
debug switch on debugging
debugging output debugging status information
depth(D) the current interpreted invocation depth is D
display(T) write term T (prefix notation) to the user stream
erase(R) erase the clause or record, reference R
expand_term(T,X) term T is a shorthand which expands to term X
fail backtrack immediately
false (same as fail)
fileerrors enable reporting of file errors
float(N) N is a floating point number
flush_output(S) flush output buffer for stream S
foreign(F,P) user defined; C function F is attached to predicate P
foreign_file(F,L) user defined; file F defines C functions in list L
functor(T,F,N) the principal functor of term T has name F, arity N
gc enable garbage collection (currently has no effect)
gcguide(F,O,N) change garbage collection parameter F from O to N
get(C) C is the next non-blank character input on the current input
get(S,C) C is the next non-blank character input on stream S

```
get0(C)  C is the next character input on the current input
get0(S,C)  C is the next character input on stream S
halt  exit from  Prolog
help  display a help message
help(T)  give help on topic T
incore(P)  (same as call)
instance(R,T)  an instance of the clause or term refereńced by R is T
integer(T)  term T is an integer
Y is X    Y is the value of arithmetic expression X
keysort(L,S)  the list L sorted by key yields S
leash(M)  set the debugger's leashing mode to M
length(L,N)  the length of list L is N
line_count(S,N)  N is number of lines read/written on stream S
line_position(S,N)  N is number of chars read/written on current line of S
listing  list all interpreted procedures
listing(P)  list the interpreted procedure(s) specified by P
load_foreign_files(F,L) load object files from list F using libraries L
manual  access top-level of on-line manual
manual(X)  access specified manual section
maxdepth(D)  limit invocation depth (interpreted code only) to D
name(A,L)  the list of characters of atom or number A is L
nl  output a new line to current output
nl(S)  output a newline on stream S
nodebug  switch off debugging
nofileerrors  disable reporting of file errors
nogc  disable garbage collection (currently has no effect)
nonvar(T)  term T is a non-variable
nospy(P)  remove spy-points from the procedure(s) specified by P
nospyall  remove all spypoints
no_style_check(A)  turn off style checking of type A
notrace  switch off debugging (same as nodebug)
number(N)  N is a number
numbervars(T,M,N)  number the variables in term T from M to N-1
op(P,T,A)  make atom A an operator of type T precedence P
open(F,M,S)  file F is opened in mode M returning stream S
open_null_stream(S)  output stream S goes nowhere
otherwise  (same as true)
phrase(P,L)  list L can be parsed as a phrase of type P
portray(T)  user defined; tells print what to do
print(T)  portray or else write the term T on the current output
print(S,T)  portray or else write term T on stream S
prompt(A,B)  change the prompt from A to B
put(C)  output character C on current output
put(S,C)  output character C on stream S
read(T)  read term T from current input
read(S,T)  read term T from stream S
recorda(K,T,R)  make term T the first record under key K, reference R
recorded(K,T,R)  term T is recorded under key K, reference R
recordz(K,T,R)  make term T the last record under key K, reference R
repeat  succeed repeatedly
restore(S)  restore the state saved in file S
retract(C)  erase the first interpreted clause of form C
save(F)  save the current state of Prolog in file F
save(F,R)  as save(F) but R is 0 first time, 1 after a 'restore'
```

```
save_program(F)  save the current static state of Prolog in file F
see(F)  make file F the current input stream
seeing(F)  the current input stream is named F
seen  close the current input stream
set_input(S)  set S to be the current input stream
set_output(S)  set S to be the current output stream
setof(X,P,S)  the set of instances of X such that P is provable is S
skip(C)  skip input on current input stream  until after character C
skip(S,C)  skip input on stream S until character C found
sort(L,S)  the list L sorted into order yields S
spy(P)  set spy-points on the procedure(s) specified by P
statistics  output various execution statistics
statistics(K,V)  the execution statistic key K has value V
style_check(A)  turn on style checking of type A
subgoal_of(G)  an interpreted ancestor goal of the current clause is G
tab(N)  output N spaces to current output
tab(S,N)  output N spaces on stream S
tell(F)  make file F the current output stream
telling(F)  the current output stream is named F
term_expansion(T,N)  user defined; tells expand_term what to do
told  close the current output stream
trace  switch on debugging and start tracing immediately
trimcore  reduce free stack space to a minimum
true  succeed
ttyflush  transmit all outstanding terminal output
ttyget(C)  the next non-blank character input from the terminal is C
ttyget0(C)  the next character input from the terminal is C
ttynl  output a new line on the terminal
ttyput(C)  the next character output to the terminal is C
ttyskip(C)  skip over terminal input until after character C
ttytab(N)  output N spaces to the terminal
unix(T)  gives access to Unix facilities
unknown(A,B)  change action on unknown procedures from A to B
user_help  user defined; tells help what to do
var(T)  term T is a variable
version  displays system identification messages
version(A)  adds the atom A to the list of introductory messages
write(T)  write the term T on the current output
write(S,T)  write term T on stream S
writeq(T)  write the term T, quoting names where necessary
writeq(S,T)  write T on S, quoting atoms where necessary
!  cut any choices taken in the current procedure
\+ P  goal P is not provable
X ^ P  there exists an X such that P is provable
X < Y  as integer values, X is less than Y
X =< Y  as integer values, X is less than or equal to Y
X > Y  as integer values, X is greater than Y
X >= Y  as integer values, X is greater than or equal to Y
X = Y  terms X and Y are equal (i.e. unified)
T =.. L  the functor and arguments of term T comprise the list L
X == Y  terms X and Y are strictly identical
X \== Y  terms X and Y are not strictly identical
X @< Y  term X precedes term Y
X @=< Y  term X precedes or is identical to term Y
```

```
X @> Y  term X follows term Y
X @>= Y  term X follows or is identical to term Y
```

APPENDIX IV

BUILT-IN OPERATORS

```
:-op( 1200, xfx, [ :-, --> ])
:-op( 1200,  fx, [ :-, ?- ])
:-op( 1150,  fx, [ mode, public, dynamic ])
:-op( 1100, xfy, [ ; ])
:-op( 1050, xfy, [ -> ])
:-op( 1000, xfy, [ ',' ])
:-op(  900,  fy, [ \+, spy, nospy ])
:-op(  700, xfx, [ =, is, =.., ==, \==, @<, @>, @=<, @>=,
                                    =:=, =\=, <, >, =<, >= ])
:-op(  500, yfx, [ +, -, /\, \/ ])
:-op(  500,  fx, [ +, - ])
:-op(  400, yfx, [ *, /, <<, >> ])
:-op(  300, xfx, [ mod ])
:-op(  200, xfy, [ ^ ])
```

INDEX

READER'S EVALUATION FORM

We are very much interested in your impressions of our documentation and in any suggestions you might have for its improvement. We would be grateful if you would take a few minutes to answer the questions below and mail this page back to us.

1. Can you find the information you need quickly and easily? Is there any information you need that you can't find?

2. Is the text clear and understandable? Please cite any paragraphs and pages that are difficult to understand.

3. Are the documents logically organized? What changes, if any, do you suggest?

4. Are there enough examples, and are the examples helpful?

5. Are there any inconsistencies between the documentation and the software?

6. How can the documents be improved? Please cite specific examples.

(optional) Name _____ Company _____

Please return to Peter Davies, Prolog Technical Support, Artificial Intelligence Ltd., Intelligence House, 58-78 Merton Road, Watford, Hertfordshire, WD1 7BY.

QUINTUS PROLOG
SOFTWARE PROBLEM ACTION REQUEST FORM

START

AIL No

DATE :

Submitted by :

Circle One
- Problem Report
- Enhancement Request
- Documentation Error

Location :

Phone No :

Customers Name

Software Release .....

Hardware (Sun or VAX) .....

Operating System (Unix 4.2 or VAX VMS) .....

Problem Area or Feature

Full Description (Use extra sheet if necessary. Give details leading to problem, and report any messages, that relate to the problem in the order in which they occured.)

Impact (Circle One) • Fatal • Serious • Moderate • Annoying • Enhancement • Documentation

Frequency (Circle One) • Every Time • Intermittent • Once Only

Supporting Documentation MUST Always Accompany An ACTION REQUEST (If there was no message etc then please say so)

- Code listings
- Log File
- Hardcopy of screen
- Sample Printouts or Tape containing Problem
- Any other information that will enable DUPLICATION OF ERROR

For AIL Use Only

- Date Received

Screened By

Impact (Circle/ One) 1 2 3 4 5 **Status** (CircleOne) • Open • Fixed • Closed • Superseded • Rejected

AR Date .....

PLEASE SEND THIS FORM TO Technical Support, AIL, 62-78 Merton Road, Watford, Herts, WD1 7BY .]