Aquarius Prolog User Manual

Ralph Clarke Haygood

Swedish Institute of Computer Science
Box 1263
164 28 Kista
Sweden

March 19, 1993

Abstract

This document is intended for users of Aquarius Prolog, a high-performance, multiple-platform Prolog implementation developed since 1989 by the Aquarius Project at the University of California, Berkeley and the Trojan Project at the University of Southern California. Aquarius Prolog includes a state-of-the-art Prolog-to-native-code compilation scheme and a run-time system with an extensive set of built-in predicates and automatic memory management. This user manual characterizes these facilities for Prolog programmers.

Preface

This document corresponds to Aquarius Prolog version 1.0 for the following processor, operating system pairs: SPARC, SunOS (SPARCstations); MIPS R3000, ULTRIX (DECstations); MIPS R3000, RISC/os (M/2000s); MC68020 (or higher), SunOS (Sun 3s); MC68020 (or higher), HP-UX (HP 9000 Series 300s). As of this publication, Aquarius Prolog is available as a stand-alone system for these platforms only, but others may be added. For up-to-date information or to obtain the software, send electronic mail to aquarius-prolog@acal.usc.edu.

If you find a bug, in the software or this document, we do not promise to fix it, but we would like to hear about it. Send bug reports to aquarius-prolog@acal.usc.edu. Your bug is more likely to be fixed if you send us a complete (though preferably small) example with a complete description of how to make the example manifest the bug.

Your comments on Aquarius Prolog are welcome. Send them to aquarius-prolog@acal.usc.edu or contact

Peter Van Roy
Digital Equipment Corporation Paris Research Laboratory
85, avenue Victor Hugo
92500 Rueil Malmaison
France
vanroy@prl.dec.com

Tom Getzinger
Advanced Computer Architecture Laboratory
3740 South McClintock Avenue Suite 131
Los Angeles, CA 90089-2561
USA
tgetzing@acal.usc.edu

Ralph Clarke Haygood
Swedish Institute of Computer Science
Box 1263
164 28 Kista
Sweden
rch@sics.se

The Aquarius Project at the University of California, Berkeley and the Trojan Project at the University of Southern California have undertaken research during the last several years on many aspects of logic programming implementation and application, and much of this research has influenced Aquarius Prolog. In its present form, Aquarius Prolog is primarily the work of Peter Van Roy, Tom Getzinger, and Ralph Clarke Haygood, but we gratefully acknowledge the contributions of our colleagues, particularly Michael Carlton and Bruce K. Holmer. Among other contributions, Mike wrote the original ‘‘BAM-to-intermediate translator,’’ and Bruce wrote the original ‘‘intermediate-to-target assembler,’’ precursors of the present back-end. We

especially thank Professor Alvin M. Despain of the University of Southern California, principal investigator of the Aquarius Project and the Trojan Project, for his consistent encouragement and support.

Code for some built-in predicates of Aquarius Prolog derives from public domain code for the same predicates by Alan Mycroft, Richard A. O'Keefe, and David H. D. Warren. We a ppreciate their making this code available to the logic programming community. A lso, we thank Michael Schroeder for contributing his implementation of malloc.

This work was sponsored by the Defense Advanced Research Projects Agency and monitored by the Federal Bureau of Investigation under contract number J-FBI 91-194. Some computing resources were provided under National Science Foundation Infrastructure Grant number CDA-8722788. Part of this work was done with the facilities of Digital Equipment Corporation, and their support is gratefully acknowledged. Likewise, the hospitality of the Swedish Institute of Computer Science is gratefully acknowledged.

1. Introduction

Aquarius Prolog is a high-performance, multiple-platform Prolog implementation developed since 1989 by the Aquarius Project at the University of California, Berkeley and the Trojan Project at the University of Southern California. Aquarius Prolog includes

- a state-of-the-art Prolog compiler. The compiler is built around an execution model for Prolog, the Berkeley Abstract Machine (BAM), that retains desirable features of the Warren Abstract Machine (WAM), a standard execution model for Prolog, but allows for significant further optimizations and is considerably easier to map onto actual general-purpose machines;
- a back-end that maps the BAM onto various actual general-purpose machines;
- a run-time system offering substantially the same built-in predicates and memory management as Quintus Prolog, a leading commercial system, with additions such as two kinds of destructive assignment;
- a top-level for program development;
- source code and notes. It is hoped these will stimulate and support experiments and extensions.

This user manual characterizes these facilities for Prolog programmers, that is, it deals with how to use Aquarius Prolog, not how Aquarius Prolog works internally. To find out about the latter, consult the source code and notes and the references at the end of this chapter.

1-1. Overview of the User Manual

This document is meant primarily as a reference, something to consult when you have a question about what Aquarius Prolog can or should do. Throughout, it assumes you are acquainted with Prolog. For an introducation to Prolog, try W. F. Clocksin and C. S. Mellish, Programming in Prolog, second edition, Springer-Verlag, 1984, hereafter occasionally cited as Clocksin & Mellish.

In this document, chapter 2 explains how to compile and run programs in Aquarius Prolog, and chapter 3 explains how to load and interpret programs; first-time users should examine these before attempting much else. The rest of the user manual describes the syntax of Aquarius Prolog (chapter 4), error behavior (chapter 5), built-in predicates and control structures (chapters 6-18) and memory management (chapter 19). Appendices summarize limits of Aquarius Prolog (appendix A), error data (appendix B) and prospective improvements (appendix C).

1-2. Notational Conventions

The user manual entry for each built-in predicate or control structure begins with a goal template such as

numbervars(*?Term*, *+FirstVar*, *−LastVar*)

Term, *FirstVar* and *LastVar* are meta-variables naming the arguments of a generic numbervars/3 goal. The characters preceding them are mode annotations; their meaning is the same here as in Quintus Prolog documentation:

\+ This argument is an input. It should be initially instantiated, or an error condition occurs.

− This argument is an output. Any value supplied for this argument is unified with the actual output value from the predicate or control structure, and if this unification fails, the goal fails.

? This argument is neither always an input nor always an output. It may be either, and it may be initially instantiated or not, depending on the purpose of the goal.

1-3. References

See the following for information about the internal workings and history of Aquarius Prolog:

Ralph Clarke Haygood and Peter Van Roy, ‘‘Aquarius Prolog: from Compiler to System,’’ *in preparation*.

Bruce K. Holmer, Barton Sano, Michael Carlton, Peter Van Roy, Ralph Haygood, Joan M. Pendleton, Tep Dobry, William R. Bush and Alvin M. Despain, ‘‘Fast Prolog with an Extended General Purpose Architecture,’’ The 17th International Symposium on Computer Architecture, IEEE Computer Society Press, 1990, pp. 282-291.

Peter Lodewijk Van Roy, ‘‘Can Logic Programming Execute as Fast as Imperative Programming?’’ Report No. UCB/CSD 90/600, Ph.D. dissertation, University of California, Berkeley, December 1990.

Peter Van Roy and Alvin M. Despain, ‘‘High Performance Logic Programming with the Aquarius Prolog Compiler,’’ IEEE Computer, IEEE Computer Society Press, January 1992, pp. 54-68.

2. Compiling Prolog to Native Code

Aquarius Prolog provides two ways to run a program. You may compile the program to a native-code executable file and run the file, or you may load the source code of the program and interpret it. Compiling generally takes longer than loading, but compiled programs generally run faster than interpreted programs. This chapter explains how to compile and run programs; see chapter 3 for how to load and interpret programs.

2-1. apc

Aquarius Prolog comes with a script

<installation directory>/apc/installed/*<platform>*/apc.*<platform>*,AQUARIUS

apc is a C-shell script that implements Prolog-to-native-code compilation. To run it from most shells, simply enter the script name. For example, if the current working directory is the Aquarius Prolog installation directory of a MIPS R3000, RISC/os machine,

```
% apc/installed/MIPS,RISCOS/apc.MIPS,RISCOS,AQUARIUS
```

runs apc. The examples below assume your command interpreter is the C-shell and you have aliased apc to *<installation directory>*/apc/installed/*<platform>*/apc.*<platform>*,AQUARIUS.

apc accepts various options and file names on the command line. According to these, it runs the Aquarius Prolog compiler and back-end and the standard assembler and linker for the platform on which the program is to run and generates an executable and/or files intermediate between that and Prolog.

As a first example, suppose our program is wholly in a single file foo.pl; then

```
% apc -v foo.pl
compiling foo.pl
mapping foo.b
assembling foo.s
linking foo.o
%
```

produces an executable named foo in the current working directory. The -v (**v** for **v**erbose) option causes apc to echo a message at the start of each step of the compilation scheme. These messages include the names of intermediate files generated along the way, as follows: foo.b, the output of the compiler, contains Berkeley Abstract Machine

(BAM) code; foo.s, the output of the back-end, contains assembly code; foo.o, the output of the assembler, contains object code. By default, apc removes the intermediate files before it exits, leaving only foo.pl, the Prolog file, and foo, the executable. You may retain the intermediate files if you wish by specifying the -sb (**sb** for **s**ave .**b**), -ss (**ss** for **s**ave .**s**) and -so (**so** for **s**ave .**o**) options to apc. Also, you may cause apc to exit after generating any of these files by specifying the -tb (**tb** for **t**hrough .**b**), -ts (**ts** for **t**hrough .**s**) or -to (**to** for **t**hrough .**o**) option. By default, with the -ts or -to option, apc removes the intermediate file or files before it exits, leaving only foo.pl and foo.s or foo.o.

As a further example, suppose our program is in two files bar.pl and baz.pl; then

```
% apc -v bar.pl baz.pl
compiling bar.pl
compiling baz.pl
mapping bar.b baz.b
assembling bar.s
linking bar.o
%
```

produces an executable named bar in the current working directory. By default, the name of the executable is the root name of the first source file. You may override the default if you wish by specifying the -o output_name (**o** for **o**utput) option to apc.

In the preceding example, as the messages from apc indicate, bar.pl and baz.pl pass through the compiler separately, but the outputs bar.b and baz.b pass through the back-end together. By default, any number of files can pass through the compiler separately, but all the outputs must pass through the back-end together. This is because the back-end maps symbolic constants (atoms and functors) to machine integers, and the mapping must be consistent throughout any program, so by default, the back-end operates on whole programs at a time. It would be preferable for Aquarius Prolog to have its own linker in place of the standard linker to permit separate processing up to linking, as in other languages. Perhaps some day...but for now, (1) the restriction is not as inconvenient as it may appear, in that for large programs, the compiler usually consumes more time than the back-end, assembler and linker combined, so separate processing is possible up to a temporally late juncture in the compilation scheme, and (2) the restriction is not absolute, in that a form of separate processing up to linking is possible by running the back-end in library mode instead of program mode, the default. To demonstrate how apc packages these possibilities, as a final example, suppose donald.pl is a ‘‘library’’ file we wish to use with a pair of ‘‘application’’ files chip.pl and dale.pl; then, exploiting possibility (1),

```
% apc -v -tb donald.pl
compiling donald.pl
% apc -v chip.pl donald.b
compiling chip.pl
mapping chip.b donald.b
assembling chip.s
linking chip.o
```

```
% apc -v dale.pl donald.b
compiling dale.pl
mapping dale.b donald.b
assembling dale.s
linking dale.o
%
```

produces executables chip and dale without recompiling donald.pl each time. The -tb option causes apc to exit after running the compiler, in this case generating donald.b. Then we specify donald.b to each subsequent run of apc. Since donald.b is a source file for each subsequent run, apc does not remove it before it exits even though donald.b is not a Prolog file. This approach extends easily. For example, having compiled donald.pl as above, to use another ‘‘library’’ file ludwig.pl along with it and another ‘‘application’’ file daisy.pl, we may compile ludwig.pl just like donald.pl and specify ludwig.b along with donald.b to a subsequent run of apc for daisy.pl, as follows:

```
% apc -v -co ludwig.pl
compiling ludwig.pl
% apc -v daisy.pl ludwig.b donald.b
compiling daisy.pl
mapping daisy.b ludwig.b donald.b
assembling daisy.s
linking daisy.o
%
```

It remains inconvenient to remap and reassemble the contents of donald.b whenever we need them for another program, especially if donald.b is large. We may avoid this by means of the second possibility above; exploiting possibility (2),

```
% apc -v -l donald.pl
compiling donald.pl
mapping (library mode) donald.b
assembling donald.s
% apc -v -x donald.x chip.pl donald.o
compiling chip.pl
mapping chip.b
assembling chip.s
linking chip.o donald.o
% apc -v -x donald.x dale.pl donald.o
compiling dale.pl
mapping dale.b
assembling dale.s
linking dale.o donald.o
%
```

produces executables chip and dale without recompiling, remapping or reassembling donald.pl each time. The -l (**l** for **l**ibrary) option causes apc to run the back-end in library mode, in this case generating not only donald.s but also donald.x, a kind of file called a chain file. donald.x contains Prolog terms that represent the symbolic constant to machine integer mapping between donald.b and donald.s; subsequently, the back-end can read this and apply the mapping to files such as chip.b or dale.b to produce complete programs with consistent mappings. The -x chain_name (**x** for chain, though the letters do not match) option tells apc which chain file to use. This is the mechanism by which user programs are ‘‘chained’’ to the built-in predicates of Aquarius Prolog, except that for this purpose there is no need to specify the -x chain_name option to apc; by default, apc uses the chain file associated with the built-in predicates. This approach does not extend quite so easily, as only one chain file can be used at a time. For example, having compiled donald.pl in library mode as above, to use ludwig.pl along with it and daisy.pl, we may compile ludwig.pl in library mode with chain file donald.x and pass along the chain file that results to a subsequent run of apc for daisy.pl, as follows:

```
% apc -v -x donald.x -l -o ludwig+donald.x ludwig.pl
compiling ludwig.pl
mapping (library mode) ludwig.b
assembling ludwig.s
% apc -v -x ludwig+donald.x daisy.pl ludwig.o donald.o
compiling daisy.pl
mapping daisy.b
assembling daisy.s
linking daisy.o ludwig.o donald.o
%
```

The first run of apc generates ludwig.o and a chain file ludwig+donald.x, a name meant to signify that the chain file contains data for both ludwig and donald. The -o output_name option specifies the name, which by default would be ludwig.x; library mode compilations do not produce executables, so the -o output_name option is available for chain files instead. The final apc command line has become rather cumbersome. If there are many ‘‘application’’ files like daisy.pl, it may be well to define an alias

```
% alias apcld 'apc -x ludwig+donald.x \!* ludwig.o donald.o'
%
```

so that we may say

```
% apcld daisy.pl
%
```

to accomplish the compilation above.

The order of options and file names on the command line is unimportant to apc, apart from the executable name convention described above. Some combinations of options and file names make no sense, but apc does not check for this.

2-2. Preparing Programs for Compilation

The Aquarius Prolog compiler has certain conventions, restrictions and features in which it differs from the built-in predicates for loading programs at execution time (consult/1 and company, section 3-1). To compile and run correctly and efficiently, programs should take account of these, as follows:

- The entry point for execution of a compiled program is the first predicate in the first file (on the apc command line) of the program. This should be of arity zero; otherwise, the arguments are undefined.
- The compiler does not support term_expansion/2 for Prolog source code transformations. (It does support grammar rules; see chapter 16.)
- The compiler does not support arbitrary embedded commands, that is, a term of the form

  :– *X*.

  in an input file is an error unless *X* is a compiler directive or one of a small collection of acceptable embedded commands. Section 2-2-1 discusses compiler directives, and section 2-2-2 discusses acceptable embedded commands.
- Predicates to be modified at execution time (using assert/[1,2] and company; see chapters 12 and 13) must be declared dynamic; by default, predicates defined using the compiler are static. Section 2-2-3 discusses the distinction and dynamic declarations.
- Programs that do not use floating point numbers (floats) may disable the float compiler option; the compiler then assumes integer arithmetic only. Section 2-2-4 discusses this in detail.
- The compiler offers an optional global dataflow analysis phase to improve the compilation by deriving mode information about predicate arguments. To use this feature, programs must enable the analyze compiler option and declare entry points for analysis. Even programs that do not use analysis can obtain the same kinds of benefits by means of mode declarations. Section 2-2-5 discusses analysis and entry and mode declarations.

Users familiar with the Quintus Prolog Runtime Generator should note that in contrast with programs compiled using that system, programs compiled using the Aquarius Prolog compiler:

- have access to the same collection of built-in predicates as interpreted programs;
- are subject to the same error conditions to which they respond in the same way as interpreted programs;

- commence execution with operator declarations and flag values specified by embedded op/3 and prolog_flag/3 commands in effect, just like interpreted programs;
- respond to control-C interrupts just like interpreted programs.

It is generally good programming style for the clauses of a predicate to be contiguous in an input file, but the compiler does not require this. The compiler reads a whole input file before compiling any of it.

2-2-1. Compiler Directives

Compiler directives are terms of the form

:– *Directive*.

in an input file. The following directives are recognized:

help	Print information about the recognized directives.
include(*FileName*)	Include the file named *FileName* in the input. *FileName* is an atom that is an absolute or relative file name.
option(*Options*)	Enable the options specified by *Options*. *Options* is an option or list of options. Options are discussed below.
notoption(*Options*)	Disable the options specified by *Options*. *Options* is an option or list of options. Options are discussed below.
options	Print the currently enabled options. Options are discussed below.
helpoptions	Print information about the recognized options.
dynamic *Predicates*	Declare the predicates specified by *Predicates* to be dynamic. *Predicates* is a predicate specification of the form *Name*/*Arity* or a sequence of such predicate specifications separated by commas.
entry((*Head* :– *Formula*))	Declare *Formula* to be the mode formula of the predicate corresponding to *Head* and use this as an entry point for analysis.
mode((*Head* :– *Formula*))	Declare *Formula* to be the mode formula of the predicate corresponding to *Head*.
entry(*Head*, *Required*, *Before*, *After*, *Simple*)	More detailed entry declaration.

mode(*Head*, *Required*, *Before*, *After*, *Simple*)
: More detailed mode declaration.

The following options are recognized:

float
: Generate floating point arithmetic as necessary; enabled by default.

analyze
: Perform global dataflow analysis; disabled by default.

compile
: Compile the input; enabled by default. Disabling this option but enabling the analyze option causes the compiler to print mode information derived by analysis in a form suitable for inclusion in a subsequent compiler input.

arithmetic_error_check
: Generate checks for errors in arithmetic operations, which the compiler handles ''in-line'' rather than by explicit calls to evaluation and comparison predicates; enabled by default. Disabling this option usually leads to smaller, faster code, but the code is unsafe in that errors in arithmetic operations may yield errant results or hardware traps.

same_number_solutions
: Retain the number of solutions on backtracking required by standard Prolog; enabled by default. Relaxing the semantics by disabling this option sometimes leads to smaller, faster code.

same_order_solutions
: Retain the order of solutions on backtracking required by standard Prolog; enabled by default. Relaxing the semantics by disabling this option sometimes leads to smaller, faster code.

Various other directives and options are recognized that are not ordinarily of interest to users. To find out about these, consult the implementation notes accompanying the compiler.

2-2-2. Embedded Commands

Embedded commands are terms of the form

:– *Command*.

in an input file, where *Command* is not a compiler directive. Only the following commands are recognized:

op(*Precedence*, *Type*, *Ops*)

prolog_flag(*Flag*, *Old*, *New*)

The purpose of the second is to permit character escapes (section 4-3) in an input file. All other commands lead to warnings and are otherwise ignored.

The recognized commands have both an immediate effect and an effect when the program runs. In the first place, they declare operators or set flag values that affect the reading of subsequent clauses by the compiler. Moreover, the resulting operator declarations or flag values persist when the program runs.* For example, if the input file contains

```
:- op(500, xfx, and).
main :- write(wait and see), nl.
```

the program writes

```
wait and see
```

when it runs.

2-2-3. Dynamic and Static Predicates

In many Prolog implementations, such as Quintus Prolog, a predicate is either dynamic or static. Dynamic predicates can be incrementally modified by adding or removing individual clauses. Static predicates, however, can be changed only by complete redefinition. Aquarius Prolog, too, makes this distinction, but Aquarius Prolog differs somewhat from Quintus Prolog, for example, in how a predicate becomes dynamic or static and in what operations are possible on either kind of predicate.

In Aquarius Prolog, predicates defined at execution time are always dynamic. Such definitions can be made using consult/1 and company (section 3-1), assert/[1,2] (section 13-1), asserta/[1,2] (section 13-2) or assertz/[1,2] (section 13-3). This is different from Quintus Prolog, whose consult/1 as well as compile/1 make predicates static by default. Also, whereas in Quintus Prolog, consult/1 completely redefines predicates, in Aquarius Prolog, consult/1 extends any existing definitions of predicates. Aquarius Prolog provides reconsult/1 to redefine predicates completely.

Predicates defined using the compiler are static unless explicitly declared dynamic as described below. As in Quintus Prolog, it is impossible to add, examine or remove individual clauses of a static predicate at execution time. It is further impossible to abolish a static predicate in Aquarius Prolog, unlike Quintus Prolog.

* This functionality is built into apc. If you run the compiler apart from apc, you must issue init_directive directives to obtain it; examine apc for details.

A dynamic declaration has the form

:– dynamic *Predicates*.

where *Predicates* is a predicate specification of the form *Name*/*Arity* or a sequence of such predicate specifications separated by commas. Note that dynamic is a built-in prefix operator, and dynamic/1 is not a predicate, so this form makes sense only as a directive. A dynamic declaration in an input file advises the compiler that the specified predicates are to be dynamic rather than the default static. Both definitions of and calls to the predicates are treated differently in this case. Dynamic declarations are also useful in files that are loaded at execution time. This is because a dynamic declaration is considered part of the definition of the specified predicates. If you call a predicate that you have declared dynamic but for which you have not provided any clauses yet, you will not get an unknown predicate warning, even if the unknown_predicate flag is warning (section 11-6), because the system regards predicates that have been declared dynamic as known.

Dynamic predicates are interpreted, even if they are defined using the compiler, so they generally run slower than static predicates, which are transformed to native code. First argument indexing of dynamic predicate clauses mitigates the interpretation penalty to some extent. For more about dynamic predicates, see chapters 12 and 13.

2-2-4. Floating Point Arithmetic

If the compiler knows when it encounters an arithmetic expression in an input file that neither operands nor result can be floats, it can generate smaller, faster code to evaluate the expression than if there were a possibility of floats. The float compiler option allows programs that do not use floats at all to take advantage of this. When the float option is disabled, the compiler assumes programs use integer arithmetic only. Even when the float option is enabled, as it is by default, the compiler does the best it can to avoid superfluous type checks and conversions, but the result is generally not as efficient in space or time. It helps to include mode declarations to let the compiler know that certain predicate arguments are always integers or always floats.

The float option applies to the whole input file, regardless of where in the input file the enabling directive appears. It pertains to whole files rather than individual predicates in files. If you require floats in one part of a program but not another, it may be worthwhile to put the parts in separate files.

You must not disable the float option if you use even a single float. Otherwise, the results will be errant.

For more about arithmetic, see chapter 9.

2-2-5. Global Dataflow Analysis and Mode Declarations

Programs that enable the analyze option are compiled with a global dataflow analysis phase, which derives mode information about predicate arguments. Besides enabling the analyze option, programs that use analysis must declare entry points for analysis, in the following sense. A program generally consists of predicates in several files. In each file, there are usually some predicates that may be called from outside the file, while the rest are known only within the file. The former constitute the entry set of the file. On the basis of what it learns from entry declarations about arguments of the predicates in the entry set, analysis infers information about arguments of the other predicates in the file, which it assumes are reachable only through the entry set. Entry declaration forms and the modes that may be specified using them are described below.

Analysis has the following characteristics:

- The analyze option applies to the whole input file, regardless of where in the input file the enabling directive appears. It pertains to whole files rather than individual predicates in files. If you desire analysis in one part of a program but not another, you must put the parts in separate files.
- All (static) predicates of arity zero are considered entry points. You need not include entry declarations for these.
- If some predicate calls a predicate outside the file being compiled, analysis assumes minimal-information modes for arguments of the called predicate, unless the file being compiled includes a mode declaration for the called predicate, or the called predicate is a built-in predicate, in which case analysis consults a database of information about built-in predicates.
- Dynamic predicate definitions are not analyzed. Since dynamic predicates can change during execution, their behavior cannot be determined at compile time. If a static predicate calls a dynamic predicate, analysis assumes minimal-information modes for the dynamic predicate unless there is a mode declaration for it. If a static predicate may be called by a dynamic predicate, you should include an entry declaration with minimal-information modes for the static predicate. Such an entry declaration has the form

  :– entry((*Head* :– true)).

  or

  :– entry(*Head*, true, true, true, n).
- call/1, setof/3, bagof/3 and findall/3 may call any predicate with any arguments, and in general it cannot be determined at compile time which predicates will be called with what kinds of arguments. For programs that use these built-in predicates, there are two cases to consider:

(1) Only dynamic predicates will be called; then analysis is unaffected, because dynamic predicate definitions are not analyzed.

(2) Some static predicates will be called; then you should include entry declarations with minimal-information modes for these predicates.

- Destructive assignment (using setarg/3 or steparg/3 and step/0; see chapter 17) is incompatible with analysis. You should not perform analysis on programs that use destructive assignment.
- The compiler prints mode information derived by analysis at the beginning of the output. If the compile option is disabled, the information is contained in mode declarations suitable for inclusion in a subsequent compiler input; otherwise, it is contained in comment lines ignored by the back-end, as in the example below.

Analysis is usually worthwhile, as it usually reduces compile time, code size and execution time. However, it must be used with care because missing or incorrect entry declarations can lead to errant code. This is why the analyze option is disabled by default.

A basic entry declaration has the form

:– entry((*Head* :– *Formula*)).

where *Head* corresponds to a predicate and *Formula* is a mode formula constructed according to table 2-1.

It is possible to include a mode declaration for a predicate that is not to be an entry point for analysis. A basic mode declaration has the form

:– mode((*Head* :– *Formula*)).

where *Head* corresponds to a predicate and *Formula* is a mode formula constructed according to table 2-1. This is useful for specifying modes to analysis for predicates outside the input file. If the predicate is within the input file, analysis results are added to the information in the mode declaration.

More detailed entry and mode declarations have the forms

:– entry(*Head*, *Required*, *Before*, *After*, *Simple*).

and

:– mode(*Head*, *Required*, *Before*, *After*, *Simple*).

Head corresponds to a predicate. *Required* is the mode formula enforced by the compiler. Only uninit, uninit_reg and deref modes are acceptable here, since only these modes can be enforced by the compiler. *Before* is the mode formula known to apply before any call. Usually, modes specified with basic entry and mode declarations are of this sort. *After* is the mode formula known to apply after any call. Simple is y or n according to whether or not this predicate adheres to the simple_call protocol of the

BAM. Some built-in routines written in BAM or assembly code employ this protocol, but user predicates ordinarily employ the call protocol instead. For more information about these technicalities, consult the implementation notes accompanying the compiler.

It is generally good programming style to place each entry or mode declaration immediately before the predicate to which it pertains, but the compiler does not require this.*

If analysis derives a mode that differs from what you specify with an entry or mode declaration, the compiler prints a warning and continues with the mode derived by analysis. Note that in no other circumstance is a mode specified with an entry declaration overridden. By contrast, modes specified with mode declarations are supplemented with modes derived by analysis.

As an example, consider the following program, which solves the 16-disk towers of Hanoi problem:†

```
hanoi_16 :- hanoi(16).

hanoi(N) :- move(N, a, b, c).

move(0, _, _, _) :- !.
move(N, A, B, C) :- N1 is N-1, move(N1, A, C, B), move(N1, C, B, A).
```

This is correct, but we can make the code smaller and faster by adding the mode declaration

```
:- mode((move(N,_,_,_):-nonnegative(N))).
```

We can do better still with analysis. Replacing the mode declaration with

```
:- option(analyze).
```

and compiling yields (in part)

```
% Modes generated:
% mode(hanoi_16,true,true,true,n)
% mode(move(D,C,B,A),true,(ground(D),rderef(D),ground(C),rderef(C\
),ground(B),rderef(B),ground(A),rderef(A)),(ground(D),rderef(D),g\
round(C),rderef(C),ground(B),rderef(B),ground(A),rderef(A)),n)
% mode(hanoi(A),true,(ground(A),rderef(A)),(ground(A),rderef(A)),\
n)
```

* BAM-code macros also may have mode declarations. The mode declaration for a BAM-code macro must precede the macro declaration. Consult the implementation notes accompanying the compiler for details.

† The *N*-disk towers of Hanoi problem is to move a tower of *N* (punctured) disks of various diameters from one peg to another with the help of an auxiliary peg and according to the rules (1) only one disk can be moved at a time and (2) a larger disk cannot be put on top of a smaller disk.

Table 2-1. Mode Formulas

var(*A*)	*A* is an uninstantiated variable.
uninit(*A*)	*A* is an uninitialized memory variable. At the Prolog level, *A* is an uninstantiated variable known not to be an alias for another variable. In the implementation, *A* is a pointer to an empty memory location. Instantiating this variable is a store with no dereferencing or trailing.
uninit_reg(*A*)	*A* is an uninitialized register variable. At the Prolog level, *A* is equivalent to an uninitialized memory variable. In the implementation, *A* is an empty machine register. This mode is often possible for ''return value'' arguments, particularly from routines in BAM or assembly code.
nonvar(*A*)	*A* is instantiated, that is, not an uninstantiated variable.
atom(*A*)	*A* is an atom.
atomic(*A*)	*A* is an atomic term.
compound(*A*)	*A* is a compound term.
cons(*A*)	*A* is a cons cell.
float(*A*)	*A* is a floating point number.
ground(*A*)	*A* is a ground term, that is, a term containing no uninstantiated variables.
integer(*A*)	*A* is an integer.
list(*A*)	*A* is a cons cell or the empty list.
negative(*A*)	*A* is a negative (strictly less than 0) integer.
nonnegative(*A*)	*A* is a nonnegative (greater than or equal to 0) integer.
number(*A*)	*A* is an integer or a floating point number.
simple(*A*)	*A* is a variable or an atomic term.
structure(*A*)	*A* is a compound term other than a cons cell.
functor(*A*,*B*,*C*)	*A* is a compound term whose principal functor has name *B* and arity *C*.
A<relation>B	*A* and *B* are in relation *<relation>*, one of ==, \==, @<, @>=, @> or @=<. Either *A* or *B* may be an atomic term.
A<relation>B	*A* and *B* are in relation *<relation>*, one of =:=, =\=, <, >=, > or =<. Either *A* or *B* may be a number.
deref(*A*)	*A* is dereferenced, that is, there is no pointer chain to follow in order to reach the value of *A*.
rderef(*A*)	*A* is recursively dereferenced, that is, *A* is dereferenced, and if *A* is a compound term, every argument of *A* is recursively dereferenced.
true	This indicates no mode information is known.
fail	This indicates execution can never reach this point.
(*F1*,*F2*)	*F1* **and** *F2*, where *F1* and *F2* are mode formulas.
(*F1*;*F2*)	*F1* **or** *F2*, where *F1* and *F2* are mode formulas.
not(*F*)	**not** *F*, where *F* is a mode formula.

Note that hanoi_16/0 is the entry point for analysis. In typical fashion, analysis has derived ground and rderef modes. uninit, uninit_reg, deref and rderef modes are somewhat tricky to specify correctly ‘‘by hand,’’ but analysis routinely derives them, and they are important for high-quality code. The highest quality code frequently results from a combination of mode declarations and analysis. In this case, retaining the mode declaration, adding analysis and compiling yields (in part)

```
% Modes generated:
% mode(hanoi_16,true,true,true,n)
% mode(move(D,C,B,A),true,(rderef(A),ground(A),rderef(B),ground(B\
),rderef(C),ground(C),rderef(D),ground(D),nonnegative(D)),(ground\
(D),rderef(D),ground(C),rderef(C),ground(B),rderef(B),ground(A),r\
deref(A)),n)
% mode(hanoi(A),true,(rderef(A),ground(A),nonnegative(A)),(ground\
(A),rderef(A)),n)
```

Note that our nonnegative(*N*) has been included in the *Before* mode formula of move/4. This helps the code by eliminating a type check.

2-3. Running Compiled Programs

An executable produced using apc is completely self-contained. To run it from most shells, simply enter the file name. Execution begins with the first predicate in the first file (on the apc command line), which should be of arity zero, and continues until this predicate succeeds or fails or an error occurs. If it succeeds, the exit code is 0, if it fails, 1, otherwise, −1.

Compiled programs can access command line arguments using unix/1 (section 18-1).

3. Loading and Interpreting Prolog

This chapter explains how to load and interpret programs in Aquarius Prolog, and it describes the Aquarius Prolog top-level, the context in which programs are loaded and interpreted most often.

3-1. consult(*+Files*), reconsult(*+Files*), [*+File*|*+Files*], ensure_loaded(*+Files*)

Aquarius Prolog provides several built-in predicates for loading clauses from files at execution time. Each file is loaded in one of two modes, consult or reconsult; the mode determines what happens to existing clauses of predicates for which new clauses are encountered in the file. In consult mode, the new clauses are appended to the existing clauses (as by assertz/[1,2], section 13-3); in reconsult mode, the existing clauses are removed (as by retractall/1, section 13-6). consult/1 loads files in consult mode, while reconsult/1 loads files in reconsult mode; these two predicates are otherwise identical. '.'/2 and ensure_loaded/1 can load files in either mode, as explained below.

The argument *Files* is either a file name or a list of file names, where a file name is one of

FileName	where *FileName* is an atom that is an absolute or relative file name;
library(*LibraryName*)	where *LibraryName* is an atom that is a relative file name. This file will be sought in the paths specified by the user-definable predicate library_directory/1 (section 3-2);
user	where user is considered the name of a pseudofile with which the standard input and output streams are permanently associated.

The standard input and output streams with which the pseudofile user is associated are usually connected to the user's terminal. Under '.'/2 and ensure_loaded/1, each file specified by *Files* is loaded in consult mode unless the file name is preceded by –, in which case the file is loaded in reconsult mode.

Thus:

- consult(*Files*) loads predicate clauses from each of *Files*, extending existing definitions of these predicates (if any).
- reconsult(*Files*) loads predicate clauses from each of *Files*, removing existing definitions of these predicates (if any).
- [*File*|*Files*], where *File* is a file name optionally preceded by −, is shorthand for (re)consult([*File*|*Files*]), where existing definitions are extended by default or removed when a file name is preceded by −.
- ensure_loaded(*Files*) loads clauses from each of *Files* that has not already been loaded, extending existing definitions by default or removing existing definitions when a file name is preceded by −.

There are a number of differences between Aquarius Prolog and Quintus Prolog with respect to consult/1 and company. In Aquarius Prolog, unlike Quintus Prolog:

- Files named with a .pl suffix are not automatically associated with their root names.
- No style checking is performed.
- Predicates defined using consult/1 and company are always dynamic.
- No multifile declaration is necessary in order to load clauses for one predicate from more than one file.
- consult/1 extends rather than completely redefines predicates.

Note that Quintus Prolog does not provide reconsult/1; instead, Quintus consult/1 has the function of Aquarius reconsult/1, and there is no Quintus built-in predicate that has the function of Aquarius consult/1. Also, Quintus Prolog does not support file names preceded by −. In both respects, Aquarius Prolog is like C-Prolog, a popular noncommercial system.

In a file processed by consult/1 and company, a term of the form

:− *X*.

constitutes a compiler directive or embedded command. For compatibility with the Aquarius Prolog compiler, recognized compiler directives (section 2-2-1) are accepted but ignored except for dynamic declarations and include directives. If *X* is not a compiler directive, it is executed immediately, as by call/1 (section 6-7). Unlike the compiler, which recognizes only a few embedded commands, consult/1 and company permit any term as an embedded command. Also, a term of the form

?− *X*.

constitutes an embedded command for any term *X*; compiler directives must use ':−'/1. If an embedded command fails, a warning message is written, but loading continues.

A dynamic declaration

:– dynamic *Predicates*.

declares the predicates specified by *Predicates* to be dynamic. *Predicates* is a predicate specification of the form *Name*/*Arity* or a sequence of such predicate specifications separated by commas. Note that dynamic is a built-in prefix operator, and dynamic/1 is not a predicate, so this form makes sense only as a directive. Predicates defined at execution time are always dynamic, so a dynamic declaration in a file processed by consult/1 and company is useful only in that a dynamic declaration is considered part of the definition of the specified predicates. If you call a predicate that you have declared dynamic but for which you have not provided any clauses yet, you will not get an unknown predicate warning, even if the unknown_predicate flag is warning (section 11-6), because the system regards predicates that have been declared dynamic as known.

An include directive

:– include(*FileName*)

loads the file named *FileName* in the current mode, consult or reconsult. *FileName* is an atom that is an absolute or relative file name. At present, a relative file name is resolved with respect to the current working directory rather than the directory containing the file in which the directive is encountered. This defect, in which Aquarius Prolog differs from Quintus Prolog, may eventually be corrected.

When a syntax error is encountered, a message is written and loading continues, starting immediately after the full-stop that terminated the erroneous input.

If *Files* is (or includes) user, the standard input stream is read. A prompt '| ' is written before each new clause is read, and if a clause occupies more than one line, another prompt '   ' (five spaces) is written before each new line is read. When you finish typing in new clauses, type the end of file character (usually control-D) to exit the loading goal.

If *Files* is not instantiated or if *Files* is a list of which any element is not instantiated, an instantiation fault occurs. If *Files* is instantiated to something other than a legal file name or list of file names as described above, a type failure occurs. If a file specified by *Files* cannot be opened for any reason, then if the file_error flag is set to error (section 11-6), an I/O error occurs, or if the file_error flag is set to fail, a simple failure occurs. An attempt to load a definition of a static or built-in predicate is a system restriction violation. Other error conditions may arise from execution of embedded commands. See chapter 5 for a discussion of error conditions.

3-2. library_directory(*?LibraryDirectory*)

You may define library_directory/1 to specify a sequence of file system paths to be searched when a file name of the form library(*LibraryName*) is given to a loading, input or output predicate. The paths, which should be atoms representing absolute or relative path names, are searched in the order in which they are returned by library_directory/1.

library_directory/1 is not predefined; Aquarius Prolog differs from Quintus Prolog in this respect.

3-3. source_file(*?AbsoluteFile*)

source_file(*AbsoluteFile*) succeeds when *AbsoluteFile* is the absolute name of a file from which predicate clauses have been loaded using consult/1 and company. If *AbsoluteFile* is not instantiated, backtracking generates every possible solution.

If *AbsoluteFile* is instantiated to something other than an atom, a type failure occurs.

3-4. source_file(*?Head*, *?AbsoluteFile*)

source_file(*Head*, *AbsoluteFile*) succeeds when *AbsoluteFile* is the absolute name of a file from which clauses of a predicate whose skeletal specification unifies with *Head* have been loaded using consult/1 and company. If either *Head* or *AbsoluteFile* is not instantiated, backtracking generates every possible solution.

If *Head* is instantiated to something other than a legal clause head or if *AbsoluteFile* is instantiated to something other than an atom, a type failure occurs.

3-5. The Top-Level

Aquarius Prolog comes with an executable

<installation directory>/top_level/installed/top_level.*<platform>*,AQUARIUS

The top-level is an interactive interface to the Aquarius Prolog interpreter. To run it from most shells, simply enter the file name. For example, if the current working directory is the Aquarius Prolog installation directory of a MIPS R3000, RISC/os machine, begin a session as follows:

```
% top_level/installed/top_level.MIPS,RISCOS,AQUARIUS

Aquarius Prolog version 1.0 (MIPS R3000, RISC/os)

| ?-
```

The prompt '| ?- ' indicates the interpreter is ready to accept input. Any term you enter will be executed immediately, a s by call/1 (section 6-7). To run a program, invoke one of consult/1 and company to load it, and invoke the top-level predicate of the program to interpret it. For example:

```
| ?- consult(user).
| hello :- write('hello, world'), nl.
| ^D

yes
| ?- hello.
hello, world

yes
| ?-
```

The top-level writes yes after each goal that succeeds and no after each goal that fails. For example:

```
| ?- 0 =:= 1.

no
| ?-
```

If a goal contains uninstantiated variables and the goal succeeds, the top-level writes the instantiations of the variables after the success. For example:

```
| ?- length(List, Length).

List = [],
Length = 0
```

length/2 is a built-in predicate. When invoked with neither argument instantiated, it generates longer and longer lists on backtracking. The solution written is only the first of the sequence. To backtrack and see the second, type ;:

```
| ?- length(List, Length).

List = [],
Length = 0 ;

List = [_1232],
Length = 1
```

You could type ; again to backtrack and see the third solution, and so on. If you do not want to backtrack and see another solution, type a newline character, a nd you may proceed to something else:

```
| ?- length(List, Length).

List = [],
Length = 0

| ?-
```

If you type ; to backtrack and no further solution exists, the top-level writes no, a s after any other failure. For example:

```
| ?- X = foo.

X = foo ;

no
| ?-
```

To end a session, type the end of file character (usually control-D).

Throughout the rest of the user manual, the behavior of built-in predicates is illustrated by means of their behavior under the top-level. Behavior under the top-level should not differ from behavior in any other context, with one exception: abort/0 (section 6-14) under the top-level returns to the top-level, which is convenient for program development, but abort/0 in any other context terminates execution (with exit code −1).

4. Syntax

Aquarius Prolog recognizes and generates almost the same syntax as Quintus Prolog.[*] Both may be said largely to extend the syntax set forth in Clocksin & Mellish. Assuming familiarity with the latter, this chapter primarily describes the extensions Aquarius Prolog provides.

4-1. Terms

A term is either a variable, a constant or a compound term. A constant is either a number or an atom. A number is either an integer or a floating point number (float). A compound term is either a structure or a list. The following paragraphs discuss each type of term successively.

A variable is written as _*<digits>*. The number thereby associated with the variable is sensitively dependent on the internal workings of Aquarius Prolog. Because of garbage collection (chapter 19), it may vary from one time to another during execution. Therefore, programs that write terms containing multiple instances of some variable should first apply numbervars/3 (section 7-26) to the term (or a copy made using copy_term/2, section 7-27) so that each instance of the variable is sure to be written the same way.

Integers are usually read and written in base 10, but it is possible to use any base from 2 to 36. To be read properly, the digits must be preceded by the base written in base 10 followed by an apostrophe. In bases greater than 10, the letters a-z or A-Z are used for digits greater than 9. For example, 16'f is equivalent to 15. Use format/[2,3] (section 10-1-8) to write integers in bases other than 10.

As a further extension of the integer syntax, a ‘‘base’’ of 0 yields the ASCII character code of the character following the apostophe. For example, 0'* is equivalent to 42.

Floats may be read or written in nonexponential format, as in 123.456, or exponential format using e or E, as in 1.23456e+02 or 1.23456E+02. Use format/[2,3] (section 10-1-8) to write floats in any of these styles. At least one digit must precede the decimal point, and at least one digit must follow the decimal point. No base other than 10 is recognized.

---

[*] The differences are largely obscure cases. For example, if [] is defined to be a postfix operator, Aquarius Prolog can read foo[], which Quintus Prolog considers a syntax error. For compatibility with Quintus Prolog, Aquarius Prolog writeq/[1,2] generates foo'[]'. See section 4-2 for a discussion of one difference that is less obscure.

An atom may be any of the following:

- Any sequence of alphanumeric characters, including _, beginning with a lower-case letter.
- Any sequence of characters from the set {+, −, ∗, /, \, ˆ, <, >, =, ‘, ˜, :, ., ?, @, #, $, &} except /∗, which begins a comment. Note that %, ; and ! are not in this set.
- Any of ;, !, [] and {}. The bracket pairs are special; none of the characters [,], { or } is an atom by itself.
- Any sequence of characters delimited by single quotes. Two consecutive single quotes within such a quoted atom yield a single quote, as in 'isn''t'. Note that delimiting any of the above forms by single quotes does not make a new atom.

Programs should avoid using atoms of the form '$ ...', '# ...' or '@ ...', as many such atoms have special meanings within the Aquarius Prolog run-time system.

A compound term has a principal functor, often known as a functor for short, which is characterized by a name and an arity. The name is an atom, and the arity is a positive integer that counts the arguments of the term. It is important not to confuse the name of a functor with the name of an atom, which is a string (see below). A functor is denoted *Name*/*Arity* when it is necessary to refer to it explicitly. To improve the legibility of compound terms such as arithmetic expressions, names of unary functors may be declared to be prefix and/or postfix operators, and names of binary functors may be declared to be infix operators; see section 4-2 for a discussion of operators.

Lists are an important special case of compound term. Formally, a list is either the empty list [] or a compound term whose principal functor is '.'/2, for example, '.'(foo, bar). However, Aquarius Prolog reads and usually writes the latter as [foo|bar]; the standard syntax is generated only by write_canonical/[1,2] (section 10-1-5) and display/1 (section 10-1-6). If the second argument, or tail, of a list is [], as in '.'(foo, []), the other term writing predicates omit the tail, as in [foo].

A further notational convenience exists for proper lists (section 7-17) of ASCII character codes, known as strings. For example, "foo" is equivalent to [102,111,111]; this is the name of the atom foo.

4-2. Operators

Aquarius Prolog, like Quintus Prolog, permits any atom to be declared an operator of precedence between 1 and 1200 and type fx, fy, xfx, xfy, yfx, xf or yf. Briefly, operator precedence and type mean the following. The operator of highest precedence in an expression is the principal functor, and the expression is said to have that precedence. For example, if + and ∗ are defined to be infix operators of precedences 500 and 300, respectively, a+b∗c is equivalent to a+(b∗c), not (a+b)∗c and has precedence

500. An expression enclosed in parentheses is said to have precedence 0. If an expression contains two or more operators of the same highest precedence, the ambiguity is resolved by the operator types, which are meant to serve as mnemonic pictograms. A y means the corresponding operand may be of the same or lower precedence than the operator, while an x means the corresponding operand must be of strictly lower precedence than the operator. For example, if + is defined to be an operator of type yfx, a+b+c is equivalent to (a+b)+c, not a+(b+c). Applying these precedence and type rules can be laborious, so it is usually desirable to use parentheses when order is significant.

An atom may be declared an operator of more than one type up to one operator in each type class, where {fx, fy} is the prefix type class, {xfx, xfy, yfx} is the infix type class and {xf, yf} is the postfix type class. For example, − is ordinarily both a prefix and an infix operator.

A number of operators are predefined in Aquarius Prolog, as by the goal

```
?- op(1200, xfx, [(:-),-->]),
   op(1200, fx, [(:-),(?-)]),
   op(1150, fx, [(dynamic),(meta_predicate),(mode),(multifile),(public)]),
   op(1100, xfy, ;),
   op(1050, xfy, ->),
   op(1000, xfy, ','),
   op(900, fy, [+,nospy,spy]),
   op(700, xfx, [=,is,=..,==,==,@<,@>,@=<,@>=,=:=,==,<,>,=<,>=]),
   op(600, xfy, :),
   op(500, yfx, [+,-,,/]),
   op(500, fx, [+,-]),
   op(400, yfx, [*,/,//,<<,>>]),
   op(300, xfx, mod),
   op(200, xfy, ^).
```

See section 4-2-1 for a discussion of op/3. These operators may be redefined if desired, though this is seldom desirable, as output written with respect to one set of operator definitions may not be readable with respect to another set. Note that meta_predicate, mode, multifile, public, nospy and spy are predefined for Quintus Prolog compatibility only; they are otherwise irrelevant to Aquarius Prolog.

Several restrictions exist to avoid potential problems with prefix operators:

- An argument of a compound term in standard syntax must have precedence strictly less than 1000. Thus, it is necessary to write

  assert((a:–b))

  instead of

  assert(a:–b)

because :– has precedence 1200.

- There must be no space between the principal functor name of a compound term in standard syntax and the open parenthesis that follows.
- There must be at least one space between a prefix operator and an argument that is enclosed in parentheses.
- If a prefix operator is written without an argument, that is, simply as an atom, it has the precedence of the prefix operator. Thus, it is necessary to write

  ```
  predicate_property(a, (dynamic))
  ```

  instead of

  ```
  predicate_property(a, dynamic)
  ```

  because dynamic is a prefix operator of precedence 1150.

Unlike Quintus Prolog, Aquarius Prolog write/[1,2] (section 10-1-2) and writeq/[1,2] (section 10-1-3) do not write a space between an infix operator and a following argument that is enclosed in parentheses. For example, Quintus Prolog writes a*(b+c) as a* (b+c), with a space between the star and the open parenthesis; Aquarius Prolog omits the space.

4-2-1. op(*+Precedence*, *+Type*, *+Ops*)

op(*Precedence*, *Type*, *Ops*) declares the atoms specified by *Ops* to be operators of type *Type* and precedence *Precedence*. *Ops* should be an atom or a list of atoms. If *Precedence* is 0, operator definitions (if any) in the type class of *Type* for the atoms specified by *Ops* are cancelled.

If *Precedence*, *Type* or *Ops* is not instantiated, an instantiation fault occurs. If *Precedence* is instantiated to something other than an integer between 0 and 1200, if *Type* is instantiated to something other than fx, fy, xfx, xfy, yfx, xf or yf or if *Ops* is instantiated to something other than an atom or a list of atoms, a type failure occurs.

4-2-2. current_op(*?Precedence*, *?Type*, *?Op*)

current_op(*Precedence*, *Type*, *Op*) succeeds when atom *Op* is defined to be an operator of type *Type* and precedence *Precedence*. If either *Type* or *Op* is not instantiated, backtracking generates every possible solution.

Note that a nondeterminate current_op/3 goal generates solutions in arbitrary order and sees changes to the operator definition ‘‘database’’ immediately. Aquarius Prolog differs from Quintus Prolog in this respect.

If *Precedence* is instantiated to something other than an integer between 1 and 1200, if *Type* is instantiated to something other than fx, fy, xfx, xfy, yfx, xf or yf or if *Op* is instantiated to something other than an atom, a type failure occurs.

4-3 Character Escapes

Aquarius Prolog, like Quintus Prolog, provides a facility that permits escape sequences to occur within quoted atoms, strings and character codes. This facility may be enabled by the goal

```
| ?- prolog_flag(character_escapes, _, on).
```

and disabled by the goal

```
| ?- prolog_flag(character_escapes, _, off).
```

See section 11-6 for a discussion of prolog_flag/3. Character escapes are disabled by default.

With character escapes enabled, the following escape sequences are recognized within quoted atoms, strings and character codes by read/[1,2] (section 10-1-1) and consult/1 and company (section 3-1):

\b	denotes backspace	(ASCII 8).
\t	denotes horizontal tab	(ASCII 9).
\n	denotes newline	(ASCII 10).
\v	denotes vertical tab	(ASCII 11).
\f	denotes form feed	(ASCII 12).
\r	denotes carriage return	(ASCII 13).
\e	denotes escape	(ASCII 27).
\d	denotes delete	(ASCII 127).
<octal digits>	denotes the character whose code is *<octal digits>* base 8, where *<octal digits>* may be one, two or three octal digits.	
\\^*<control character>*	denotes the character whose code is the code of *<control character>* mod 32.	
<layout character>	denotes no character, where the code of *<layout character>* is less than or equal to 32 or greater than or equal to 127. Thus,	

for example, a newline in a quoted atom is ignored if it is preceded by a backslash.

`\c` denotes no character, and all characters up to but not including the next nonlayout character are ignored. This is sometimes useful when formatting quoted atoms and strings for readability.

<other character> denotes *<other character>*, where *<other character>* is any character not mentioned above. Thus, for example, a backslash should be written \\.

Note also that with character escapes enabled, the escape sequence \' within a quoted atom denotes the same character as '', namely one single quote, and the escape sequence \" within a string denotes the same character as "", namely one double quote.

Depending on whether character escapes are on or off, `writeq/[1,2]` (section 10-1-3) and `write_canonical/[1,2]` (section 10-1-4) write quoted atoms differently. With character escapes enabled:

- ASCII 9 (horizontal tab), 32 (space) and 33-38, 40-91 and 93-126 (nonlayout characters except single quote and backslash) are written as themselves.
- ASCII 8, 10, 11, 12, 13, 27 and 127 are written as *<lowercase letter>*, where the corresponding *<lowercase letter>* for each character is given above.
- ASCII 39 (single quote) is written as two consecutive single quotes.
- ASCII 92 (backslash) is written as two consecutive backslashes.
- All other characters are written as \^*<control character>*, where *<control character>* is the nonlayout character of minimum code such that the code of the character to be written is the code of *<control character>* mod 32.

With character escapes disabled:

- ASCII 39 (single quote) is written as two consecutive single quotes.
- All other characters are written as themselves.

5. Errors

Aquarius Prolog error handling is meant to be comprehensive, consistent and informative. For example, the arguments of every call to a built-in predicate are checked for appropriateness. If they are inappropriate in a way thought likely to result from a mistake of some sort, an error condition is entered and in general reported to the user. The cost of this is kept acceptable for compiled code by checking as much as possible at compile time rather than execution time. When the Aquarius Prolog compiler can determine (as it often can) that a particular condition on the arguments of a built-in goal is certainly satisfied, it tries to generate a call to a so-called modal entry point to the built-in predicate that omits checking that condition. For example, for the goal

arg(1, *X*, *Y*)

the compiler generates a call to an entry point to arg/3 (section 7-20) that assumes the first argument is appropriate (that is, instantiated to a positive integer). This modal entry point scheme is used even if global dataflow analysis (section 2-2-5) is not. However, it is likely to be significantly more effective in conjunction with analysis, since from analysis the compiler can gain considerable information about the arguments of built-in goals.

5-1. Error Conditions

An error condition is continuable if and only if when the condition arises there is some reasonable way to continue the computation. Usually, the only reasonable way to continue in Prolog is to fail the goal that produces the condition, though occasionally there are alternatives. The run-time system need not necessarily report a continuable error condition to the user. In Aquarius Prolog, you may use prolog_flag/3 (section 11-6) to specify whether to report such conditions (as warnings) and then continue (the default behavior) or simply continue with no report. A user-defined error handler (section 5-2) may also choose how to handle such conditions.

Error conditions fall into several categories:

- unknown predicate failure

  A predicate is called for which no clauses are or ever have been (at least since the most recent abolish/[1,2] of the predicate) defined. Equivalently, the predicate has no clauses and is not known to be dynamic. This is a continuable error condition - logically, the goal should simply fail. With prolog_flag/3, you may specify whether to report the condition (as a warning) and then fail (the default behavior) or simply fail with no report.

- type failure

  An argument to a built-in predicate is inappropriately instantiated (for example, the first argument of arg(a, *X*, *Y*)). This is a continuable error condition - logically, the goal should simply fail. With prolog_flag/3, you may specify whether to report the condition (as a warning) and then fail (the default behavior) or simply fail with no report.

- instantiation fault

  The arguments to a built-in predicate are insufficiently instantiated. It may be that an uninstantiated variable is passed where an input value is required (for example, the first argument to arg/3). There are also built-in predicates at least some subset of the arguments to which must be instantiated (for example, functor/3). Logically, there is nothing wrong with this. The Herbrand universe of a set of clauses is countable, so a Prolog implementation could in principle backtrack over all possible solutions for an ''underinstantiated'' goal. There are of course good reasons not to do this. Generally, the cost would be large and the benefit small. However, the choice not to backtrack is a choice of the implementation, so this condition is considered an implementation fault rather than a logical error. It is not continuable. The system reports it (as a fault) and then aborts.

- representation fault

  For example, an attempt is made to construct a term with an arity that exceeds the limit of the functor representation. See Appendix A for the limits of Aquarius Prolog. Such a condition is not continuable. The system reports it (as a fault) and then aborts.

- I/O error

  For example, an attempt is made to read past the end of a stream. Such a condition is not continuable with one exception: it is logically acceptable and convenient in some circumstances that predicates that open files simply fail, rather than report the condition (as an error) and then abort (the default behavior), when the specified file cannot be opened. You may select a behavior for such file errors with prolog_flag/3. The system always reports other I/O error conditions (as errors) and then aborts.

- system restriction violation

  For example, an attempt is made to assert a clause for a built-in predicate. Such a condition is not continuable. The system reports it (as an error) and then aborts.

- system limit violation

  For example, virtual memory is exhausted. This is an implementation fault so severe that it may be impossible to continue even to the extent of executing the usual error handler. The system reports it (as a disaster) and terminates execution (with exit code -1). This is done through an error handling mechanism dedicated to this sort of condition.

- s yntax error

  This can occur only during read operations. It is continuable - when a syntax error is encountered, it is reported (as a syntax error), and the read operation tries again. This is done through an error handling mechanism dedicated to this sort of condition.

Error messages generally have the form

```
[<error phrase>: <detail>:
 <culprit>]
```

except for system limit violation messages, which have the forms

```
[DISASTER: <detail>]

[DISASTER: <detail>:
 <culprit>]

[DISASTER: <detail>:
 <culprit>
 {advice}]
```

and syntax error messages, which have the forms

```
[SYNTAX ERROR: <detail>]

[SYNTAX ERROR: <detail>...
 <part of term>
 ...HERE:
 <rest of term>]
```

5-2. handle_error(*+ErrorNumber*, *+Blanks*, *+Goal*), error_data(*+ErrorNumber*, *-ErrorPhrase*, *-DetailTemplate*, *-Action*), unknown_predicate_condition(*+ErrorNumber*), type_failure_condition(*+ErrorNumber*), file_error_condition(*+ErrorNumber*)

The default error handler behaves as described above for each category of error condition. You may supersede this by defining handle_error/3. I f defined, handle_error/3 is called in place of the default error handler when an error condition is entered. *ErrorNumber* is a unique number associated with the error condition. *Blanks* contains whatever terms are necessary to fill in the blanks in the default error message associated with *ErrorNumber*; it is in a form suitable for use with format/[2,3] (section 10-1-8). *Goal* is the culprit goal. Ordinarily it is a user-level goal, but for a few built-in predicates such as call/1 (section 6-7) that operate recursively, it may be a subgoal of a user-level goal.

ErrorNumber may be expanded via error_data/4. *ErrorPhrase* is one of 'WARNING', 'FAULT' or 'ERROR'. *DetailTemplate* is a template for the detail of the default error message associated with *ErrorNumber*; it is in a form suitable for use with format/[2,3] (section 10-1-8). Action is one of abort, fail, halt or true. See Appendix B for an effective definition of error_data/4.

It is possible to determine whether *ErrorNumber* corresponds to an unknown predicate failure, a type failure or a file error via unknown_predicate_condition/1, type_failure_condition/1 or file_error_condition/1, respectively.

The default error handler actually behaves as if handle_error/3 were defined with four clauses:

```
handle_error(ErrorNumber, _, _) :-
        unknown_predicate_condition(ErrorNumber),
        prolog_flag(unknown_predicate, fail), !,
        fail.
handle_error(ErrorNumber, _, _) :-
        type_failure_condition(ErrorNumber),
        prolog_flag(type_failure, fail), !,
        fail.
handle_error(ErrorNumber, _, _) :-
        file_error_condition(ErrorNumber),
        prolog_flag(file_error, fail), !,
        fail.
handle_error(ErrorNumber, Blanks, Goal) :-
        error_data(ErrorNumber, ErrorPhrase, DetailTemplate, Action),
        current_output(Output),
        set_output(user_error),
        format("~N[~a: ", ErrorPhrase),
        format(DetailTemplate, Blanks),
        format(":~n ~q]~n", [Goal]),
        set_output(Output),
        call(Action).
```

If you wish to handle only a few error conditions in an unusual way, you should include these clauses or something like them to handle the rest in the usual way. Note that you may do anything you wish with an error condition, but you should be careful; if, for example, you try to continue an instantiation fault, you are asking for a disaster.

System limit violations and syntax errors are handled separately. There is no provision for superseding the default handlers for these error conditions.

6. Control

This chapter describes fundamental predicates and control structures.

6-1. *+P*, *+Q*

(*P*, *Q*), the conjunction of *P* and *Q*, succeeds if first *P* succeeds and then *Q* succeeds. This is not a built-in predicate, but it is like a built-in predicate in that call((*P*, *Q*)) executes the conjunction of *P* and *Q*.

6-2. *+P* ; *+Q*

(*P* ; *Q*), the disjunction of *P* and *Q*, succeeds if either first *P* succeeds or then *Q* succeeds. This is not a built-in predicate, but it is like a built-in predicate in that call((*P* ; *Q*)) executes the disjunction of *P* and *Q*.

6-3. !

!, the cut, succeeds immediately, but if backtracking returns to it, the parent goal (the goal that matched the head of the clause containing the cut) fails. This is not a built-in predicate, but it is like a built-in predicate in that goals containing cuts may be passed to call/1; the execution of such goals is described with call/1.

6-4. *+P* -> *+Q* ; *+R*

P –> *Q* ; *R* is the conditional goal read

if *P* then *Q* else *R*

First *P* is executed. If it succeeds, then *Q* is executed, and if *Q* fails, the conditional goal fails. If *P* fails, then *R* is executed, and if *R* fails, the conditional goal fails. Note that if *P* succeeds but *Q* fails, backtracking into *P* does not occur; –> /2 behaves like a cut whose range is restricted to the disjunction in which it occurs. Also note that *P* is allowed to contain cuts; Aquarius Prolog differs from Quintus Prolog in this respect. This is not a built-in predicate, but it is like a built-in predicate in that call((*P* –> *Q* ; *R*)) executes the conditional goal *P* –> *Q* ; *R*.

6-5. *+P* -> *+Q*

P –> *Q*, when other than the first argument of a disjunction, is the conditional goal read

if *P* then *Q* else fail

that is, it is equivalent to *P* –> *Q* ; fail. First *P* is executed, and if it succeeds, then *Q* is executed. Note that if *P* succeeds but *Q* fails, backtracking into *P* does not occur; –> /2 behaves like a cut whose range is restricted to the conditional goal. Also note that *P* is allowed to contain cuts; Aquarius Prolog differs from Quintus Prolog in this respect. This is not a built-in predicate, but it is like a built-in predicate in that call((*P* –> *Q*)) executes the conditional goal *P* –> *Q*.

6-6. \+ *+P*

\+ *P*, the negation-by-failure of *P*, fails if *P* succeeds and succeeds if *P* fails. It is equivalent to *P* –> fail ; true. Note that *P* is allowed to contain cuts; Aquarius Prolog differs from Quintus Prolog in this respect. This is not a built-in predicate, but it is like a built-in predicate in that call(\+ *P*) executes the negation-by-failure of *P*.

Note that with a prefix operator such as \+ /1, it is necessary to be careful about spaces if the argument is in parentheses. For example,

\+ (*P*, *Q*)

is the negation-by-failure of the conjunction of *P* and *Q*, but

\+(*P*, *Q*)

is a call to \+ /2 with arguments *P* and *Q*.

6-7. call(*+Goal*)

call(*Goal*) behaves as if

call(*Goal*) :– *Goal*.

Thus, for example, call((foo(*X*), !)) behaves as if

call((foo(*X*), !)) :– foo(*X*), !.

Goal may contain cuts, but those cuts cannot escape from call(*Goal*) to affect the parent goal; call(!) is equivalent to true.

If *Goal* is not instantiated, an instantiation fault occurs. If *Goal* is instantiated to something other than a term having the form of a legal goal, that is, an atom or compound term, a type failure occurs. If *Goal* is instantiated to a call to an unknown predicate, an unknown predicate failure occurs. Further error conditions may arise from execution of *Goal*. T he same criteria apply to each subgoal of *Goal*.

Note that error messages from call/1 mention only the goal from which the error condition arises, which may be a subgoal of the goal call/1 is passed at user-level. For example:

```
| ?- call((X,true)).
[FAULT: insufficient instantiation:
 call(_27)]

no
| ?-
```

6-8. true

true/0 always succeeds.

true/0 has no associated error conditions.

6-9. otherwise

otherwise/0 is identical to true/0. It is provided for laying out conditional goals in a readable way, as in

```
( Test1 ->
    Goal1
; Test2 ->
    Goal2
...
; TestN ->
    GoalN
; otherwise ->
    Goal
)
```

otherwise/0 has no associated error conditions.

6-10. fail

fail/0 always fails.

fail/0 has no associated error conditions.

6-11. false

false/0 is identical to fail/0.

false/0 has no associated error conditions.

6-12. repeat

repeat/0 succeeds immediately when called and whenever reentered by backtracking. It can be used to simulate the loop constructs provided by traditional imperative languages. The typical repeat loop has the form

```
repeat
   Goal1,
   Goal2,
   ...,
   GoalN,
   Test,
!
```

This executes *Goal1*, *Goal2*, ..., *GoalN* in sequence followed by the *Test*. If the *Test* succeeds, the cut terminates the loop. If the *Test* fails, backtracking into repeat/0 occurs, which succeeds again and initiates another iteration of *Goal1*, *Goal2*, ..., *GoalN*. Generally, each *Goal1*, *Goal2*, ..., *GoalN* should succeed determinately. It can be confusing if these sometimes fail, so that backtracking happens before the *Test* is reached, or if these are nondeterminate, so that backtracking does not always go straight to repeat/0. Note that the loop is only useful if at least one of *Goal1*, *Goal2*, ..., *GoalN* executes a side-effect, that is, a database modification or an I/O operation; otherwise, each iteration of the loop is identical, and it never terminates.

Repeat loops are seldom needed; usually recursive predicate calls are preferable. However, note that since storage allocated for execution of a goal is immediately recovered on backtracking through that goal, storage allocated for *Goal1*, *Goal2*, ..., *GoalN* of a repeat loop is very efficiently recovered. When an iterative construct is realized with recursive predicate calls, storage recovery can only take place through garbage collection (section 19-1).

repeat/0 has no associated error conditions.

6-13. break

break/0 interrupts the current execution and displays the prompt

```
[1]
| ?-
```

The interpreter is ready to accept input, as under the top-level. Another call to break/0 increments the break level to 2, and so on. The break level appears in square brackets on a separate line before the top-level prompt, as shown above.

When an end of file is encountered, the break level closes, break/0 succeeds and the suspended execution resumes. Alternatively, a call to abort/0 abandons the suspended execution.

break/0 has no associated error conditions.

6-14. abort

abort/0 abandons the current execution. If the current execution began with the top-level, abort/0 returns to the top-level; all break levels (suspended execution states created via break/0) a re destroyed. If the current execution did not begin with the top-level, abort/0 terminates execution (with exit code −1).

abort/0 is a drastic measure ordinarily used only when an error condition arises from which there is no reasonable way to go on.

abort/0 has no associated error conditions.

6-15. halt

halt/0 terminates execution (with exit code 0).

halt/0 has no associated error conditions.

7. Terms

This chapter describes fundamental predicates that inspect or alter the instantiation state of a term.

7-1. var(*?Term*)

var(*Term*) succeeds if and only if *Term* is an uninstantiated variable. For example:

```
| ?- var(X).

X = _23

| ?- var([X]).

no
| ?-
```

var/1 has no associated error conditions.

7-2. nonvar(*?Term*)

nonvar(*Term*) succeeds if and only if *Term* is instantiated, that is, not an uninstantiated variable. For example:

```
| ?- nonvar(X).

no
| ?- nonvar(foo(X)).

X = _39

| ?-
```

nonvar/1 has no associated error conditions.

7-3. atom(*?Term*)

atom(*Term*) succeeds if and only if *Term* is instantiated to an atom. For example:

```
| ?- atom(X).

no
| ?- atom('Aquarius Prolog').

yes
| ?- atom(['Aquarius Prolog']).

no
| ?-
```

atom/1 has no associated error conditions.

7-4. atomic(*?Term*)

atomic(*Term*) succeeds if and only if *Term* is instantiated to an atomic term. For example:

```
| ?- atomic(X).

no
| ?- atomic(5).

yes
| ?- atomic(3.14).

yes
| ?- atomic('Aquarius Prolog').

yes
| ?- atomic(foo(5)).

no
| ?-
```

atomic/1 has no associated error conditions.

7-5. compound(*?Term*)

compound(*Term*) succeeds if and only if *Term* is instantiated to a compound term. For example:

```
| ?- compound(X).

no
| ?- compound(foo(X)).

X = _39

| ?- compound([foo]).

yes
| ?- compound([]).

no
| ?-
```

Note that Quintus Prolog does not provide compound/1.

compound/1 has no associated error conditions.

7-6. cons(*?Term*)

cons(*Term*) succeeds if and only if *Term* is instantiated to a cons cell. For example:

```
| ?- cons(X).

no
| ?- cons([]).

no
| ?- cons([0]).

yes
| ?- cons([X|foo]).

X = _27

| ?- cons(foo([0])).

no
| ?-
```

Note that Quintus Prolog does not provide cons/1.

cons/1 has no associated error conditions.

7-7. denumerable(*?Term*)

denumerable(*Term*) succeeds if and only if *Term* is instantiated to an integer or atom. For example:

```
| ?- denumerable(X).

no
| ?- denumerable(5).

yes
| ?- denumerable(3.14).

no
| ?- denumerable('Aquarius Prolog').

yes
| ?- denumerable(foo(X)).

no
| ?-
```

Note that Quintus Prolog does not provide denumerable/1.

denumerable/1 has no associated error conditions.

7-8. float(*?Term*)

float(*Term*) succeeds if and only if *Term* is instantiated to a floating point number. F or example:

```
| ?- float(X).

no
| ?- float(3.14).

yes
| ?- float([3.14]).

no
| ?-
```

float/1 has no associated error conditions.

7-9. integer(*?Term*)

integer(*Term*) succeeds if and only if *Term* is instantiated to an integer. F or example:

```
| ?- integer(X).

no
| ?- integer(5).

yes
| ?- integer(foo(5)).

no
| ?-
```

integer/1 has no associated error conditions.

7-10. list(*?Term*)

list(*Term*) succeeds if and only if *Term* is instantiated to a cons cell or the empty list. For example:

```
| ?- list(X).

no
| ?- list([]).

yes
| ?- list([0]).

yes
| ?- list([X|foo]).

X = _29

| ?- list(foo([0])).

no
| ?-
```

Note that Quintus Prolog does not provide list/1.

list/1 has no associated error conditions.

7-11. negative(*?Term*)

negative(*Term*) succeeds if and only if *Term* is instantiated to a negative (strictly less than 0) integer. For example:

```
| ?- negative(X).

no
| ?- negative(-5).

yes
| ?- negative(0).

no
| ?- negative(-3.14).

no
| ?- negative(foo(5)).

no
| ?-
```

Note that Quintus Prolog does not provide negative/1.

negative/1 has no associated error conditions.

7-12. nil(*?Term*)

nil(*Term*) succeeds if and only if *Term* is instantiated to the empty list. For example:

```
| ?- nil(X).

no
| ?- nil([]).

yes
| ?- nil([X]).

no
| ?-
```

Note that Quintus Prolog does not provide nil/1.

nil/1 has no associated error conditions.

7-13. nonnegative(*?Term*)

nonnegative(*Term*) succeeds if and only if *Term* is instantiated to a nonnegative (greater than or equal to 0) integer. For example:

```
| ?- nonnegative(X).

no
| ?- nonnegative(-5).

no
| ?- nonnegative(0).

yes
| ?- nonnegative(3.14).

no
| ?- nonnegative(foo(5)).

no
| ?-
```

Note that Quintus Prolog does not provide nonnegative/1.

nonnegative/1 has no associated error conditions.

7-14. number(*?Term*)

number(*Term*) succeeds if and only if *Term* is instantiated to an integer or floating point number. For example:

```
| ?- number(X).

no
| ?- number(5).

yes
| ?- number(3.14).

yes
| ?- number([5]).

no
| ?-
```

number/1 has no associated error conditions.

7-15. simple(*?Term*)

simple(*Term*) succeeds if and only if *Term* is either an uninstantiated variable or instantiated to an atomic term. For example:

```
| ?- simple(X).

X = _29

| ?- simple(5).

yes
| ?- simple(3.14).

yes
| ?- simple('Aquarius Prolog').

yes
| ?- simple(foo(X)).

no
| ?-
```

Note that Quintus Prolog does not provide simple/1.

simple/1 has no associated error conditions.

7-16. structure(*?Term*)

structure(*Term*) succeeds if and only if *Term* is instantiated to a compound term other than a cons cell. For example:

```
| ?- structure(X).

no
| ?- structure(foo(X)).

X = _39

| ?- structure([foo]).

no
| ?- structure([]).

no
| ?-
```

Note that Quintus Prolog does not provide structure/1.

structure/1 has no associated error conditions.

7-17. is_list(*?Term*), is_partial_list(*?Term*), is_proper_list(*?Term*)

The following classification is as in Richard A. O'Keefe, ''Practical Prolog for Real Programmers,'' p age 7.

A proper list is either [] or [*H*|*T*] where *H* is any term and *T* is a proper list. [a,b,c] and [_,_] are proper lists, but [a|_] is not a proper list (because its tail is a variable).

A partial list is either a variable or [*H*|*T*] where *H* is any term and *T* is a partial list. [a,b,c] and [_,_] are not partial lists (because their tails are []), but [a|_] is a partial list.

A list is either a proper list or a partial list.

This classification suggests a trio of predicates that are useful enough that they have been built into Aquarius Prolog:

is_list(*Term*) succeeds if and only if *Term* is a list.

is_partial_list(*Term*) succeeds if and only if *Term* is a partial list.

is_proper_list(*Term*) succeeds if and only if *Term* is a proper list.

Note that Quintus Prolog has a library predicate called is_list/1 that behaves like is_proper_list/1.

None of these predicates has any associated error conditions.

7-18. *?Term1* = *?Term2*

Term1 = *Term2* unifies *Term1* and *Term2*. It fails if *Term1* and *Term2* fail to unify.

= /2 has no associated error conditions.

7-19. functor(*?Term*, *?Name*, *?Arity*)

If *Term* is instantiated, functor(*Term*, *Name*, *Arity*) unifies *Name* with the name and *Arity* with the arity of the principal functor of *Term*; if *Term* is not instantiated, functor(*Term*, *Name*, *Arity*) constructs a term whose principal functor has name *name* and arity *Arity* and unifies *Term* with it. Note that the name of the principal functor of an atomic term is the term itself, with arity 0. For example:

```
| ?- functor(foo(0, a), Name, Arity).

Name = foo,
Arity = 2

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

Term = foo(_85,_86)

| ?- functor(42, Name, Arity).

Name = 42,
Arity = 0

| ?- functor(Term, a, 0).

Term = a

| ?-
```

If both *Term* and *Name* or both *Term* and *Arity* are not instantiated, an instantiation fault occurs. If *Name* is instantiated to something other than an atomic term or if *Arity* is instantiated to something other than a nonnegative integer, a type failure occurs.

7-20. arg(*+Number*, *+Term*, *?Arg*)

arg(*Number*, *Term*, *Arg*) unifies *Arg* with the *Number*th argument of *Term*. *Number* must be instantiated to a positive integer, and *Term* must be instantiated to something other than a number. For example:

```
| ?- arg(1, foo(0), Arg).

Arg = 0

| ?- arg(1, foo(X), 0).

X = 0

| ?-
```

Note that arg(1, a, *X*) simply fails, but arg(1, 0, *X*) type fails. This is because a might come from a goal functor(*T*, a, 0), and it makes sense to consider it the ‘‘empty’’ case of a ‘‘vector’’ class. But no other kind of atomic term (for example, 0) has any related compound term.

If *Number* or *Term* is not instantiated, an instantiation fault occurs. If *Number* is instantiated to something other than a positive integer or if *Term* is instantiated to a number, a type failure occurs.

7-21. *?Term* =.. *?List*

Term =.. *List* unifies *List* with a list whose head is the name of the principal functor of *Term* and whose tail is a list of the arguments of *Term*. Note that the name of the principal functor of an atomic term is the term itself. If *Term* is not instantiated, *List* must be instantiated to a proper list whose head is an atomic term. For example:

```
| ?- foo(0, a) =.. List.

List = [foo,0,a]

| ?- Term =.. [0].

Term = 0
```

```
| ?- bar(X) =.. [Y, 0].

X = 0,
Y = bar

| ?-
```

=.. /2 is pronounced ‘‘univ’’ after a predicate of that name and similar function in the original Marseilles Prolog implementation.

Anything that can be done with =.. /2 can also be done with some combination of arg/3 and functor/3, and it is generally faster to do it this way, as arg/3 and functor/3 are highly optimized in Aquarius Prolog. =.. /2 is less optimized, partly because of the complicated argument checking it requires.

If *Term* is not instantiated and *List* is insufficiently instantiated, an instantiation fault occurs. If *List* is instantiated to something other than a nonempty list, if the head of *List* is instantiated to something other than an atomic term or if the tail of *List* is instantiated to something other than a list, a type failure occurs.

7-22. length(*?List*, *?Length*)

length(*List*, *Length*) unifies *List* with a list of length *Length*. If *List* is a partial list and *Length* is not instantiated, length(*List*, *Length*) is nondeterminate; on backtracking it generates longer and longer lists whose elements are variables. For example:

```
| ?- length([1,2,3], Length).

Length = 3

| ?- length(List, 3).

List = [_62,_64,_66]

| ?- length(List, Length).

List = [],
Length = 0 ;

List = [_82],
Length = 1 ;

List = [_82,_84],
Length = 2 ;

...
```

If *List* is instantiated to something other than a list or if *Length* is instantiated to something other than a nonnegative integer, a type failure occurs.

7-23. atom_chars(*?Atom*, *?Chars*)

If *Atom* is instantiated to an atom, atom_chars(*Atom*, *Chars*) unifies *Chars* with the name of *Atom* (a proper list of ASCII character codes). If *Atom* is not instantiated, it is unified with the atom whose name is *Chars*. For example:

```
| ?- atom_chars(atom, Chars).

Chars = [97,116,111,109]

| ?- atom_chars(Atom, [97,116,111,109]).

Atom = atom

| ?-
```

If *Atom* is not instantiated and *Chars* is insufficiently instantiated, an instantiation fault occurs. If *Chars* is instantiated to something other than a list of ASCII character codes, a type failure occurs.

7-24. number_chars(*?Number*, *?Chars*)

If *Number* is instantiated to an number, number_chars(*Number*, *Chars*) unifies *Chars* with the name of *Number* (a proper list of ASCII character codes). If *Number* is not instantiated, it is unified with the number whose name is *Chars*. For example:

```
| ?- number_chars(42, Chars).

Chars = [52,50]

| ?- number_chars(Number, [52,50]).

Number = 42

| ?- number_chars(3.14, Chars).

Chars = [51,46,49,52]
```

```
| ?- number_chars(Number, [51,46,49,52]).

Number = 3.14

| ?-
```

If *Number* is not instantiated and *Chars* is insufficiently instantiated, an instantiation fault occurs. If *Chars* is instantiated to something other than a list of ASCII character codes, a type failure occurs.

7-25. name(*?Constant*, *?Chars*)

If *Constant* is instantiated to an atom or number, name(*Constant*, *Chars*) unifies *Chars* with the name of *Constant* (a proper list of ASCII character codes). If *Constant* is not instantiated, it is unified with the atom or number whose name is *Chars*. If in this case *Chars* corresponds to a number, *Constant* is unified with that number, else *Constant* is unified with an atom. Thus, atoms such as '0' can be converted into character strings by name/2 but cannot be constructed by name/2, since

```
| ?- name('0', Chars).

Chars = [48]

| ?-
```

but

```
| ?- name(X, [48]), atom(X).

no
| ?- name(X, [48]), integer(X).

X = 0

| ?-
```

Use atom_chars/2 or number_chars/2 as appropriate to avoid this anomaly when the type of *Constant* is known in advance.

If *Constant* is not instantiated and *Chars* is insufficiently instantiated, an instantiation fault occurs. If *Chars* is instantiated to something other than a list of ASCII character codes, a type failure occurs.

7-26. numbervars(*?Term*, *+FirstVar*, *-LastVar*)

numbervars(*Term*, *FirstVar*, *LastVar*) instantiates each variable in *Term* to a term of the form '$VAR'(*N*), where *N* is an integer. *FirstVar* must be an integer. This integer is the value of *N* associated with the first variable in *Term*, beginning from the left. The second distinct variable in *Term* is instantiated to '$VAR'(*J*), where *J* is *FirstVar*+1, the third to '$VAR'(*K*), where *K* is *J*+1 and so on. *LastVar* is unified with the value of *L*+1, where the last distinct variable in *Term* is instantiated to '$VAR'(*L*). For example:

```
| ?- Term = foo([X, Y], X, Y), numbervars(Term, 0, LastVar), display(Term).
foo(.($VAR(0),.($VAR(1),[])),$VAR(0),$VAR(1))
Term = foo([A,B],A,B),
X = A,
Y = B,
LastVar = 2

| ?-
```

Note that in this example we use display/1 rather than write/1 to see the instantiations. write/1, as well as writeq/1, which the Aquarius Prolog top-level uses to write variable bindings, treats terms of the form '$VAR'(*N*) specially; it writes A if *N* is 0, B if *N* is 1, ..., Z if *N* is 25, A1 if *N* is 26, A2 if *N* is 27, ...

If *FirstVar* is not instantiated, an instantiation fault occurs. If *FirstVar* or *LastVar* is instantiated to something other than an integer, a type failure occurs.

7-27. copy_term(*+Term*, *-Copy*)

copy_term(*Term*, *Copy*) makes a copy of *Term* in which all variables have been replaced by variables that occur nowhere else and unifies *Copy* with it.

clause/[2,3] (section 13-4), recorded/3 (section 14-3) and instance/2 (section 12-4) return new copies of database items in precisely the same sense that copy_term/2 returns a new copy.

copy_term/2 has no associated error conditions.

8. Term Comparison

This chapter describes predicates that compare and sort terms according to a standard order.

8-1. The Standard Order

The standard order is a total ordering on Prolog terms as follows:

variables @< numbers @< atoms @< compound terms

Within each category of term, the standard order is as follows:

- Variables are put in temporal order, the older of two variables coming first. Note that this order is unrelated to variable names.
- Numbers are put in numerical order. For a number that may be represented by an integer or a floating point number, as in 1 and 1.0, the integer is regarded as infinitesimally less than the floating point number.
- Atoms are put in (ASCII) alphabetical order by name.
- Compound terms are ordered first by arity, then by principal functor name and finally by arguments from left to right. Lists are regarded as compound terms with principal functor '.'/2.

For example,

[*X*, 1, 1.0, a, foo(*X*), foo(a), [1], *X*=a]

is a list of terms in standard order.

8-2. *?Term1 == ?Term2*

Term1 == *Term2* succeeds if *Term1* and *Term2* are the same term. Note that

```
| ?- X == Y.

no
| ?- X = Y, X == Y.
```

```
X = _15,
Y = _15

| ?-
```

The first goal fails because *X* and *Y* are distinct uninstantiated variables. The second goal succeeds because the *X* = *Y* goal unifies the two variables.

== /2 has no associated error conditions.

8-3. *?Term1 \== ?Term2*

Term1 \== *Term2* succeeds if *Term1* and *Term2* are not the same term. Note that

```
| ?- X \== Y.

X = _15,
Y = _33

| ?- X = Y, X \== Y.

no
| ?-
```

The first goal succeeds because *X* and *Y* are distinct uninstantiated variables. The second goal fails because the *X* = *Y* goal unifies the two variables.

\== /2 has no associated error conditions.

8-4. *?Term1 @< ?Term2*

Term1 @< *Term2* succeeds if *Term1* precedes *Term2* in the standard order. F or example:

```
| ?- 0 @< atom.

yes
| ?- atom @< X.

no
| ?- foo(0) @< foo(0).

no
| ?-
```

@< /2 has no associated error conditions.

8-5. *?Term1 @>= ?Term2*

Term1 @>= *Term2* succeeds if *Term1* does not precede *Term2* in the standard order. For example:

```
| ?- 0 @>= atom.

no
| ?- atom @>= X.

X = _2f

| ?- foo(0) @>= foo(0).

yes
| ?-
```

@>= /2 has no associated error conditions.

8-6. *?Term1 @> ?Term2*

Term1 @> *Term2* succeeds if *Term1* succeeds *Term2* in the standard order. F or example:

```
| ?- 0 @> atom.

no
| ?- atom @> X.

X = _2d

| ?- foo(0) @> foo(0).

no
| ?-
```

@> /2 has no associated error conditions.

8-7. *?Term1* @=< *?Term2*

Term1 @=< *Term2* succeeds if *Term1* does not succeed *Term2* in the standard order. For example:

```
| ?- 0 @=< atom.

yes
| ?- atom @=< X.

no
| ?- foo(0) @=< foo(0).

yes
| ?-
```

@=< /2 has no associated error conditions.

8-8. compare(*?Relation*, *?Term1*, *?Term2*)

compare(*Relation*, *Term1*, *Term2*) unifies *Relation* with < if *Term1* precedes *Term2* in the standard order, > if *Term1* succeeds *Term2* in the standard order or = if *Term1* and *Term2* are the same term. For example:

```
| ?- compare(Relation, 0, atom).

Relation = <

| ?- compare(>, atom, X).

X = _47

| ?-
```

If *Relation* is instantiated to something other than <, > or =, a t ype failure occurs.

8-9. sort(*+List*, *-Sorted*)

sort(*List*, *Sorted*) sorts the elements of *List* into standard order, removing any duplicate occurrences of an element, and unifies the sorted list with *Sorted*. For example:

```
| ?- sort([foo(a),0,foo(X),X,a,[0],foo(X)], Sorted).

X = _61,
Sorted = [_61,0,a,foo(_61),foo(a),[0]]

| ?-
```

The algorithm is a merge sort. The time required to sort a list of length *N* is at worst order *N*∗log(*N*).

If *List* is a partial list, an instantiation fault occurs. If *List* is otherwise instantiated to something other than a proper list or if *Sorted* is instantiated to something other than a list, a type failure occurs.

8-10. msort(*+List*, *-Sorted*)

msort(*List*, *Sorted*) sorts the elements of *List* into standard order and unifies the sorted list with *Sorted*. It does not remove duplicate occurrences of an element. msort/2 is identical to sort/2 except that it does not remove duplicates. For example:

```
| ?- msort([foo(a),0,foo(X),X,a,[0],foo(X)], Sorted).

X = _65,
Sorted = [_65,0,a,foo(_65),foo(_65),foo(a),[0]]

| ?-
```

The algorithm is a merge sort. The time required to sort a list of length *N* is at worst order *N*∗log(*N*).

Note that Quintus Prolog does not provide msort/2.

If *List* is a partial list, an instantiation fault occurs. If *List* is otherwise instantiated to something other than a proper list or if *Sorted* is instantiated to something other than a list, a type failure occurs.

8-11. keysort(*+List*, *-Sorted*)

keysort(*List*, *Sorted*) sorts the elements of *List* into standard order by *Key*, where each element of *List* has the form *Key*−*Value*, and unifies the sorted list with *Sorted*. It scans *List* from left to right, and it does not remove duplicate occurrences of an element. For example:

```
| ?- keysort([0-foo(X),1-[0],0-a,2-X,0-foo(X),1-0,0-foo(a)], Sorted).

X = _57,
Sorted = [0-foo(_57),0-a,0-foo(_57),0-foo(a),1-[0],1-0,2-_57]

| ?-
```

Note that *Value* in each *Key*−*Value* element is ignored.

The algorithm is a merge sort. The time required to sort a list of length *N* is at worst order *N*∗log(*N*).

If *List* is a partial list, an instantiation fault occurs. If *List* is otherwise instantiated to something other than a proper list of *Key*−*Value* elements or if *Sorted* is instantiated to something other than a list, a type failure occurs.

9. Arithmetic

This chapter describes predicates that take arithmetic expressions as arguments and evaluate them to integers or floating point numbers (floats).

9-1. Arithmetic Expressions

Arithmetic evaluation and comparison predicates (is/2, < /2, =< /2, > /2, >= /2, =:= /2, =\= /2) take arithmetic expressions as arguments. Arithmetic expressions are terms built from numbers and variables combined into compound terms that represent arithmetic operations. Such expressions can be evaluated to yield numerical results that may be integers or floats. The type is determined by rules specified below.

At evaluation time, an arithmetic expression must be a ground term, that is, each variable in the expression must be instantiated, or an instantiation fault occurs. Each such variable must be instantiated to a number, or a type failure occurs. Note in particular that such a variable may not be instantiated to another arithmetic expression. There is one exception to this type requirement. If the expression is an argument of a goal passed to call/1, it may contain variables instantiated to other arithmetic expressions. For example:

```
| ?- X = 2+1, Y is 3+X.
[WARNING: 2+1 is not a number:
 '$ add'(3,2+1,_10)]

no
| ?- X = 2+1, call(Y is 3+X).

X = 2+1,
Y = 6

| ?-
```

The first goal elicits an error message that mentions a goal '$ add'(3,2+1,_0). This is an internally-generated subgoal that attempts to carry out part of the user-level goal *Y* is 3+*X*. Such subgoals are generated at load or compile time in an effort to maximize the speed of arithmetic operations and, in the Aquarius Prolog compiler, to promote determinacy detection. The speed of an arithmetic operation is usually considerably greater when the operation is handled ‘‘in-line’’ rather than by an explicit call to an evaluation or comparison predicate. Goals passed to call/1 are not subjected to subgoal generation. Subgoal generation for arithmetic expressions in code loaded using consult/1 and company can be disabled by setting the arithmetic_expansion flag to off (section 11-6).

In the rest of this section, *X* and *Y* represent general arithmetic expressions. The following compound terms are allowed in arithmetic expressions:

–*X*	evaluates to the negative of *X*. The type of the result is the type of *X*.
X + *Y*	evaluates to the sum of *X* and *Y*. If both operands are integers, the result is an integer; otherwise, it is a float.
X – *Y*	evaluates to the difference of *X* and *Y*. If both operands are integers, the result is an integer; otherwise, it is a float.
X * *Y*	evaluates to the product of *X* and *Y*. If both operands are integers, the result is an integer; otherwise, it is a float.
X / *Y*	evaluates to the quotient of *X* and *Y*. The result is always a float, regardless of the types of *X* and *Y*.
X // *Y*	evaluates to the integer quotient of *X* and *Y*. *X* and *Y* must both be integers. The result is the quotient of *X* and *Y* truncated to the nearest integer between it and 0.
X mod *Y*	evaluates to the remainder after the integer division of *X* and *Y*. *X* and *Y* must both be integers. The result has the same sign as *X*.
integer(*X*)	evaluates to *X* if *X* is an integer or to the nearest integer between *X* and 0 if *X* is a float.
float(*X*)	evaluates to *X* if *X* is a float or to the float equivalent of *X* is *X* is an integer.

The following bit-vector operations are allowed only with integer operands:

\(*X*)	evaluates to the bit-wise complement of *X*.
X /\ *Y*	evaluates to the bit-wise conjunction of *X* and *Y*.
X \/ *Y*	evaluates to the bit-wise disjunction of *X* and *Y*.
X << *Y*	evaluates to *X* shifted left *Y* places.
X >> *Y*	evaluates to *X* shifted right *Y* places with sign extension.

The following operation is included for compatibility with earlier Prolog implementations:

[*X*]	evaluates to *X*. The type of the result is the type of *X*. This is for doing arithmetic with character codes, as in

Letter is *N* – 10 + "A"

which is equivalent to

Letter is *N* – 10 + [65].

A cleaner way to do this is

Letter is *N* – 10 + 0'A.

9-2. *-X* is *+Expression*

X is *Expression* evaluates *Expression* as an arithmetic expression and unifies the result with *X*.

If *Expression* is not a ground term, an instantiation fault occurs. If *Expression* is instantiated to something other than an expression that evaluates to a number, a type failure occurs. Note that *Expression* may not contain variables that are instantiated to arithmetic expressions other than numbers.

9-3. *+X* =:= *+Y*

X =:= *Y* evaluates *X* and *Y* as arithmetic expressions and succeeds if the results are equal. If one of *X* or *Y* evaluates to an integer and the other to a float, the integer is converted to a float before the comparison is made.

If *X* or *Y* is not a ground term, an instantiation fault occurs. If *X* or *Y* is instantiated to something other than an expression that evaluates to a number, a t ype failure occurs. Note that *X* and *Y* may not contain variables that are instantiated to arithmetic expressions other than numbers.

9-4. *+X* =\= *+Y*

X =\= *Y* evaluates *X* and *Y* as arithmetic expressions and succeeds if the results are not equal. If one of *X* or *Y* evaluates to an integer and the other to a float, the integer is converted to a float before the comparison is made.

If *X* or *Y* is not a ground term, an instantiation fault occurs. If *X* or *Y* is instantiated to something other than an expression that evaluates to a number, a t ype failure occurs. Note that *X* and *Y* may not contain variables that are instantiated to arithmetic expressions other than numbers.

9-5. *+X < +Y*

X < Y evaluates *X* and *Y* as arithmetic expressions and succeeds if the result of evaluating *X* is strictly less than the result of evaluating *Y*. If one of *X* or *Y* evaluates to an integer and the other to a float, the integer is converted to a float before the comparison is made.

If *X* or *Y* is not a ground term, an instantiation fault occurs. If *X* or *Y* is instantiated to something other than an expression that evaluates to a number, a type failure occurs. Note that *X* and *Y* may not contain variables that are instantiated to arithmetic expressions other than numbers.

9-6. *+X >= +Y*

X >= Y evaluates *X* and *Y* as arithmetic expressions and succeeds if the result of evaluating *X* is greater than or equal to the result of evaluating *Y*. If one of *X* or *Y* evaluates to an integer and the other to a float, the integer is converted to a float before the comparison is made.

If *X* or *Y* is not a ground term, an instantiation fault occurs. If *X* or *Y* is instantiated to something other than an expression that evaluates to a number, a type failure occurs. Note that *X* and *Y* may not contain variables that are instantiated to arithmetic expressions other than numbers.

9-7. *+X > +Y*

X > Y evaluates *X* and *Y* as arithmetic expressions and succeeds if the result of evaluating *X* is strictly greater than the result of evaluating *Y*. If one of *X* or *Y* evaluates to an integer and the other to a float, the integer is converted to a float before the comparison is made.

If *X* or *Y* is not a ground term, an instantiation fault occurs. If *X* or *Y* is instantiated to something other than an expression that evaluates to a number, a type failure occurs. Note that *X* and *Y* may not contain variables that are instantiated to arithmetic expressions other than numbers.

9-8. *+X =< +Y*

X =< *Y* evaluates *X* and *Y* as arithmetic expressions and succeeds if the result of evaluating *X* is less than or equal to the result of evaluating *Y*. If one of *X* or *Y* evaluates to an integer and the other to a float, the integer is converted to a float before the comparison is made.

If *X* or *Y* is not a ground term, an instantiation fault occurs. If *X* or *Y* is instantiated to something other than an expression that evaluates to a number, a type failure occurs. Note that *X* and *Y* may not contain variables that are instantiated to arithmetic expressions other than numbers.

10. Input and Output

This chapter describes predicates that perform input and output. These are of two types: those that handle general terms and those that handle individual characters.

Input and output occur with respect to streams associated with files and terminals. At any time there is a current input stream and a current output stream. Initially, these are the standard input stream and the standard output stream, which are usually connected to the user's terminal but may be reset in order to perform input or output elsewhere. In relation to streams, there are three types of input and output predicates: those that use the current input or output stream, those that use the standard input or output stream and those that take an argument to specify explicitly which stream to use.

The Aquarius Prolog top-level refers to the standard input and output streams, not the current input and output streams, if these are different.

10-1. Input and Output of Terms

10-1-1. read(-*Term*)

read(*Term*) reads the next term from the current input stream and unifies it with *Term*. The term is read with respect to current operator definitions, and character escape sequences are recognized in quoted atoms, strings and character codes when the character_escapes flag is on (section 11-6). The term must be followed by a full-stop (a period followed by a layout character such as newline), which is consumed from the stream but is not part of the term.

read/1 continues to seek characters from the current input stream until a full-stop or an end of file is encountered. When an end of file is encountered, *Term* is unified with the atom end_of_file. O nce this occurs, it is an error to attempt to read from the stream again.

When a syntax error is encountered, a message is written and read/1 tries again, starting immediately after the full-stop that terminated the erroneous input. That is, read/1 perseveres until it manages to read a term.

If an attempt is made to read past the end of a stream, an I/O error occurs.

10-1-2. write(*?Term*)

write(*Term*) writes *Term* to the current output stream. *Term* is written with respect to current operator definitions. '.'(*H*,*T*) is written [*H*|*T*], {}(*X*) is written {*X*} and '$VAR'(*N*) is written A if *N* is 0, B if *N* is 1, ..., Z if *N* is 25, A1 if *N* is 26, A2 if *N* is 27, ... Quotes are not put around atoms that can't be read in by read/[1,2] without them, and character escape sequences are never produced in such atoms. For example:

```
| ?- write([X+5, {'Aquarius Prolog'}, '$VAR'(0)]).
[_31+5,{Aquarius Prolog},A]
X = _31

| ?-
```

Use writeq/[1,2] (section 10-1-3) or write_canonical/[1,2] (section 10-1-4) to be sure that atoms can be read in by read/[1,2].

write/1 has no associated error conditions.

10-1-3. writeq(*?Term*)

writeq(*Term*) writes *Term* to the current output stream. *Term* is written with respect to current operator definitions. '.'(*H*,*T*) is written [*H*|*T*], {}(*X*) is written {*X*} and '$VAR'(*N*) is written A if *N* is 0, B if *N* is 1, ..., Z if *N* is 25, A1 if *N* is 26, A2 if *N* is 27, ... Quotes are put around atoms that can't be read in by read/[1,2] without them, and character escape sequences are produced in such atoms when the character_escapes flag is on (section 11-6); these are the only respects in which writeq/1 differs from write/1. For example:

```
| ?- writeq([X+5, {'Aquarius Prolog'}, '$VAR'(0)]).
[_33+5,{'Aquarius Prolog'},A]
X = _33

| ?-
```

writeq/1 has no associated error conditions.

10-1-4. write_canonical(*?Term*)

write_canonical(*Term*) writes *Term* to the current output stream in canonical syntax. Operator expressions are never produced, and '.'(*H*,*T*), {}(*X*) and '$VAR'(*N*) are not treated specially. Quotes are put around atoms that can't be read in by read/[1,2] without them, and character escape sequences are produced in such atoms when the character_escapes flag is on (section 11-6). For example:

```
| ?- write_canonical([X+5, {'Aquarius Prolog'}, '$VAR'(0)]).
'.'(+(_47,5),'.'({}('Aquarius Prolog'),'.'('$VAR'(0),[])))
X = _47

| ?-
```

write_canonical/1 is provided so that a term can be written and read in by read/[1,2] regardless of special characters it may contain and differing operator definitions between write time and read time.

write_canonical/1 has no associated error conditions.

10-1-5. print(*?Term*)

print(*Term*) writes *Term* to the current output stream. By default, print/1 has the same effect as write/1, but you may change this by defining the predicate print_hook/1.

If *Term* is a variable, it is written as by write/1. Otherwise, the predicate print_hook/1 is called (if defined). If this succeeds, it is assumed that *Term* has been printed, and print/1 succeeds. If this fails and if *Term* is an atomic term, it is written as by write/1, or if *Term* is a compound term, the principal functor of *Term* is written as by write/1 and print_hook/1 is called (if defined) on the arguments of *Term*. Lists receive special treatment. If print/1 is given a list, print_hook/1 is called as usual. If this fails, the list is written as by write/1 in bracket notation, and print_hook/1 is called on each element. Since [*X*,*Y*,...,*Z*] is a different notation for .(*X*,[*Y*,...,*Z*]), one might expect print_hook/1 to be given *X* followed by [*Y*,...,*Z*], but this does not happen; print_hook/1 is given each of *X*, *Y*, ... and *Z* in succession.

For compatibility with Quintus Prolog and other older Prolog implementations, portray/1 is accepted in place of print_hook/1; if print_hook/1 is not defined but portray/1 is defined, portray/1 is called instead of print_hook/1. However, if both are defined, print_hook/1 is called, and portray/1 is ignored.

print/1 has no associated error conditions (except for any that may arise from execution of print_hook/1 or portray/1).

10-1-6. display(*?Term*)

display(*Term*) writes *Term* to the standard output stream, which may or may not be the current output stream. Operator expressions are never produced, and '.'(*H*,*T*), {}(*X*) and '$VAR'(*N*) are not treated specially. Quotes are not put around atoms that can't be read in by read/[1,2] without them, and character escape sequences are never produced in such atoms. For example:

```
| ?- display([X+5, {'Aquarius Prolog'}, '$VAR'(0)]).
.(+(_29,5),.({}(Aquarius Prolog),.($VAR(0),[])))
X = _29
```

```
| ?-
```

display/1 has no associated error conditions.

10-1-7. portray_clause(*?Clause*)

portray_clause(*Clause*) writes clause *Clause* to the current output stream in a format that has been carefully designed to be clear followed by a terminating full-stop. This is the operation used by listing/[0,1] (sections 11-4 and 11-5). If you want to write a clause to be read back by consult/1 and company (section 3-1) or the Aquarius Prolog compiler, u se this operation. For example:

```
| ?- consult(user).
| :- dynamic foo/1.
| foo(X) :- ( nonnegative(X) -> write(nonnegative) ; write(negative) ), nl.
|

yes
| ?- clause(foo(X), Body), portray_clause((foo(X):-Body)).
foo(A) :-
        ( nonnegative(A) ->
            write(nonnegative)
        ; write(negative)
        ),
        nl.

X = _39,
Body = (nonnegative(_39)->write(nonnegative);write(negative)),nl

| ?-
```

Note that portray_clause((foo(*X*):–*Body*)) does not bind *X*.

portray_clause/1 has no associated error conditions.

10-1-8. format(*+Control*, *+Args*)

format(*Control*, *Args*) writes *Args* to the current output stream according to *Control*. *Control* must be an atom or a string, which may contain character sequences of the form

˜*<n><c>*

where *<c>* is a control option character and *<n>* is its optional nonnegative integer argument. Characters that are not part of such a control option are put directly to the current output stream. *Args* is usually a list of arguments that are interpreted and possibly written by control options. But if there is only one argument, it need not be enclosed in a list. Note, however, that a string is itself a list, so it must be enclosed in a list even if it is the only argument. If *<n>* is allowed, it may be the character *, which indicates that the next argument in *Args* takes the place of *<n>*.

The following control options write the next argument in *Args* to the current output stream. If the argument is not instantiated to a term of the appropriate type, a type failure occurs.

~a argument is an atom, which is written without quotes.

```
| ?- format('~a', 'Aquarius Prolog').
Aquarius Prolog
yes
| ?-
```

~*<n>*c argument is an ASCII character code, which is written *<n>* times. If *<n>* is not specified, it defaults to 1.

```
| ?- format('~c', 0'*).
*
yes
| ?-
```

~*<n>*e argument is a float, which is written in exponential format with *<n>* digits to the right of the decimal point if *<n>* > 0. If *<n>* = 0, no decimal point is written. If *<n>* is not specified, it defaults to 6. Argument may also be an integer, in which case it is coerced to a float.

```
| ?- format('~e', 3.14).
3.140000e+00
yes
| ?-
```

~*<n>*E identical to ~*<n>*e except E is used instead of e.

```
| ?- format('~E', 3.14).
3.140000E+00
yes
| ?-
```

~*<n>*f argument is a float, which is written in nonexponential format with *<n>* digits to the right of the decimal point if *<n>* > 0. If *<n>* = 0, no decimal point is written. If *<n>* is not specified, it defaults to 6. Argument may also be an integer, in which case it is coerced to a float.

```
| ?- format('~2f', 123.456).
123.45
yes
| ?-
```

~<n>d argument is an integer, which is written as a signed decimal number shifted right <n> places. If <n> is not specified, it defaults to 0.

```
| ?- format('~d', 123).
123
yes
| ?- format('~2d', 123).
1.23
yes
| ?-
```

~<n>D identical to ~<n>d except commas are inserted to separate digits to the left of the decimal point into groups of three.

```
| ?- format('~2D', 123456).
1,234.56
yes
| ?-
```

~<n>r argument is an integer, which is written in radix <n>, where 2 =< <n> =< 36, using digits 0-9 and letters a-z. If <n> is not specified, it defaults to 8.

```
| ?- format('~r', 15).
17
yes
| ?- format('~2r', 15).
1111
yes

| ?- format('~16r', 15).
f
yes
| ?-
```

~<n>R identical to ~<n>r except A-Z are used instead of a-z.

```
| ?- format('~16R', 15).
F
yes
| ?-
```

~<n>s argument is a string, that is, a proper list of ASCII character codes, the first <n> of which are written. If <n> is not specified or is greater than the length of the string, it defaults to the length of the string.

```
| ?- format('~s', ["Prolog"]).
Prolog
yes
```

```
| ?- format('~3s', ["Prolog"]).
Pro
yes
| ?- format('~s', "Prolog").
[WARNING: 80 is not a proper list of ASCII character codes:
 format('~s',[80,114,111,108,111,103])]

no
| ?-
```

The following control options take an argument of any type.

~i argument is ignored.

```
| ?- format('~i', foo).

yes
| ?-
```

~k argument is passed to write_canonical/1.

~p argument is passed to print/1.

~q argument is passed to writeq/1.

~w argument is passed to write/1.

The following control options do not interpret an argument.

~~ writes one ~.

```
| ?- format('~~', []).
~
yes
| ?-
```

~<*n*>n writes <*n*> newline characters. If <*n*> is not specified, it defaults to 1.

```
| ?- format('a~nz', []).
a
z
yes
| ?-
```

~N writes a newline character if current output stream is not already at the beginning of a line.

```
| ?- write(foo),
|     format('~Na~n~Nz', []).
```

```
foo
a
z
yes
| ?-.
```

Note that Quintus Prolog also provides the ˜<*n*>g, ˜<*n*>G, ˜<*n*>|, ˜<*n*>+ and ˜<*n*>t control options. Aquarius Prolog does not provide these at present. They may or may not be implemented in the future.

Besides the argument type requirements described above, if *Control* is insufficiently instantiated, an instantiation fault occurs. If *Control* is instantiated to something other than an atom or a string, if *Control* contains a character sequence starting with the character ˜ that is not a legal control option or if *Args* does not correspond to *Control* appropriately (there must be exactly one argument for each control option that requires one), a type failure occurs.

10-2. Input and Output of Characters

10-2-1. get0(-*Char*)

get0(*Char*) unifies *Char* with (the ASCII code of) the next character from the current input stream. When an end of file is encountered, *Char* is unified with −1. Once the end of a stream is reached, it is an error to attempt to get from the stream again.

If *Char* is instantiated to something other than an ASCII character code, a type failure occurs. If an attempt is made to read past the end of a stream, an I/O error occurs.

10-2-2. get(-*Char*)

get(*Char*) unifies *Char* with (the ASCII code of) the next nonlayout character from the current input stream. Nonlayout characters are those whose codes are in the inclusive range from 33 (0'!) to 126 (0'˜). When an end of file is encountered, *Char* is unified with −1. Once the end of a stream is reached, it is an error to attempt to get from the stream again.

If *Char* is instantiated to something other than an ASCII character code, a type failure occurs. If an attempt is made to read past the end of a stream, an I/O error occurs.

10-2-3. skip(+*Number*)

skip(*Number*) skips over characters from the current input stream up to and including the first occurrence of the character specified by *Number*. *Number* should be an expression that evaluates to an ASCII character code, that is, an integer between 0 and 127. If this character does not occur in the current input stream, it is considered an attempt to read past the end of the stream.

skip/1 cannot skip to the end of the current input stream. To do this, use the goal

```
| ?- repeat, get0(-1), !.
```

If *Number* is not a ground term, an instantiation fault occurs. If *Number* is instantiated to something other than an expression that evaluates to an ASCII character code, a type failure occurs. If an attempt is made to read past the end of a stream, an I/O error occurs.

10-2-4. put(+*Number*)

put(*Number*) writes a character specified by *Number* to the current output stream. *Number* should be an expression that evaluates to an ASCII character code, that is, an integer between 0 and 127.

If *Number* is not a ground term, an instantiation fault occurs. If *Number* is instantiated to something other than an expression that evaluates to an ASCII character code, a type failure occurs.

10-2-5. nl

nl writes a newline character to the current output stream, as by the goal

```
| ?- put(10). % ASCII code for newline character is 10.
```

nl/0 has no associated error conditions.

10-2-6. tab(+*Number*)

tab(*Number*) writes a number of spaces specified by *Number* to the current output stream. *Number* should be an expression that evaluates to an integer. A call to tab/1 with an integer less than or equal to 0 succeeds with no effect.

If *Number* is not a ground term, an instantiation fault occurs. If *Number* is instantiated to something other than an expression that evaluates to an integer, a type failure occurs.

10-3. prompt(*-Old*, *+New*)

prompt(*Old*, *New*) unifies *Old* with the current value of the read prompt and sets the value of the read prompt to *New*. The read prompt is the atom written without quotes when the standard input stream is read by read/[1,2], get0/[1,2], ttyget0/1, get/[1,2], ttyget/1, skip/[1,2] or ttyskip/1. The default value is '|: '. Note that

```
| ?- prompt(Prompt, Prompt).
```

simply returns *Prompt* without changing the read prompt.

If *New* is a variable distinct from *Old*, an instantiation fault occurs. If *New* is instantiated to something other than an atom, a type failure occurs.

10-4. Streams and Files

Each input or output predicate refers implicitly or explicitly to a stream. A stream is connected to a file or terminal and is used for input or for output, but not for both.

As in Quintus Prolog, Aquarius Prolog streams are of two types, those opened by open/3 and those opened by see/1 or tell/1. A stream of the former type is referred to by a stream object, and a stream of the latter type is referred to by a file name.

A stream object is a term that identifies an input or output stream. Every stream is associated with a unique stream object, assigned when the stream is opened and not reassigned when the stream is closed, that is, unique not only at any given time but over the whole duration of execution. Any attempt to use the stream object of a stream that has been closed is an error, with one exception: an attempt to close the stream again succeeds with no effect. Note that a stream object is associated even with a stream opened by see/1 or tell/1, though access to such a stream ordinarily occurs through a file name rather than a stream object.

In Aquarius Prolog, a stream object is in general an ordinary Prolog structure of the form '$ stream'(*N*), where *N* is a nonnegative integer. Note that it is unwise to write programs that rely on this, since it is by no means standard. In addition, the standard streams provided by the operating system are identified by the ordinary Prolog atoms user_input, user_output and user_error. The standard streams, which are usually connected to the user's terminal, are always open; an attempt to close a standard stream succeeds with no effect.

A file name is one of

FileName	where *FileName* is an atom that is an absolute or relative file name;

library(*LibraryName*)	where *LibraryName* is an atom that is a relative file name. This file will be sought in the paths specified by the user-definable predicate library_directory/1 (section 3-2);
user	where user is considered the absolute file name of a pseudofile with which the standard input and output streams are permanently associated.

A file name may refer to a terminal (as a character-type special file, say). Files and terminals may be open for input and output simultaneously (though this may produce strange results with files). The pseudofile user is exclusively and permanently associated with the standard input and output streams. Opening user succeeds only with these standard streams; by contrast, any number of streams may be connected to any other file or terminal (though this may produce strange results). Closing user succeeds with no effect. Note that user is considered to be opened initially by see/1 and tell/1, so that, for example, seeing(*X*) succeeds with *X* = user, not *X* = user_input, and telling(*X*) succeeds with *X* = user, not *X* = user_output.

Local state information is associated with streams. For any stream, the number of characters read or written, the number of lines read or written and the number of characters read or written on the current line may be queried via character_count/2, line_count/2 and line_position/2 (sections 10-4-10, 10-4-11 and 10-4-12), respectively. Separate local state information is associated with each stream, except that all streams associated with the same terminal share local state information. This is convenient because input from terminals is echoed to terminals, so it is desirable for input streams to share local state information with output streams. Unlike Quintus Prolog, however, Aquarius Prolog distinguishes one terminal from another; it does not lump together all streams connected to any terminal whatever.

10-4-1. open(*+FileName*, *+Mode*, *-Stream*)

open(*FileName*, *Mode*, *Stream*) tries to open the file whose name is *FileName* in mode *Mode*. *Mode* must be one of read, write or append. read mode is for input. For read mode, the file must exist and be readable. write mode and append mode are for output. write mode creates a new file; if the file already exists, it is removed and recreated. append mode appends to the end of an existing file or creates the file if it does not already exist. For write mode and append mode, the file must be writable. If the file can be opened, a stream object associated with the file is returned in *Stream*.

open/3 recognizes the pseudofile user and associates it with the standard input and output streams; Aquarius Prolog differs from Quintus Prolog in this respect (Quintus Prolog open/3 considers user the name of a file in the current working directory).

If *FileName* or *Mode* is not instantiated, an instantiation fault occurs. If *FileName* is instantiated to something other than a legal file name, if *Mode* is instantiated to something other than read, write or append or if *Stream* is instantiated to something other than a legal stream object, a type failure occurs. If the file specified by *FileName* cannot be opened for any reason, then if the file_error flag is set to error (section 11-6), an I/O error occurs, or if the file_error flag is set to fail, a simple failure occurs.

10-4-2. open_null_stream(-*Stream*)

open_null_stream(*Stream*) creates an output stream that is not associated with any file and returns a stream object in *Stream*. Characters and terms sent to this stream are thrown away. This can be useful because local state information is kept for such streams.

If *Stream* is instantiated to something other than a legal stream object, a type failure occurs.

10-4-3. set_input(+*Stream*)

set_input(*Stream*) makes *Stream* the current input stream. Subsequent input goals (read/1 or get0/1 goals, for example) use this stream.

If *Stream* is not instantiated, an instantiation fault occurs. If *Stream* is instantiated to something other than a legal stream object, a type failure occurs. If *Stream* is instantiated to a stream object that does not correspond to an open input stream, an I/O error occurs.

10-4-4. set_output(+*Stream*)

set_output(*Stream*) makes *Stream* the current output stream. Subsequent output goals (write/1 or put/1 goals, for example) use this stream.

If *Stream* is not instantiated, an instantiation fault occurs. If *Stream* is instantiated to something other than a legal stream object, a type failure occurs. If *Stream* is instantiated to a stream object that does not correspond to an open output stream, an I/O error occurs.

10-4-5. current_input(*-Stream*)

current_input(*Stream*) unifies *Stream* with the current input stream.

If *Stream* is instantiated to something other than a legal stream object, a type failure occurs.

10-4-6. current_output(*-Stream*)

current_output(*Stream*) unifies *Stream* with the current output stream.

If *Stream* is instantiated to something other than a legal stream object, a type failure occurs.

10-4-7. flush_output(*+Stream*)

flush_output(*Stream*) flushes the buffer associated with output stream *Stream*.

Output to a stream is not necessarily sent to the file or terminal at once. Instead, it is buffered. This predicate flushes the buffer to assure that whatever has been written to the stream is actually sent to the file or terminal.

If *Stream* is not instantiated, an instantiation fault occurs. If *Stream* is instantiated to something other than a legal stream object, a type failure occurs. If *Stream* is instantiated to a stream object that does not correspond to an open output stream, an I/O error occurs.

10-4-8. close(*+FileNameOrStream*)

close(*FileNameOrStream*) closes the stream specified by *FileNameOrStream*. If *FileNameOrStream* is a stream object, then if an open stream is associated with it, the stream is closed, or else the close/1 goal succeeds with no effect. If *FileNameOrStream* is a file name, then if a stream is associated with it that was opened by see/1 or tell/1, the stream is closed, or else the close/1 goal type fails. If the stream to be closed is the current input or output stream, the current input or output stream is reset to user_input or user_output, respectively.

close/1 recognizes the pseudofile user and the standard stream objects user_input, user_output and user_error. A call to close/1 with any of these succeeds with no effect.

If *FileNameOrStream* is not instantiated, an instantiation fault occurs. If *FileNameOrStream* is instantiated to something other than a legal stream object or file name or if it is instantiated to a file name that is not associated with a stream opened by see/1 or tell/1, a type failure occurs.

10-4-9. current_stream(*?AbsoluteFileName*, *?Mode*, *?Stream*)

current_stream(*AbsoluteFileName*, *Mode*, *Stream*) succeeds when the file whose absolute name is *AbsoluteFileName* is open in mode *Mode* with stream object *Stream*. *Mode* is one of read, write or append. If *Stream* and either *Mode* or *AbsoluteFileName* are not instantiated, backtracking generates every possible solution.

Note that a nondeterminate current_stream/3 goal generates solutions in arbitrary order and sees changes to the stream ‘‘database’’ immediately.

current_stream/3 recognizes the pseudofile user and the standard stream objects user_input, user_output and user_error; Aquarius Prolog differs from Quintus Prolog in this respect.

If *AbsoluteFileName* is instantiated to something other than an atom, if *Mode* is instantiated to something other than read, write or append or if *Stream* is instantiated to something other than a legal stream object, a type failure occurs.

10-4-10. character_count(*+Stream*, *-CharacterCount*)

character_count(*Stream*, *CharacterCount*) unifies *CharacterCount* with the number of characters read or written on *Stream*. A freshly opened stream has a character count of 0.

If *Stream* is not instantiated, an instantiation fault occurs. If *Stream* is instantiated to something other than a legal stream object or if *CharacterCount* is instantiated to something other than a nonnegative integer, a type failure occurs. If *Stream* is instantiated to a stream object that does not correspond to an open stream, an I/O error occurs.

10-4-11. line_count(*+Stream*, *-LineCount*)

line_count(*Stream*, *LineCount*) unifies *LineCount* with the number of lines read or written on *Stream*. A freshly opened stream has a line count of 1.

If *Stream* is not instantiated, an instantiation fault occurs. If *Stream* is instantiated to something other than a legal stream object or if *LineCount* is instantiated to something other than a nonnegative integer, a type failure occurs. If *Stream* is instantiated to a stream object that does not correspond to an open stream, an I/O error occurs.

10-4-12. line_position(*+Stream*, *-LinePosition*)

line_position(*Stream*, *LinePosition*) unifies *LinePosition* with the number of characters read or written on the current line of *Stream*. A freshly opened stream has a line position of 0.

If *Stream* is not instantiated, an instantiation fault occurs. If *Stream* is instantiated to something other than a legal stream object or if *LinePosition* is instantiated to something other than a nonnegative integer, a type failure occurs. If *Stream* is instantiated to a stream object that does not correspond to an open stream, an I/O error occurs.

10-4-13. absolute_file_name(*+FileName*, *-AbsoluteFileName*)

absolute_file_name(*FileName*, *AbsoluteFileName*) unifies the absolute name of the file whose name is *FileName* with *AbsoluteFileName*.

absolute_file_name/2 recognizes the pseudofile user. When *FileName* is user, *AbsoluteFileName* is unified with user even if there is a file user in the current working directory.

Aquarius Prolog absolute_file_name/2 does not associate files whose names have a .pl suffix with their basenames; Aquarius Prolog differs from Quintus Prolog in this respect.

If *FileName* is not instantiated, an instantiation fault occurs. If *FileName* is instantiated to something other than a legal file name or if *AbsoluteFileName* is instantiated to something other than an atom, a type failure occurs.

10-5. Stream-Specific Input and Output of Terms

The following predicates are alternatives to those described in section 10-1. Unlike those, which use the current input or output stream, these take a first argument *Stream* to specify explicitly which stream to use. These predicates do not change the current input or output stream.

- **read(*+Stream*, *-Term*)**
- **write(*+Stream*, *?Term*)**
- **writeq(*+Stream*, *?Term*)**
- **write_canonical(*+Stream*, *?Term*)**
- **print(*+Stream*, *?Term*)**
- **format(*+Stream*, *+Control*, *+Args*)**

If *Stream* is not instantiated, an instantiation fault occurs. If *Stream* is instantiated to something other than a legal stream object, a type failure occurs. If *Stream* is instantiated to a stream object that does not correspond to an open stream, input or output, as the case may require, an I/O error occurs. Error behavior is otherwise as described in section 10-1.

10-6. Stream-Specific Input and Output of Characters

The following predicates are alternatives to those described in section 10-2. Unlike those, which use the current input or output stream, these take a first argument *Stream* to specify explicitly which stream to use. These predicates do not change the current input or output stream.

- **get0(*+Stream*, *-Char*)**
- **get(*+Stream*, *-Char*)**
- **skip(*+Stream*, *+Number*)**
- **put(*+Stream*, *+Number*)**
- **nl(*+Stream*)**
- **tab(*+Stream*, *+Number*)**

If *Stream* is not instantiated, an instantiation fault occurs. If *Stream* is instantiated to something other than a legal stream object, a type failure occurs. If *Stream* is instantiated to a stream object that does not correspond to an open stream, input or output, as the case may require, an I/O error occurs. Error behavior is otherwise as described in section 10-2.

10-7. DEC-10 Prolog-Style File Handling

The following predicates embody the alternative, less flexible file and stream handling scheme that originated with DEC-10 Prolog.

10-7-1. see(*+FileNameOrStream*)

see(*FileNameOrStream*) makes the stream specified by *FileNameOrStream* the current input stream. If *FileNameOrStream* is a file name, then if there is an open stream associated with the file name and if the stream was opened by see/1, see/1 makes it the current input stream, or else see/1 tries to open the file whose name is the file name for input (read mode) and, if this succeeds, makes the associated stream the current input stream. If *FileNameOrStream* is a stream object corresponding to an open input stream, see/1 makes it the current input stream.

Different file names specify different streams, even if they designate the same file. For example, foo and './foo' designate the same file, but

```
| ?- see(foo), see('./foo').
```

opens two input streams, both connected to this file.

see/1 recognizes the pseudofile user and associates it with the standard input stream.

If *FileNameOrStream* is not instantiated, an instantiation fault occurs. If *FileNameOrStream* is instantiated to something other than a legal file name or stream object, a type failure occurs. If *FileNameOrStream* designates a file that cannot be opened for any reason, then if the file_error flag is set to error (section 11-6), an I/O error occurs, or if the file_error flag is set to fail, a s imple failure occurs. If *FileNameOrStream* is instantiated to a stream object that does not correspond to an open input stream, an I/O error occurs.

10-7-2. tell(*+FileNameOrStream*)

tell(*FileNameOrStream*) makes the stream specified by *FileNameOrStream* the current output stream. If *FileNameOrStream* is a file name, then if there is an open stream associated with the file name and if the stream was opened by tell/1, tell/1 makes it the current output stream, or else tell/1 tries to open the file whose name is the file name for output (write mode) and, if this succeeds, makes the associated stream the current output stream. If *FileNameOrStream* is a stream object corresponding to an open output stream, tell/1 makes it the current output stream.

Different file names specify different streams, even if they designate the same file. For example, foo and './foo' designate the same file, but

```
| ?- tell(foo), tell('./foo').
```

opens two output streams, both connected to this file.

tell/1 recognizes the pseudofile user and associates it with the standard output stream.

If *FileNameOrStream* is not instantiated, an instantiation fault occurs. If *FileNameOrStream* is instantiated to something other than a legal file name or stream object, a type failure occurs. If *FileNameOrStream* designates a file that cannot be opened for any reason, then if the file_error flag is set to error (section 11-6), an I/O error occurs, or if the file_error flag is set to fail, a s imple failure occurs. If *FileNameOrStream* is instantiated to a stream object that does not correspond to an open output stream, an I/O error occurs.

10-7-3. seeing(-*FileNameOrStream*)

If the current input stream was opened by see/1, seeing(*FileNameOrStream*) unifies *FileNameOrStream* with the file name associated with this stream. Otherwise, seeing(*FileNameOrStream*) unifies *FileNameOrStream* with the stream object of the current input stream.

If *FileNameOrStream* is instantiated to something other than a legal file name or stream object, a type failure occurs.

10-7-4. telling(-*FileNameOrStream*)

If the current output stream was opened by tell/1, telling(*FileNameOrStream*) unifies *FileNameOrStream* with the file name associated with this stream. Otherwise, telling(*FileNameOrStream*) unifies *FileNameOrStream* with the stream object of the current output stream.

If *FileNameOrStream* is instantiated to something other than a legal file name or stream object, a type failure occurs.

10-7-5. seen

seen closes the current input stream and resets the current input stream to user_input.

seen/0 has no associated error conditions.

10-7-6. told

told closes the current output stream and resets the current output stream to user_output.

told/0 has no associated error conditions.

10-8. DEC-10 Prolog-Style TTY Character Input and Output

The following predicates are alternatives to those described in section 10-2. Unlike those, which use the current input or output stream, these use the standard input or output stream. Their names, which originated with DEC-10 Prolog, refer to the fact that the standard input and output streams are usually connected to the user's terminal, or TTY.

10-8-1. ttyget0(-*Char*)

ttyget0(*Char*) unifies *Char* with (the ASCII code of) the next character from the standard input stream. When an end of file is encountered, *Char* is unified with −1.

If *Char* is instantiated to something other than an ASCII character code, a type failure occurs. If an attempt is made to read past the end of a stream, an I/O error occurs.

10-8-2. ttyget(-*Char*)

ttyget(*Char*) unifies *Char* with (the ASCII code of) the next nonlayout character from the standard input stream. Nonlayout characters are those whose codes are in the inclusive range from 33 (0'!) t o 126 (0'~). When an end of file is encountered, *Char* is unified with −1.

If *Char* is instantiated to something other than an ASCII character code, a type failure occurs. If an attempt is made to read past the end of a stream, an I/O error occurs.

10-8-3. ttyskip(+*Number*)

ttyskip(*Number*) skips over characters from the standard input stream up to and including the first occurrence of the character specified by *Number*. *Number* should be an expression that evaluates to an ASCII character code, that is, an integer between 0 and 127.

If *Number* is not a ground term, an instantiation fault occurs. If *Number* is instantiated to something other than an expression that evaluates to an ASCII character code, a type failure occurs. If an attempt is made to read past the end of a stream, an I/O error occurs.

10-8-4. ttyput(+*Number*)

ttyput(*Number*) writes a character specified by *Number* to the standard output stream. *Number* should be an expression that evaluates to an ASCII character code, that is, an integer between 0 and 127.

If *Number* is not a ground term, an instantiation fault occurs. If *Number* is instantiated to something other than an expression that evaluates to an ASCII character code, a type failure occurs.

10-8-5. ttynl

ttynl writes a newline character to the standard output stream, as by the goal

```
| ?- ttyput(10). % ASCII code for newline character is 10.
```

ttynl/0 has no associated error conditions.

10-8-6. ttytab(+*Number*)

ttytab(*Number*) writes a number of spaces specified by *Number* to the standard output stream. *Number* should be an expression that evaluates to an integer. A call to ttytab/1 with an integer less than or equal to 0 succeeds with no effect.

If *Number* is not a ground term, an instantiation fault occurs. If *Number* is instantiated to something other than an expression that evaluates to an integer, a type failure occurs.

10-8-7. ttyflush

ttyflush flushes the buffer associated with the standard output stream.

Output to the standard output stream is not necessarily sent to the file or terminal at once. Instead, it is buffered. This predicate flushes the buffer to assure that whatever has been written to the standard output stream is actually sent to the file or terminal.

ttyflush/0 has no associated error conditions.

11. P rogram State

This chapter describes predicates that examine or modify the program state.

11-1. current_atom(*?Atom*)

current_atom(*Atom*) succeeds when *Atom* is an atom. If *Atom* is not instantiated, backtracking instantiates it successively to each known atom. If *Atom* is instantiated to something other than an atom, current_atom(*Atom*) simply fails.

Note that a nondeterminate current_atom/1 goal generates atoms in arbitrary order and sees changes to the known atom ‘‘database’’ i mmediately.

current_atom/1 has no associated error conditions.

11-2. current_predicate(*?Name*, *?Head*)

current_predicate(*Name*, *Head*) succeeds when *Head* unifies with a skeletal specification for a user-defined predicate (dynamic or compiled) whose principal functor name unifies with *Name*. I f *Head* is not instantiated, backtracking generates every possible solution. If *Name* is instantiated to something other than an atom or if *Head* is instantiated to something other than an atom or structure, current_predicate(*Name*, *Head*) simply fails. For example:

```
| ?- consult(user).
| foo(_).
| foo.
|

yes
| ?- current_predicate(foo, Head).

Head = foo ;

Head = foo(_88) ;

no
| ?-
```

The goal

```
| ?- current_predicate(_, Head).
```

may be used to backtrack through every user-defined predicate.

Note that a nondeterminate current_predicate/2 goal generates solutions in arbitrary order and sees changes to the clause database immediately.

current_predicate/2 has no associated error conditions.

11-3. predicate_property(*?Head*, *?Property*)

predicate_property(*Head*, *Property*) succeeds when *Head* unifies with a skeletal specification for a predicate (built-in or user-defined, dynamic or compiled) a property of which unifies with *Property*. *Property* is one of built_in, dynamic, interpreted or compiled. If either *Head* or *Property* is not instantiated, backtracking generates every possible solution. For example:

```
| ?- predicate_property(predicate_property(_, _), Property).

Property = built_in ;

Property = compiled ;

no
| ?- consult(user).
| foo.
|

yes
| ?- predicate_property(foo, Property).

Property = dynamic ;

Property = interpreted ;

no
| ?-
```

The goal

```
| ?- predicate_property(Head, (dynamic)).
```

may be used to backtrack through every dynamic predicate. The atom dynamic must be enclosed in parentheses because it is a prefix operator of precedence greater than

1000.

Note that a nondeterminate predicate_property/2 goal generates solutions in arbitrary order and sees changes to the clause database immediately.

The dynamic and interpreted properties are equivalent in Aquarius Prolog, but both are recognized for Quintus Prolog compatibility. Note that Quintus Prolog also recognizes the multifile, foreign, exported and imported_from(_) properties, which Aquarius Prolog does not.

If *Head* is instantiated to something other than a legal clause head or if *Property* is instantiated to something other than built_in, dynamic, interpreted or compiled, a type failure occurs.

11-4. listing

listing writes every clause of every dynamic predicate to the current output stream.

Note that listing/0 does not work on compiled predicates.

listing/0 has no associated error conditions.

11-5. listing(*?Predicates*)

listing(*Predicates*) writes every clause of the dynamic predicates specified by *Predicates* to the current output stream. *Predicates* may be:

- an atom *Name*, specifying all dynamic predicates whose principal functor name is *Name*;
- a *Name*/*Arity* pair, specifying the dynamic predicate whose principal functor name is *Name* with *Arity* arguments;
- a list [*Predicate1*, ..., *PredicateN*] of predicate specifications in either of the above forms, specifying all dynamic predicates specified by any of *Predicate1* through *PredicateN*.

If any of *Name*, *Arity* and *Predicate1* through *PredicateN* is uninstantiated, all dynamic predicates whose specifications unify with the given form are specified.

Note that listing/1 does not work on compiled predicates.

listing/1 has no associated error conditions.

11-6. prolog_flag(*+Flag*, *-Old*, *+New*)

prolog_flag(*Flag*, *Old*, *New*) unifies *Old* with the current value of flag *Flag* and sets the value of this flag to *New*. The following flags and values are recognized, where the first value listed for each flag is the default:

arithmetic_expansion — on | off

Enable or disable subgoal generation for arithmetic expressions (section 9-1) in code loaded using consult/1 and company.

character_escapes — off | on

Disable or enable character escaping in quoted atoms, strings and character codes.

unknown_predicate — warning | fail

Enable or disable the writing of a warning message on a call to an unknown predicate.

unknown — trace | fail

unknown is a synonym for unknown_predicate included for Quintus Prolog compatibility. Since Aquarius Prolog has no trace mode, trace does not indicate trace mode as it would in the Quintus Prolog development system. Instead, it is a synonym for warning, the same effect it would have in the Quintus Prolog run-time kernel.

type_failure — warning | fail

Enable or disable the writing of a warning message when an argument to a built-in predicate is inappropriately instantiated.

file_error — error | fail

Enable or disable the entering of an I/O error condition when an attempt is made to open a file that cannot be opened.

fileerrors — on | off

fileerrors is a synonym for file_error included for Quintus Prolog compatibility. on and off are synonyms for error and fail, respectively.

To inspect the value of a flag without changing it, use prolog_flag/2 or the goal

```
| ?- prolog_flag(Flag, Value, Value)
```

where *Flag* is bound to one of the flags listed above.

Note that Quintus Prolog also provides the debugging, gc, gc_margin, gc_trace and module flags, which Aquarius Prolog does not.

If *Flag* is not instantiated or if *New* is a variable distinct from *Old*, an instantiation fault occurs. If *Flag* is instantiated to something other than a recognized flag or if *New* is instantiated to something other than a recognized value of flag *Flag*, a type failure occurs.

11-7. prolog_flag(*?Flag*, *-Value*)

prolog_flag(*Flag*, *Value*) is equivalent to prolog_flag(*Flag*, *Value*, *Value*), except that if *Flag* is not instantiated, backtracking instantiates it successively to each recognized flag.

If *Flag* is instantiated to something other than a recognized flag, a type failure occurs.

11-8. unknown(*-Old*, *+New*)

unknown(*Old*, *New*) unifies *Old* with the current value of the unknown flag and sets the value of the unknown flag to *New*. It then writes a message to the standard output stream indicating what will happen on subsequent calls to unknown predicates. Note that

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

returns *Value* without changing the unknown flag.

unknown/2 is a special case of prolog_flag/3 included for Quintus Prolog compatibility.

If *New* is a variable distinct from *Old*, an instantiation fault occurs. If *New* is instantiated to something other than trace or fail, the two legal values of the unknown flag, a type failure occurs.

11-9. fileerrors

fileerrors sets the value of the fileerrors flag to on.

fileerrors/0 is a special case of prolog_flag/3 included for Quintus Prolog compatibility.

fileerrors/0 has no associated error conditions.

11-10. nofileerrors

nofileerrors sets the value of the fileerrors flag to off.

nofileerrors/0 is a special case of prolog_flag/3 included for Quintus Prolog compatibility.

nofileerrors/0 has no associated error conditions.

12. Databases

Aquarius Prolog provides two databases, a clause database of dynamic predicate clauses and an internal or ‘‘recorded’’ database of general terms. The clause database is manipulated by assert/[1,2], retract/1 etc.; chapter 13 describes this in detail. The internal database is manipulated by recorda/3, recorded/3 etc.; chapter 14 describes this in detail. This chapter describes important features common to both databases and a pair of predicates, erase/1 and instance/2, that operate on either database.

12-1. Virtual Copy Semantics

Consider a dynamic predicate foo/0:

```
| ?- consult(user).
| foo :- assertz(foo), fail.
|

yes
| ?-
```

A call to foo/0 adds a new last clause to foo/0; does the call then succeed?

If the clause database operates according to so-called immediate update semantics, the call succeeds. In this convention, a change to the database becomes globally visible on the success of the built-in predicate that makes the change. Several Prolog implementations, including Quintus Prolog prior to Release 2.0, have used this scheme. Besides occasionally leading to obscure code, immediate update semantics affects memory usage and execution speed significantly for the worse. (See Timothy G. Lindholm and Richard A. O’Keefe, ‘‘Efficient Implementation of a Defensible Semantics for Dynamic Prolog Code,’’ Proceedings of the Fourth International Conference on Logic Programming, MIT Press, 1987, pp. 21-39 for a discussion of these issues.)

A coherent alternative to immediate update semantics is so-called virtual copy or logical semantics. If the clause database operates according to this scheme, the definition of a dynamic predicate visible to a call is frozen when the call occurs; for the duration of the call, the predicate has exactly the clauses it has at the outset of the call. The call to foo/0 above fails because the new last clause is invisible to it. Aquarius Prolog, like Quintus Prolog since Release 2.0, adopts this convention. Conceptually, it is as though a dynamic predicate call makes a copy of the predicate definition and then runs the copy instead of the original. Changes to the predicate definition are made in the database at once but not in the copy. Practically, the system does not actually make a copy, which would be quite expensive in general, but it keeps track of calls and database modifications to produce the same effect. The memory usage and execution speed of this scheme are superior to immediate update.

As a further example of how virtual copy semantics works, consider a dynamic predicate bar/1:

```
| ?- consult(user).
| bar(1).
| bar(2).
|

yes
| ?- bar(N), write(N), nl, retract(bar(2)), fail.
1
2

no
| ?- bar(N), write(N), nl, fail.
1

no
| ?-
```

At the first call to bar/1, there are two clauses. The second clause is retracted from the database, but the call continues to see it, and therefore *N* becomes bound to 2. At the second call to bar/1, the second clause is no longer in the database, so the call sees only the first clause.

So far we have discussed virtual copy semantics only with respect to assertion and retraction on the clause database, but Aquarius Prolog, like Quintus Prolog since Release 2.0, adopts it for all operations on either the clause database or the internal database. Consider an internal database analogue of the bar/1 example above:

```
| ?- recordz(bar, 1, Ref0),
     recordz(bar, 2, Ref1).

Ref0 = '$ ref'(1),
Ref1 = '$ ref'(3)

| ?- recorded(bar, N, _), write(N), nl, erase('$ ref'(3)), fail.
1
2

no
| ?- recorded(bar, N, _), write(N), nl, fail.
1

no
| ?-
```

Virtual copy semantics is a uniform principle of database operations in Aquarius Prolog.

12-2. Database References

A database reference is a term that identifies a clause in the clause database or a record in the internal database. Each database item is associated with a unique database reference, assigned at item insertion into the database, retained throughout item presence in the database and not reassigned at item removal from the database (if this occurs), that is, unique not only at any given time but over the whole duration of execution. Any attempt to use a reference to an item that has been removed simply fails. Note that a reference is unique with respect to both databases; clauses and records never have identical references.

In Aquarius Prolog, a database reference is an ordinary Prolog structure of the form '$ ref'(*N*), where *N* is a nonnegative integer that is even for a record and odd for a clause. It is unwise to write programs that rely on this, however, since it is by no means standard; at present, there is no standard, formal or de facto, for the form of Prolog database references. In practice this constitutes no real difficulty, since references are ordinarily generated by built-in predicates that add new items to the databases, and therefore programs can treat references as ‘‘black boxes.’’

Use of database references with the clause database is optional, as each of the predicates that inserts, inspects or removes a clause is provided in two versions, one that includes a reference among its arguments and one that does not, for example, assert/2 and assert/1. With the internal database, use of database references is less optional, as all the predicates that insert, inspect or remove a record include a reference among their arguments, and though references are not essential for inserting or inspecting, they are essential for removing, since the predicate erase/1 that removes a record requires a reference. This arrangement is customary for the standard built-in predicates that manipulate the databases. However, the user can easily construct an interface to the internal database that hides references, if desired. Such an interface might include, for example,

```
retract_record(Key, Term) :-
        recorded(Key, Term, Ref),
        erase(Ref).
```

In this way, internal database operations can be made isomorphic to reference-free clause database operations. On the other hand, a program that manipulates the databases frequently using operations with references instead of reference-free versions may run significantly faster by avoiding repeated searches through the clauses under a head principal functor or the records under a key. The user must choose appropriately for each program.

12-3. erase(*+Ref*)

erase(*Ref*) erases from the clause or record database the clause or record whose database reference is *Ref*. Any further attempt to use the reference fails. However, an erased item continues to be seen by goals that search the database (nondeterminate clause/[2,3] or recorded/3 goals, for example) that were called before the erase/1 goal and whose scope of potential backtracking includes the erased item.

If *Ref* is not instantiated, an instantiation fault occurs. If *Ref* is instantiated to something other than a legal database reference, a type failure occurs.

12-4. instance(*+Ref*, *-Instance*)

instance(*Ref*, *Instance*) unifies *Instance* with the clause or recorded term associated with database reference *Ref*. If the reference is to a unit clause, *Instance* is unified with *UnitClause* :– true.

If *Ref* is not instantiated, an instantiation fault occurs. If *Ref* is instantiated to something other than a legal database reference, a type failure occurs.

13. The Clause Database

The clause database stores dynamic predicate clauses. As a database, it associates an ordered list of terms with each nonnumeric principal functor; each term associated with a functor has the form of a dynamic predicate clause the functor of whose head is this functor.

Clauses are indexed on their first arguments. The indexing speeds execution of dynamic predicates, but it slows insertion into the clause database. For an application that needs to insert items into a database and then refer to them seldom, it may be more efficient to use the internal database, which has no indexing; see chapter 14.

13-1. assert(*+Clause*), assert(*+Clause*, *-Ref*)

assert(*Clause*) adds *Clause* to the clause database. In principle the position of the newly-added clause with respect to other clauses of the same predicate is arbitrary (though in practice Aquarius Prolog does observe a particular convention). To make the position of the new clause explicit, use asserta/1 or assertz/1. *Clause* should be instantiated to a term having the form of a dynamic predicate clause, that is, it should be instantiated to a nonnumeric term, and if it has the form *Head* :– *Body*, *Head* should also be instantiated to a nonnumeric term.

assert(*Clause*, *Ref*) is equivalent to assert(*Clause*) except that it returns in *Ref* a database reference that uniquely identifies the newly-asserted clause. It may be possible to guess a database reference so as to instantiate *Ref* to a value that unifies with the return value, but this is not encouraged; *Ref* should be uninstantiated. assert/2 fails if *Ref* is instantiated but does not unify with the return value.

assert/[1,2] may not be applied to a static or built-in predicate.

Note that if you want to pass a term of the form *Head* :– *Body* to assert/[1,2], y ou must put it in parentheses because the infix operator precedence of :– is greater than 1000. For example,

```
| ?- assert(a:-write(a)).
```

is a syntax error; instead, write

```
| ?- assert((a:-write(a))).
```

If *Clause* is not instantiated to a legal clause or if *Ref* is instantiated to something other than a legal database reference, a type failure occurs. An attempt to apply assert/[1,2] to a static or built-in predicate is a system restriction violation.

13-2. asserta(*+Clause*), asserta(*+Clause*, *-Ref*)

asserta(*Clause*) adds *Clause* to the clause database as the first clause of a predicate. *Clause* should be instantiated to a term having the form of a dynamic predicate clause, that is, it should be instantiated to a nonnumeric term, and if it has the form *Head* :– *Body*, *Head* should also be instantiated to a nonnumeric term.

asserta(*Clause*, *Ref*) is equivalent to asserta(*Clause*) except that it returns in *Ref* a database reference that uniquely identifies the newly-asserted clause. It may be possible to guess a database reference so as to instantiate *Ref* to a value that unifies with the return value, but this is not encouraged; *Ref* should be uninstantiated. asserta/2 fails if *Ref* is instantiated but does not unify with the return value.

asserta/[1,2] may not be applied to a static or built-in predicate.

Note that if you want to pass a term of the form *Head* :– *Body* to asserta/[1,2], you must put it in parentheses because the infix operator precedence of :– is greater than 1000.

If *Clause* is not instantiated to a legal clause or if *Ref* is instantiated to something other than a legal database reference, a type failure occurs. An attempt to apply asserta/[1,2] to a static or built-in predicate is a system restriction violation.

13-3. assertz(*+Clause*), assertz(*+Clause*, *-Ref*)

assertz(*Clause*) adds *Clause* to the clause database as the last clause of a predicate. *Clause* should be instantiated to a term having the form of a dynamic predicate clause, that is, it should be instantiated to a nonnumeric term, and if it has the form *Head* :– *Body*, *Head* should also be instantiated to a nonnumeric term.

assertz(*Clause*, *Ref*) is equivalent to assertz(*Clause*) except that it returns in *Ref* a database reference that uniquely identifies the newly-asserted clause. It may be possible to guess a database reference so as to instantiate *Ref* to a value that unifies with the return value, but this is not encouraged; *Ref* should be uninstantiated. assertz/2 fails if *Ref* is instantiated but does not unify with the return value.

A newly-asserted clause is not seen by goals that search the database (nondeterminate clause/[2,3] goals, for example) and that were called before the assertz/[1,2] goal.

assertz/[1,2] may not be applied to a static or built-in predicate.

Note that if you want to pass a term of the form *Head* :– *Body* to assertz/[1,2], you must put it in parentheses because the infix operator precedence of :– is greater than 1000.

If *Clause* is not instantiated to a legal clause or if *Ref* is instantiated to something other than a legal database reference, a type failure occurs. An attempt to apply assertz/[1,2] to a static or built-in predicate is a system restriction violation.

13-4. clause(*+Head*, *?Body*), clause(*?Head*, *?Body*, *?Ref*)

clause(*Head*, *Body*) searches the clause database for a clause whose head unifies with *Head* and whose body unifies with *Body*. A unit clause is regarded as having a body consisting of the single goal true. *Head* and *Body* are unified successively with each head and body asserted for the dynamic predicate associated with *Head* until a match is found or no more clauses remain. Backtracking continues the search.

clause(*Head*, *Body*, *Ref*) searches the clause database for a clause whose head unifies with *Head*, whose body unifies with *Body* and whose database reference unifies with *Ref*. If *Ref* is instantiated, *Head* and *Body* are unified with the head and body associated with *Ref*. Otherwise, *Head* and *Body* are unified successively with each head and body asserted for the dynamic predicate associated with *Head* until a match is found or no more clauses remain. Backtracking continues the search.

clause/[2,3] may not applied to compiled or built-in predicates (for which Aquarius Prolog does not retain clauses in any form resembling Prolog). A call to clause/[2,3] with an unknown predicate simply fails.

If both *Head* and *Ref* are not instantiated, an instantiation fault occurs. If *Head* is instantiated to something other than a legal clause head or if *Ref* is instantiated to something other than a legal database reference, a type failure occurs. An attempt to apply clause/[2,3] to a compiled or built-in predicate is a system restriction violation.

13-5. retract(*+Clause*)

retract(*Clause*) erases the first clause in the clause database that matches *Clause*. retract(*Clause*) is nondeterminate; as backtracking returns to it, successive clauses matching *Clause* are erased. If no clause matches, retract(*Clause*) fails. *Clause* must be instantiated to a term having the form of a dynamic predicate clause. A unit clause is regarded as having a body consisting of the single goal true. For example:

```
| ?- retract((a(_) :- _)), fail.
```

erases every clause, including unit clauses, of the dynamic predicate a/1.

Retracted clauses continue to be seen by goals that search the database (nondeterminate clause/[2,3] goals, for example) that were called before the retract/1 goal and whose scope of potential backtracking includes the retracted clauses.

retract/1 may not be applied to static or built-in predicates. A c all to retract/1 with an unknown predicate simply fails.

If *Clause* is not instantiated to a legal clause, a type failure occurs. An attempt to apply retract/1 to a static or built-in predicate is a system restriction violation.

13-6. retractall(*+Head*)

retractall(*Head*) erases every clause in the clause database whose head matches *Head*. *Head* must be instantiated to a term having the form of a call to a dynamic predicate. For example:

```
| ?- retractall(a(_)).
```

erases every clause of the dynamic predicate a/1.

Retracted clauses continue to be seen by goals that search the database (nondeterminate clause/[2,3] goals, for example) that were called before the retractall/1 goal and whose scope of potential backtracking includes the retracted clauses.

retractall/1 may not be applied to static or built-in predicates. A c all to retractall with an unknown predicate succeeds with no effect.

retractall/1 is useful for erasing every clause of a dynamic predicate without forgetting that it is dynamic. abolish/[1,2] not only erases the clauses but forgets everything about the predicate. retractall/1 only erases the clauses.

retractall/1 is determinate; it erases every clause whose head matches *Head*. N o variables in *Head* are instantiated in the process. If there is no clause whose head matches *Head*, retractall/1 succeeds with no effect.

If *Head* is not instantiated to a legal clause head, a type failure occurs. An attempt to apply retractall/1 to a static or built-in predicate is a system restriction violation.

13-7. abolish(*+Predicates*)

abolish(*Predicates*) abolishes the dynamic predicates specified by *Predicates*. Not only their clauses but all knowledge of their existence is wiped out. *Predicates* may be:

- a *Name*/*Arity* pair, specifying the dynamic predicate whose principal functor name is *Name* with *Arity* arguments;
- a list [*Predicate1*, . .., *PredicateN*] of predicate specifications in the above form, specifying all dynamic predicates specified by any of *Predicate1* through *PredicateN*.

Any dynamic predicate may be abolished. Static predicates may not be abolished; Aquarius Prolog differs from Quintus Prolog in this respect. Built-in predicates may not be abolished. A call to abolish/1 with an unknown predicate succeeds with no effect.

Abolished predicates continue to be seen by goals that search the database (nondeterminate clause/[2,3] goals, for example) and that were called before the abolish/1 goal. For example:

```
| ?- consult(user).
| :- dynamic a/1.
| a(1).
| a(2).
|

yes
| ?- a(N), write(N), nl, abolish(a/1), fail.
1
2

no
| ?- a(N).
[WARNING: a/1 is unknown:
 call(a(_21))]

no
| ?-
```

If *Predicates* is not instantiated or if *Predicates* is a list of which any element is not instantiated, an instantiation fault occurs. If *Predicates* is instantiated to something other than a legal predicate specification or list of predicate specifications as described above, a type failure occurs. An attempt to abolish a static or built-in predicate is a system restriction violation.

13-8. abolish(*+Name*, *+Arity*)

abolish(*Name*, *Arity*) abolishes the dynamic predicate whose principal functor name is *Name* with *Arity* arguments. Not only its clauses but all knowledge of its existence is wiped out.

Any dynamic predicate may be abolished. Static predicates may not be abolished; Aquarius Prolog differs from Quintus Prolog in this respect. Built-in predicates may not be abolished. A call to abolish/2 with an unknown predicate succeeds with no effect.

Abolished predicates continue to be seen by goals that search the database (nondeterminate clause/[2,3] goals, for example) and that were called before the abolish/2 goal.

abolish/2 is a special case of abolish/1 included for compatibility with earlier Prolog implementations.

If either of *Name* or *Arity* is not instantiated, an instantiation fault occurs. If *Name* is instantiated to something other than an atom or if *Arity* is instantiated to something other than a nonnegative integer, a type failure occurs. An attempt to abolish a static or built-in predicate is a system restriction violation.

14. The Internal Database

The internal database associates an ordered list of terms with each integer and non-numeric principal functor, which constitutes the key under which the terms are recorded. The built-in predicates that insert and inspect records include a key among their arguments. Any Prolog term other than a float may be given as such an argument, but if a compound term is given, only its principal functor is significant; foo(0) and foo(1) designate the same key. There is no restriction on the terms that may be recorded under a key.

Note that various Aquarius Prolog built-in predicates use the internal database for their own purposes. The keys they use are of the form '$ ...'. To avoid chaos, programs should avoid such keys.

The internal database operates by linear search through the terms under a key; there is no indexing or other search acceleration mechanism. This makes insertion faster but access slower for the internal database than for the clause database, which has first argument indexing. For an application that needs to insert items into a database and then refer to them often, the clause database may be preferable, even if the items are not to be executed as dynamic predicates; see chapter 13.

14-1. recorda(*+Key*, *?Term*, *-Ref*)

recorda(*Key*, *Term*, *Ref*) adds *Term* to the record database as the first term under *Key* and returns a database reference to the newly-recorded term in *Ref*. If *Key* is a compound term, only its principal functor is significant.

If *Key* is not instantiated, an instantiation fault occurs. If *Key* is instantiated to something other than an integer, atom or compound term or if *Ref* is instantiated to something other than a legal database reference, a type failure occurs.

14-2. recordz(*+Key*, *?Term*, *-Ref*)

recordz(*Key*, *Term*, *Ref*) adds *Term* to the record database as the last term under *Key* and returns a database reference to the newly-recorded term in *Ref*. If *Key* is a compound term, only its principal functor is significant.

A newly-recorded term is not seen by goals that search the database (nondeterminate recorded/3 goals) and that were called before the recordz/3 goal.

If *Key* is not instantiated, an instantiation fault occurs. If *Key* is instantiated to something other than an integer, a tom or compound term or if *Ref* is instantiated to something other than a legal database reference, a type failure occurs.

14-3. recorded(*?Key*, *?Term*, *?Ref*)

recorded(*Key*, *Term*, *Ref*) searches the record database for a record under *Key* whose term unifies with *Term* and whose database reference unifies with *Ref*. I f *Ref* is instantiated, *Key* and *Term* are unified with the key and term associated with *Ref*. Otherwise, *Term* is unified successively with each term recorded under *Key* until a match is found or no more records remain. Backtracking continues the search. If *Key* is a compound term, only its principal functor is significant.

If both *Key* and *Ref* are not instantiated, an instantiation fault occurs. If *Key* is instantiated to something other than an integer, a tom or compound term or if *Ref* is instantiated to something other than a legal database reference, a type failure occurs.

14-4. current_key(*?KeyName*, *?KeyTerm*)

current_key(*KeyName*, *KeyTerm*) succeeds when *KeyTerm* unifies with a key under which at least one record is currently recorded and whose principal functor name unifies with *KeyName*. I f *KeyTerm* is a compound term, only its principal functor is significant. If *KeyTerm* is not instantiated, backtracking generates every possible solution. For example:

```
| ?- recorda(foo, a, _).

yes
| ?- recorda(foo(_), b, _).

yes
| ?- recorda(0, c, _).

yes
| ?- recorda([_], d, _).

yes
| ?- current_key(foo, KeyTerm).

KeyTerm = foo ;

KeyTerm = foo(_87) ;

no
```

```
| ?- current_key(KeyName, KeyTerm).

KeyName = 0,
KeyTerm = 0 ;

KeyName = '.',
KeyTerm = [_98|_99] ;

KeyName = foo,
KeyTerm = foo ;

KeyName = foo,
KeyTerm = foo(_99) ;

no
| ?-
```

Note that a nondeterminate current_key/2 goal generates solutions in arbitrary order and sees changes to the record database immediately.

If *KeyName* is instantiated to something other than an atomic term or if *KeyTerm* is instantiated to something other than an integer, a tom or compound term, a type failure occurs.

15. Sets and Bags

This chapter describes predicates that collect all solutions to a goal.

15-1. setof(*?Template*, *+Generator*, *-Set*)

setof(*Template*, *Generator*, *Set*) unifies *Set* with the set of all instances of *Template* such that *Generator* is provable, where this set is nonempty. *Generator* is a goal to be invoked as by call/1. *Set* is a list of terms in standard order with no duplicates.

The set to be enumerated should be enumerable in finite time. Instances of *Template* such that *Generator* is provable may contain variables, but *Set* is then only an imperfect representation of an infinite set.

If *Generator* contains uninstantiated variables that do not also occur in *Template*, setof/3 can succeed nondeterminately, producing alternative values of *Set* for different instantiations of these free variables in *Generator*. It is to allow for such usage that *Set* is constrained to be nonempty. For example:

```
| ?- consult(user).
| likes(mary, food).
| likes(mary, wine).
| likes(john, wine).
| likes(john, mary).
|

yes
| ?- setof(X, likes(X, Y), Set).

X = _29,
Y = food,
Set = [mary] ;

X = _29,
Y = mary,
Set = [john] ;

X = _29,
Y = wine,
Set = [john,mary] ;

no
| ?-
```

Note how backtracking generates alternative solutions. All this data can also be collected with a single call; for example:

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

S = _43,
X = _59,
Y = _81,
Set = [likes([john],mary),likes([john,mary],wine),likes([mary],food)] ;

no
| ?-
```

An uninstantiated variable in *Generator* that does not also occur in *Template* is not considered free if it is the first argument of a findall/3, bagof/3 or setof/3 structure, as `Y` is in the preceding example, or if it is explicitly existentially quantified, as `Y` is in the following example,

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

X = _29,
Y = _43,
Set = [john,mary] ;

no
| ?-
```

X^Goal means ''there exists *X* such that *Goal* is provable.''

If *Set* is instantiated to something other than a list, a type failure occurs. Other error conditions may arise from execution of *Generator*.

15-2. bagof(*?Template*, *+Generator*, *-Bag*)

bagof/3 is identical to setof/3 except that the list of provable instances is not ordered and may contain duplicates. Such lists are known as bags rather than sets. Bags in general take less time and space to generate than sets.

As with setof/3, i f Bag is instantiated to something other than a list, a type failure occurs. Other error conditions may arise from execution of *Generator*.

15-3. findall(*?Template*, *+Generator*, *-List*)

findall/3 is identical to bagof/3 except that all free variables in *Generator* (uninstantiated variables that do not also occur in *Template*) are taken to be existentially quantified. As a corollary, findall/3 is determinate, and *List* may be empty. Dispensing with free variable tracking in general saves time and space.

Note that Quintus Prolog does not provide findall/3. findall/3 is described on pp. 162-163 of Clocksin & Mellish. The code there has a bug, which Aquarius Prolog has not.

As with bagof/3, if List is instantiated to something other than a list, a type failure occurs. Other error conditions may arise from execution of *Generator*.

15-4. *?X^+Goal*

X^Goal means ''there exists *X* such that *Goal* is provable.'' Use of this explicit existential quantifier outside setof/3 and bagof/3 is superfluous and is treated as equivalent to call(*Goal*).

^/2 has no associated error conditions.

16. Grammar Rules

Aquarius Prolog provides the grammar rule notation for definite clause grammars, as set forth in Clocksin & Mellish, for example. Assuming familiarity with grammar rules, this chapter primarily describes predicates associated with them.

16-1. Definite Clause Grammars

Definite clause grammars may be expressed in Prolog using the grammar rule notation

<head> --> *<body>*

which means ‘‘*<body>* is a possible form of *<head>*.’’ Note the following characteristics of grammar rules:

- A nonterminal may be any term other than a number.
- A terminal may be any term whatever. Sequences of terminals are enclosed in lists. If the terminals are ASCII character codes, the lists may be written as strings.
- *<head>* should be a nonterminal optionally followed by a sequence of terminals.
- *<body>* may contain conjunctions, disjunctions, cuts, conditionals and negations in the standard syntax. In addition to nonterminals and terminals, *<body>* may contain arbitrary goals enclosed in curly brackets ({ and }).

Grammar rules are of course merely abbreviations for ordinary clauses, into which they are translated as they are compiled using the Aquarius Prolog compiler or loaded using consult/1 and company (section 3-1). Usually, the procedural interpretation is that each clause takes an input list, analyzes some initial portion and gives the remaining portion (possibly enlarged) as output. Handling of the input, output and any intermediate lists is generated automatically in the translation from grammar rules into ordinary clauses. For example:

```
?- consult(user).
sum(X) --> number(X), !.
sum(X) --> number(Y), [+], sum(Z), {X is Y+Z}.

number(X) --> [X], {number(X)}.
^D

yes
```

```
| ?- listing([sum/3,number/3]).

sum(A,B,C) :-
        number(A,B,C),
        !.
sum(A,B,C) :-
        number(D,B,E),
        'C'(E,+,F),
        sum(G,F,C),
        '$ add'(D,G,A).

number(A,B,C) :-
        'C'(B,A,C),
        number(A).

yes
| ?- sum(X, [-5, +, 3.14], []).

X = -1.86

| ?-
```

16-2. expand_term(*+Term*, *-Term1*)

expand_term(*Term*, *Term1*) translates *Term* into *Term1*. consult/1 and company (section 3-1) use expand_term/2 to translate grammar rules. By default, if *Term* is a grammar rule, *Term1* is the corresponding ordinary clause, otherwise *Term1* is simply *Term* unchanged. However, you may change this behavior by defining the predicate term_expansion/2. expand_term/2 calls term_expansion/2 first, and if this succeeds, the standard grammar rule translation is not attempted. This is effective only in files processed by consult/1 and company. To achieve this effect with the compiler, load your Prolog source file using consult/1, call listing/0 (section 11-4) with output to a file and compile that file.

expand_term/2 has no associated error conditions (except for any that may arise from execution of term_expansion/2).

16-3. phrase(*+Form*, *?List*, *?Rest*)

phrase(*Form*, *List*, *Rest*) succeeds when some initial portion of *List* has the form *Form* and the remaining portion of *List* is *Rest*. *Form* is usually a nonterminal, that is, it unifies with the nonterminal in the head of some grammar rule that has been loaded. For example:

```
| ?- consult(user).
| sum(X) --> number(X), !.
| sum(X) --> number(Y), [+], sum(Z), {X is Y+Z}.
|
| number(X) --> [X], {number(X)}.
| ^D

yes
| ?- phrase(sum(X), [-5, +, 3.14], []).

X = -1.86

| ?-
```

Form may also be a terminal enclosed in a list. In this usage, phrase/3 allows for grammar rules what call/1 (section 6-7) allows for ordinary clauses, namely, t he appearance of variables in bodies. For example:

```
| ?- consult(user).
| optional(Item) --> Item, !.
| optional(_)    --> [].
| ^D

yes
| ?- listing(optional/3).

optional(A,B,C) :-
        phrase(A,B,C),
        !.
optional(A,B,C) :-
        B=C.

yes
| ?-
```

If *Form* is instantiated to something other than a nonterminal or terminal enclosed in a l ist, phrase(*Form*, *List*, *Rest*) simply fails.

If *Form* is not instantiated, an instantiation fault occurs.

16-4. phrase(*+Form*, *?List*)

phrase(*Form*, *List*) is equivalent to phrase(*Form*, *List*, [ ]). This is a convenient way to start execution of grammar rules.

16-5. 'C'(*?S*, *?Terminal*, *?S1*)

'C'(*S*, *Terminal*, *S1*) behaves as if 'C'/3 were defined with the clause:

'C'([*X*|*S*], *X*, *S*).

'C'/3 is used in the translations of grammar rules. It is not ordinarily of interest to users.

'C'/3 has no associated error conditions.

17. Destructive Assignment

Aquarius Prolog provides two varieties of destructive assignment to arguments of compound terms, backtrackable destructive assignment and stepped destructive assignment.

17-1. Backtrackable Destructive Assignment

Backtrackable destructive assignment is available via setarg/3. A setarg/3 goal is much like an arg/3 (section 7-20) goal that instantiates an uninstantiated argument of a compound term, the obvious difference being that setarg/3 does not require the argument to be uninstantiated. Whatever the value of the argument, the setarg/3 goal overwrites it, whereas the arg/3 goal would simply fail (unless its third argument unified with the value). Before the value is overwritten, a record of it is made so that it can be restored in the event of backtracking past the setarg/3 goal.

You should not permit an argument you manipulate using setarg/3 to be aliased, that is, if the argument is a variable for some period, you should not permit it to be unified with another variable during this period. If aliasing occurs, the effect of setarg/3 on aliases of the argument is undefined.

Global dataflow analysis (section 2-2-5) is incompatible with backtrackable destructive assignment. You must not perform analysis on programs that use setarg/3.

Backtrackable destructive assignment is available in SICStus Prolog, but not Quintus Prolog.

17-1-1. setarg(*+Number*, *+Term*, *+Arg*)

setarg(*Number*, *Term*, *Arg*) sets the *Number*th argument of *Term* to *Arg*. This operation is destructive, meaning that the present value of the argument is irrelevant, but backtrackable, meaning that the value is restored in the event of subsequent backtracking past the setarg/3 goal. *Number* must be instantiated to a positive integer, and *Term* must be instantiated to something other than a number. For example:

```
| ?- X = foo(1),
     ( setarg(1, X, 2), write(X), nl, fail
     ; write(X), nl
     ).
foo(2)
foo(1)

X = foo(1)

| ?-
```

Note that Quintus Prolog does not provide setarg/3.

If *Number* or *Term* is not instantiated, an instantiation fault occurs. If *Number* is instantiated to something other than a positive integer or if *Term* is instantiated to a number, a type failure occurs.

17-2. Stepped Destructive Assignment

Stepped destructive assignment is available via steparg/3 and step/0. A steparg/3 goal is like a setarg/3 goal except that it has no effect on the argument of the compound term until the next step/0 goal. The steparg/3 goal queues an assignment but does not carry it out. Until the next step/0 goal, there may be any number of steparg/3 goals that queue assignments to the same argument; when step/0 is called, the last assignment gives the subsequent value of the argument. Assignments are unqueued on backtracking past steparg/3 goals that queued them and undone on backtracking past step/0 goals that carried them out. Thus, ''stepped destructive assignment'' is as backtrackable as ''backtrackable destructive assignment.'' However, stepped destructive assignment is meant to be used in such a way that backtracking over step/0 goals never occurs.

A step/0 goal is analogous to a clock pulse in a digital logic circuit: various operations proceed in real time throughout the system, but the ''state of the system,'' embodied in register values in the digital logic circuit or compound terms in steparg/3 goals, does not change until the next clock pulse or step/0 goal. Indeed, this analogy points toward the main motivation for stepped destructive assignment, which is to facilitate the simulation of finite state machines such as microprocessors by Prolog programs.

As with setarg/3, you should not permit an argument you manipulate using steparg/3 and step/0 to be aliased. If aliasing occurs, the effect of steparg/3 and step/0 on aliases of the argument is undefined.

Global dataflow analysis (section 2-2-5) is incompatible with stepped destructive assignment. You must not perform analysis on programs that use steparg/3 and step/0.

Stepped destructive assignment is not available in any other Prolog implementation.

17-2-1. steparg(*+Number*, *+Term*, *+Arg*)

steparg(*Number*, *Term*, *Arg*) queues an assignment that sets the *Number*th argument of *Term* to *Arg* at the next step/0 goal. This operation is destructive, meaning that the present value of the argument is irrelevant, but backtrackable, meaning that the assignment is unqueued in the event of failure before the next step/0 goal or undone in the event of failure after the next step/0 goal. *Number* must be instantiated to a positive integer, and *Term* must be instantiated to something other than a number. For example:

```
| ?- X = foo(1),
     ( steparg(1, X, 2), write(X), nl, step, write(X), nl, fail
     ; write(X), nl
     ).
foo(1)
foo(2)
foo(1)

X = foo(1)

| ?-
```

Note that Quintus Prolog does not provide steparg/3.

If *Number* or *Term* is not instantiated, an instantiation fault occurs. If *Number* is instantiated to something other than a positive integer or if *Term* is instantiated to a number, a type failure occurs.

17-2-2. step

step carries out assignments to arguments of compound terms queued by steparg/3 goals since the last step/0 goal. This operation is destructive, meaning that the present values of the arguments are irrelevant, but backtrackable, meaning that the assignments are undone in the event of subsequent backtracking past the step/0 goal. The preceding section includes an example.

Note that Quintus Prolog does not provide step/0.

step/0 has no associated error conditions.

18. Access to UNIX

Aquarius Prolog, like Quintus Prolog, provides access to UNIX via the built-in predicate unix/1.

18-1. unix(argv(*-Args*))

unix(argv(*Args*)) unifies *Args* with a list of constants that represent the arguments typed on the command line at program start-up. Arguments that are numbers, whether integers or floating point numbers, are returned as such, and other arguments are returned as atoms. For example, if the current working directory is the Aquarius Prolog installation directory of a MIPS R3000, RISC/os machine:

```
% top_level/installed/top_level.MIPS,RISCOS,AQUARIUS -5 3.14 foo/bar

Aquarius Prolog version 1.0 (MIPS R3000, RISC/os)

| ?- unix(argv(Args)).

Args = [-5,3.14,'foo/bar']

| ?-
```

Note that memory management arguments (section 19-2) are not returned to programs by unix/1 but are seen by the Aquarius Prolog memory and signal manager only.

If *Args* is instantiated to something other than a list, a type failure occurs.

18-2. unix(cd)

unix(cd) changes the current working directory of the process to your home directory.

unix(cd) has no associated error conditions.

18-3. unix(cd(*+Directory*))

unix(cd(*Directory*)) changes the current working directory of the process to *Directory*, which should be an atom representing an absolute or relative path name.

If *Directory* is not instantiated, an instantiation fault occurs. If *Directory* is instantiated to something other than an atom, a type failure occurs. If it is impossible to change to *Directory* for any reason, an I/O error occurs.

18-4. unix(shell)

unix(shell) starts up an interactive session of the shell specified by your SHELL environment variable. To end the session, type the end of file character (usually control-D), or in the C-shell, if you have set ignoreeof (perhaps in your .cshrc file), type exit.

unix(shell) has no associated error conditions.

18-5. unix(shell(*+Command*))

unix(shell(*Command*)) starts up a noninteractive session of the shell specified by your SHELL environment variable and passes *Command* to it for execution. *Command* should be an atom. If the shell returns a nonzero value, unix(shell(*Command*)) simply fails. This occurs if the shell considers *Command* invalid in some way.

If *Command* is not instantiated, an instantiation fault occurs. If *Command* is instantiated to something other than an atom, a type failure occurs.

18-6. unix(system(*+Command*))

unix(system(*Command*)) starts up a noninteractive session of the Bourne shell /bin/sh and passes *Command* to it for execution. *Command* should be an atom. If the shell returns a nonzero value, unix(system(*Command*)) simply fails. This occurs if the shell considers *Command* invalid in some way.

If *Command* is not instantiated, an instantiation fault occurs. If *Command* is instantiated to something other than an atom, a type failure occurs.

19. Memory Management

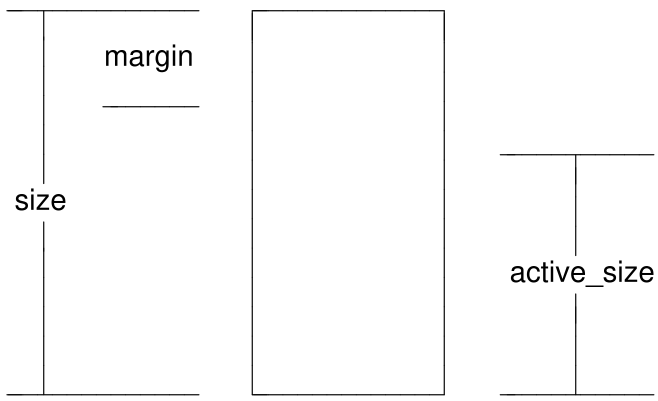
This chapter discusses memory management in Aquarius Prolog and how programs can monitor and influence it. More than any other chapter, this one deals with the internal workings of Aquarius Prolog. The features it describes are highly specific to Aquarius Prolog; very little here corresponds in detail to Quintus Prolog or any other implementation besides Aquarius Prolog. It should be possible to use Aquarius Prolog extensively without knowledge of this chapter.

19-1. Memory Organization

Aquarius Prolog maintains data in seven sections of memory that may change size during program execution. The sections and their contents are as follows:

heap	variables, compound terms
trail	variable binding records, backtrackable destructive assignment records
stack	predicate activation records, backtracking records
firm heap	atom names, asserted clauses, recorded terms
system heap	descriptors for atoms, functors, database items etc.
atom table	for each atom, a pointer to a descriptor
SDA queue	stepped destructive assignment records

Each section has the same general structure:



size is the total amount of memory allocated for the section. active_size is the amount of memory containing live data; this varies continually during program execution. Thus, the amount of memory available for additional live data is the difference between size and active_size. If this becomes less than margin, an overflow is said to occur. The section is in danger of exceeding its memory allocation. If active_size becomes greater than size, a hard overflow is said to occur. This is a disaster, because if another section lies beyond this one, part of it has been overwritten and thus lost, or if not, a reference has been made beyond the limit of virtual memory allocated so far (known as the break in UNIXese), a fatal mistake (known as a segmentation violation). To forestall disaster, Aquarius Prolog monitors the growth of each section during program execution. When a section overflows, it is expanded, any sections lying beyond it being shifted to make room for this, unless it is the heap, trail or firm heap. These three sections often contain large numbers of objects that, though within active_size, are no longer accessible to computation; these are called garbage. When the heap, trail or firm heap overflows, the manager first attempts to reclaim inaccessible objects; this is called garbage collection. Only if garbage collection fails to reclaim enough memory is the section expanded. One twist in this scheme deserves mention: when the heap overflows, not only the heap but the trail is garbage collected, assuming the trail has not been garbage collected since the last time the heap was garbage collected; this is because the trail is full of references to the heap, and whatever of these are garbage when the heap is garbage collected may reduce the efficiency of memory reclamation.

All this memory management is automatic, but occasionally you may wish to exercise some control over it. There are two main ways to do this. First, you may specify at program start-up many of the parameters according to which the Aquarius Prolog memory manager operates (section 19-2). Second, you may invoke the heap garbage collector explicitly using garbage_collect/0 (section 19-3).

19-2. Memory Management Parameters

For each section, three parameters are initialized at program start-up: initial_size, margin and expansion_factor. initial_size is simply the initial value of size, explained above along with margin. expansion_factor is multiplied by size to obtain the increment by which the section is expanded when this is necessary. For example, if expansion_factor is 0.25, size increases by one quarter whenever the section expands. For the heap, trail and firm heap, there is one other parameter: garbage_collection_margin. This is the minimum amount of memory that garbage collection must reclaim to prevent the section from being expanded when it overflows. Finally, all sections are affected by the overflow_check_frequency. This is approximately the number of times per second Aquarius Prolog checks to see whether any section has overflowed.

overflow_check_frequency is an integral number of checks per second, expansion_factor is a dimensionless integer or floating point number and initial_size, margin and garbage_collection_margin are integral numbers of 32-bit words. On all platforms supported so far, the smallest unit of memory with which Aquarius Prolog ordinarily works is the 32-bit word. In particular, this is the space occupied by a variable or atomic term.

By default, Aquarius Prolog uses what are thought to be reasonable values for all these parameters, so that usually you need not pay any attention to them.* But if you wish, you may specify any of them using command line arguments. All memory management arguments have the form

-aP*<memory management parameter acronym>*=*<number>*.

This form has been chosen to avoid conflicts with forms that are significant to user programs. Note that no spaces separate the equal sign from the other components. Arguments of this form are seen by the Aquarius Prolog memory manager only, that is, they are not returned to programs by unix/1 (section 18-1). Specifically, recognized memory management arguments are the following:

-aPHIS=*<number>*, -aPHM=*<number>* and -aPHEF=*<number>*
: specify heap initial_size, margin and expansion_factor, respectively.

-aPTIS=*<number>*, -aPTM=*<number>* and -aPTEF=*<number>*
: specify trail initial_size, margin and expansion_factor, respectively.

-aPSIS=*<number>*, -aPSM=*<number>* and -aPSEF=*<number>*
: specify stack initial_size, margin and expansion_factor, respectively.

-aPFHIS=*<number>*, -aPFHM=*<number>* and -aPFHEF=*<number>*
: specify firm heap initial_size, margin and expansion_factor, respectively.

-aPSHIS=*<number>*, -aPSHM=*<number>* and -aPSHEF=*<number>*
: specify system heap initial_size, margin and expansion_factor, respectively.

-aPATIS=*<number>*, -aPATM=*<number>* and -aPATEF=*<number>*
: specify atom table initial_size, margin and expansion_factor, respectively.

-aPSQIS=*<number>*, -aPSQM=*<number>* and -aPSQEF=*<number>*
: specify SDA queue initial_size, margin and expansion_factor, respectively.

-aPHGCM=*<number>*
: specifies heap garbage_collection_margin.

-aPTGCM=*<number>*
: specifies trail garbage_collection_margin.

-aPFHGCM=*<number>*
: specifies firm heap garbage_collection_margin.

-aPOCF=*<number>*
: specifies overflow_check_frequency.

---

* Examine *<installation directory>*/include/memory.h for the default values.

Command line argument processing is sensitive to prefixes such as -aPHIS=. If the string following the equal sign is not appropriate, this is reported (as a disaster), and execution terminates (with exit code −1).

Several limitations of this arrangement should be noted. First, for the firm heap, system heap or atom table, there is a minimum initial_size determined by the atoms and functors seen during compilation. Specifying an initial_size smaller than this has no effect. Thus, what one specifies is a lower bound, that is, the initial_size is at least whatever one specifies for any of these sections. Second, on all platforms supported so far, the maximum attainable overflow_check_frequency is about 100 checks per second. Specifying a rate larger than this is unlikely to have any effect.

In practice, you are likely to be most interested in adjusting the heap garbage_collection_margin and expansion_factor. If both are small, the heap grows slowly, and garbage collections are fast and frequent. This may be undesirable, because the total amount of time spent garbage collecting may be greater than it would be if the heap were larger, making garbage collections less frequent. This is especially true if the program periodically executes a deep failure, since this immediately reclaims a large fraction of the heap. On the other hand, when the heap grows large, as it quickly may if either garbage_collection_margin or expansion_factor is large, each garbage collection takes a long time, perhaps several seconds. This is unacceptable in interactive applications. It may be unacceptable even in noninteractive applications if the heap grows too large in relation to the physical memory available, because garbage collection may become dominated by paging.* For each program, you must determine by intuition and experiment what values of garbage_collection_margin and expansion_factor are best.

19-3. garbage_collect

garbage_collect invokes the heap garbage collector explicitly. The heap is not expanded even if the number of words reclaimed is less than the heap garbage collection margin, so garbage_collect/0 is effective only when a large fraction of the heap is garbage. Note that if the trail has not been garbage collected since the last time the heap was garbage collected, garbage_collect/0 causes the trail as well as the heap to be garbage collected.

garbage_collect/0 has no associated error conditions.

* Paging can be reduced and garbage collection performance generally improved using generational techniques such as have become common in Lisp and SmallTalk implementations. These may eventually be introduced into Aquarius Prolog.

19-4. statistics(*?Specifier*, *?Statistics*)

statistics/2 is discussed in this chapter because most of the data obtainable using it pertain to memory management. It broadly resembles statistics/2 of other Prolog implementations, but some aspects of it are particular to Aquarius Prolog.

statistics(*Specifier*, *Statistics*) unifies *Statistics* with a list of execution statistics associated with *Specifier*. If *Specifier* is not instantiated, backtracking instantiates it successively to each recognized specifier. Recognized specifiers and statistics associated with them are as follows:

runtime	[*Total*, *Increment*] *Total* is UNIX user time in milliseconds consumed by the process so far; *Increment* is UNIX user time in milliseconds consumed since the last statistics/[0,2] goal, if any, or by the process so far, if none. Garbage collection time is included.
memory	[*Size*, 0] *Size* is the size in bytes of the UNIX data segment. Ordinarily, this accounts for most of the memory used by Aquarius Prolog.
core	[*Size*, 0] core is a synonym for memory.
overflows	[*Time*] *Time* is UNIX user time in milliseconds consumed handling memory section overflows so far. Garbage collection time is included.
memory(*Section*)	[*Active*, *Free*] *Active* is active_size in bytes of memory section *Section*. *Free* is size minus active_size in bytes; this is memory that is allocated for *Section* but not in use at present, including margin.
garbage_collections(*Section*)	[*Count*, *Freed*, *Time*] *Count* is how many garbage collections of memory section *Section* have occurred so far, *Freed* is how many bytes they have reclaimed, and *Time* is how much UNIX user time in milliseconds they have consumed.
shifts(*Section*)	[*Count*, *Time*] *Count* is how many shifts of memory section *Section* have occurred so far, and *Time* is how much UNIX user time in milliseconds they have consumed.

Section should be one of heap, trail, stack, firm_heap, system_heap, atom_table or sda_queue.

If *Specifier* is instantiated to something other than a recognized specifier or if *Statistics* is instantiated to something other than a list, a type failure occurs.

19-5. statistics

statistics displays on the standard output stream an execution statistics summary. For example:

```
| ?- statistics.

memory section     total bytes    active      free   shifts   time spent
------------------ ----------- --------- ----------   ------   ----------
   system space
      firm heap         174928     23054    151874        0        0.000
      system heap        32768     10078     22690        0        0.000
      atom table          8192       898      7294        0        0.000
   global space
      heap             2101248       268   2100980        0        0.000
      trail             135168         0    135168        0        0.000
      SDA queue           2048         0      2048        0        0.000
   local space
      stack             131072        30    131042        0        0.000
memory (total)        20622264

memory section   garbage collections   bytes freed   time spent
--------------   -------------------   -----------   ----------
   firm heap                       0             0        0.000
   heap                            0             0        0.000
   trail                           0             0        0.000

0.050 seconds runtime, 0.000 seconds handling overflows

yes
| ?-
```

statistics/0 has no associated error conditions.

Appendix A. Limits

The following are simple limits of Aquarius Prolog:

- Structures and predicates cannot have arities greater than 255 (2^8 - 1).
- Integers should be in the range -268435456 (-2^28) to 268435455 (2^28 - 1) inclusive. There is no range checking (integer arithmetic is thus effectively modulo 2^29).
- The range of floating point number is approximately 2.2e-44 to 3.4e+38. Floats have an 8-bit binary exponent and a 20 (+1 sign)-bit mantissa, a precision of no more than six decimal digits. Floating point overflow or underflow behavior is platform-dependent.

There is no limit on the number of characters an atom name may have, and there is no limit on how many predicates or clauses thereof may be defined, using either the Aquarius Prolog compiler or consult/1 and company (section 3-1).

Long-running programs may encounter two other limits of Aquarius Prolog:

- The maximum number of atoms a program may use is 1048576 (2^20). This includes atoms occuring in the run-time system (no more than 1000), atoms occurring in the program text, atoms read in by read/[1,2] (section 10-1) and atoms created by atom_chars/2 (section 7-23) and name/2 (section 7-25). The atom that is the name of the principal functor of a compound term is always created if not already present when the compound term is created. Atoms once present remain present always; there is no ‘‘garbage collection’’ of atoms.
- Virtual copy semantics for the databases (section 12-1) are implemented using a ‘‘clock’’ that starts at zero and is incremented whenever a clause is asserted or retracted or an internal database record is recorded or erased. The maximum value of the ‘‘clock’’ is 268435455 (2^28 - 1).

There is no checking for these two limits. If either is exceeded, chaos will ensue.

Aquarius Prolog imposes no limit on how many input/output streams may be open at one time, but some operating systems do. In practice, this is very rarely a problem.

Appendix C. Prospective Improvements

Besides corrections of deficiencies and mistakes in version 1.0, we hope (though we do not promise) that subsequent versions of Aquarius Prolog will include at least some of the following:

- A substantially faster compiler. The present compiler was written to generate code that runs fast, not to run fast itself. We suspect that different choices of data structures for some purposes, for example, could speed it up considerably without reducing the quality of the code it generates.

- A compilation scheme that permits separate processing of files up to linking.

- More efficient garbage collectors using and generational techniques.

- Debugging tools. It would not be difficult to add ''procedure box''-style debugging for interpreted code. More difficult but more worthwhile would be a symbolic debugger for compiled code.

- A module system. We have already implemented a prototype.

- A scheme for calling C from Prolog. The run-time system already incorporates a scheme for internal use that is a large start toward this.

Table of Contents