Release Notes for Quintus Prolog Release 2.0 FCS

1. Introduction

The main new features in Release 2.0 are

1. A module system.
2. Garbage collection.
3. Indexing of interpreted procedures.
4. An interprocess communication package which allows Prolog to be called from C.

The module system allows a large program to be partitioned into separate name spaces, thus making it easier for a group of programmers to work together on a large project, and also making it feasible to regard library predicates as extensions to the set of built-in predicates. The module system is fully described in the new manual which is available on-line. For example, having started up Prolog under Emacs, you can type help(modules) to find out about modules.

For the most part, you won't need to know much about the incremental garbage collector, except that it is there. For an explanation of what it does, as well as information on how to monitor its behavior and tune it if necessary, see chapter 6 of the Quintus Prolog System-Dependent Features Manual. (This too, is available on-line: manual(sysdep-6)).

Indexing of interpreted procedures is not covered in the current version of the manual, and is therefore explained below.

The interprocess communication package is in a set of Prolog and C files which can be used to build a Prolog "goal server" which accepts goals from another process and returns solutions of that goal to that process. The calling process may be written in C or in Prolog; a more general interface is provided for the Prolog-to-Prolog case. The package can be found in the directory called IPC in the Quintus Prolog installation directory. In addition to the sources, documentation is provided in a file called IPC.doc. The same documentation is also provided in PostScript form in IPC.ps in case you have access to a suitable printer.

Section 2, below, explains the indexing of interpreted code and section 3 explains the related issue of the changed semantics of dynamic code. Section 4 describes all the other changes in the Prolog system. Section 5 describes changes in the library; and finally, section 6 gives some performance measurements comparing this release with Release 1.6.

See also the file called Incompatibility_notes in the Quintus Prolog installation directory. This file describes a couple of difficulties which may arise in porting programs from earlier releases of Quintus Prolog to Release 2.0.

2. Interpreted Code Is Now Indexed

Many applications involve run-time changes to dynamic Prolog procedures, using such predicates as **assert**/1 and **retract**/1. All dynamic code, as well as static code loaded by **consult**/1, is implemented in Quintus Prolog via an interpreter.

Prior to Quintus Prolog Release 2.0, the clauses of an interpreted procedure were linked together sequentially. The time required to access clauses of interpreted procedures was thus proportional to the number of clauses in the procedure. In Quintus Prolog Release 2.0, all interpreted code is indexed. Indexing provides hash-table access to interpreted clauses. Indexes are maintained automatically by the built-in predicates manipulating the Prolog database (e.g., **assert**/1, **retract**/1 and **consult**/1). Indexing is on the primary functor of the first argument of each clause, as is indexing on compiled static code.

In addition to providing fast clause access, the new semantics for interpreted code allow indexing to greatly improve determinacy detection. This can, in turn, provide considerable memory savings by means of tail recursion optimization.

3. New Semantics for Data Base Predicates

The semantics of those predicates which modify the Prolog data base (e.g., **assert**/1 and **retract**/1) have changed. The new semantics are more intuitive, improve memory performance, and increase the effectiveness of the indexing of interpreted code. The remainder of this section discusses the motivation for the change, and contrasts the behavior of Quintus Prolog Release 2.0 with that of previous releases.

In earlier releases of Quintus Prolog, changes to the Prolog data base became globally visible upon the success of the built-in predicate modifying the data base. An unsettling consequence was that the definition of a procedure could change while it was being run. This could lead to code that was difficult to understand. Furthermore, memory performance of an interpreter implementing these semantics was poor. Worse yet, the semantics rendered ineffective the added determinacy detection available through indexing.

In Release 2.0, the definition of an interpreted procedure that is to be visible to a call is effectively frozen when the call is made. A procedure always contains, as far as a call to it is concerned, exactly the clauses it contained when the call was made.

For the purposes of explanation, a call to an interpreted procedure makes a virtual copy of the procedure, then runs the copy rather than the original procedure. Any changes to the procedure made by the call are immediately reflected in the Prolog data base, but not in the copy being run. Thus, changes to a running procedure will not be visible on backtracking. A subsequent call, however, makes and runs a virtual copy of the modified Prolog data base. Any changes to the procedure that were made by an earlier call will be visible to this call.

In addition to being more intuitive and easy to understand, the new semantics allow interpreted code to execute with the same determinacy detection (and excellent memory performance) as static compiled code. It is likely that interpreted code developed on previous Quintus Prolog releases will run under Quintus Prolog Release 2.0 without any noticeable differences. However, the new semantics can be demonstrated when modifying the Prolog data base.

3.1. Example: assertz/1

Consider the procedure foo/0 defined by

```
:- dynamic foo/0.
foo :- assertz(foo), fail.
```

Each call to foo/0 asserts a new last clause for foo/0. After the Nth call to foo/0 there will be N+1 clauses for foo/0. When the first call to foo/0 is made, the procedure has exactly one clause. Under earlier releases of Quintus Prolog, the change to the Prolog data base made by **assertz**/1 took effect as soon as the call to **assertz**/1 succeeded. Suddenly the procedure had two clauses. **fail**/0 then forced the call to foo/0 to backtrack. The backtracking call found the newly asserted clause and succeeded. So in Quintus Prolog Release 1.6, we have

```
| ?- compile(user).
| :- dynamic foo/0.
| foo :- assertz(foo), fail.
|
[user compiled 0.117 sec 188 bytes]

yes
| ?- foo.  % The asserted clause succeeds!

yes
| ?-
```

When foo/0 is first called under Quintus Prolog Release 2.0, a virtual copy of the procedure is made, effectively freezing the definition of foo/0 for that call. At the time of the call foo/0 has exactly one clause. Thus, when **fail**/0 forces backtracking, the call to foo/0 simply fails: it finds no alternatives. Under Quintus Prolog Release 2.0,

```
| ?- compile(user).
| :- dynamic foo/0.
| foo :- assertz(foo), fail.
|
[user compiled 0.100 sec 204 bytes]

yes
| ?- foo.  % The asserted clause is not found

no
| ?- foo.  % A later call does find it, however

yes
| ?-
```

Even though the virtual copy of foo/0 being run by the first call is not changed by the assertion, the Prolog data base is. Thus, when a second call to foo/0 is made, the virtual copy for that call contains two clauses. The first clause fails, but on backtracking the second clause is found and the call succeeds.

3.2. Example: retract/1

Under Quintus Prolog Release 1.6, retracted clauses were made invisible immediately:

```
| ?- assert(p(1)), assert(p(2)), assert(p(3)).

yes
| ?- p(N), write(N), nl, retract(p(2)), retract(p(3)), fail.
1

no
| ?-
```

We first assert three clauses into the (previously empty) procedure p/1, then make a query. The call to p/1 succeeds, first binding N to 1 and writing out N. The two retractions then take place, the retracted clauses becoming invisible before **fail**/0 was called. On backtracking the call to p/1 finds no further clauses, so the call to p/1 fails. Thus, the query fails.

Under Quintus Prolog Release 2.0, the side effects caused by **retract**/1 go unrecognized by the backtracking call, but are seen by a subsequent call to p/1:

```
| ?- assert(p(1)), assert(p(2)), assert(p(3)).

yes
| ?- p(N), write(N), nl, retract(p(2)), retract(p(3)), fail.
1
2
3

no
| ?- p(N), write(N), fail.
1
no
| ?-
```

At the first call to p/1, the procedure has three clauses. These remain visible throughout execution of the call to p/1. Thus, when backtracking is forced by **fail**/0, N is rebound to 2 and written. The retraction is again attempted, causing backtracking into p/1. N is rebound to 3 and written out. The call to **retract**/1 fails. There are no more clauses in p/1, so the query finally fails. A subsequent call to p/1, made after the retractions, sees only one clause.

3.3. Example: abolish/2

An example using **abolish**/2 exhibits behavior similar to that of **retract**/1. Under earlier releases of Quintus Prolog:

```
| ?- compile(user).
| :- dynamic q/1.
| q(1).
| q(2).
| q(3).
| ^D
[user compiled 0.117 sec 260 bytes]

yes
| ?- q(N), write(N), nl, abolish(q,1), fail.
1

no
| ?- q(N).

[Warning: The procedure q/1 is undefined]
   (1) 0 Call: q(_432) ?
   (1) 0 Fail: q(_432) ?

no
| ?-
```

Under Quintus Prolog Release 2.0:

```
| ?- compile(user).
| :- dynamic q/1.
| q(1).
| q(2).
| q(3).
| ^D
[user compiled 0.117 sec 260 bytes]

yes
| ?- q(N), write(N), nl, abolish(q,1), fail.
1
2
3

no
| ?- q(N).

[Warning: The procedure q/1 is undefined]
   (1) 0 Call: q(_432) ?
   (1) 0 Fail: q(_432) ?

no
| ?-
```

3.4. A warning concerning retract/1

A warning may be in order against a common mistake in the use of **retract**/1. This applies equally to users of earlier releases of Quintus Prolog as well as Release 2.0.

retract/1 is a non-determinate predicate. Thus, we can use

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

to retract all clauses for **foo**/1. A non-determinate predicate in Quintus Prolog (including user-defined ones) uses a choice point, a data structure kept on an internal stack, to implement backtracking. In a simple model, a choice point is created for each call to a non-determinate predicate, and is deleted on determinate success or failure of that call, when backtracking is no longer possible. (In fact, Quintus Prolog employs advanced determinacy detection to recognize contexts in which choice points required by the simple model can be avoided, or are no longer needed.)

The Prolog **cut** works by removing choice points, disabling the potential backtracking they represented. A choice point can thus be viewed as an "outstanding call", and a **cut** as deleting outstanding calls.

To avoid leaving inconsistencies between the Prolog data base and outstanding calls, a retracted clause is reclaimed only when the system determines that there are no choice points on the stack that may try to backtrack into it. Thus, the existence of a single choice point on the stack can disable reclamation of retracted clauses for the procedure whose call created the choice point. Space is recovered only when the choice point is deleted.

Often **retract**/1 is used determinately, e.g., to retract a single clause as in

```
| ?- <do some stuff>
     retract(Clause),
    <do more stuff without backtracking>.
```

No backtracking by **retract**/1 is intended. Nonetheless, if Clause may match more than one clause in its procedure, a choice point will be created by **retract**/1. While executing <do more stuff without backtracking>, that choice point will remain on the stack, possibly "freezing" the retracted Clause. (Such careless choice points can also disable tail recursion optimization.) This can be avoided by simply cutting away the choice point with an explicit **cut** or a local cut (**->**). If not cut away, the choice point can also lead to runaway retraction on the unexpected failure of a subsequent goal. Thus, in the previous example, it is preferable to write either

```
| ?- <do some stuff>
     retract(Clause),
     !,
    <do more stuff without backtracking>.
```

or

```
| ?- <do some stuff>
    ( retract(Clause) -> true ),
    <do more stuff without backtracking>.
```

This will reduce stack size and allow the earliest possible reclamation of retracted clauses.

4. Other Changes in Release 2.0

4.1. Changed Interpretation of Relative File Names

When a file being loaded into Prolog contains an embedded command to load another file, a relative file name in that command is interpreted with reference to the directory which contains the first file. For example, if the file /usr/fred/test.pl contains the following commands

```
:- ensure_loaded(foo).
:- compile('../whatsit').
:- load_foreign_files('test.o',[]).
```

then the files to be loaded would be /usr/fred/foo.pl (or /usr/fred/foo), /usr/whatsit.pl (or /usr/whatsit) and /usr/fred/test.o. This is different from previous releases of Quintus Prolog in which the current working directory of the Prolog system would have been used.

The advantage of the new scheme is that you can more conveniently have a file which loads up several other files. For example, you can have a file called mainfile.pl containing

```
:- compile([file1, file2, file3]).
```

and provided that you keep all of these files in the same directory as mainfile.pl, you can compile them all, no matter what your current working directory is, by giving compile/1 a file specification for mainfile.pl.

4.2. Built-in Operator Properties Can Be Overridden

Prior to Quintus Prolog Release 2.0, the properties of predefined "built-in operators" could not be overridden. This restriction has been lifted in Release 2.0: system operators are indistinguishable from user-supplied ones except by their predefinition. Declarations for operators built into the Prolog system can be found in Appendix II of the *Quintus Prolog Reference Manual.*

Note that ':'/2 is now a built-in infix operator, as well as being a built-in predicate — *Module*:*Goal* means call *Goal* in module *Module*. Since built-in operator properties can be overridden, the operator properties will not affect any programs which use ':'/2 in data. However, since ':'/2 is a built-in predicate, it is not possible in this release to define clauses for ':'/2.

4.3. Memory Management Improvements

In addition to the garbage collector, Release 2.0 incorporates a number of other improvements in memory management. The two most important of these are described below.

4.3.1. Space for Abolished Compiled Code Is Reclaimed

Space for abolished static compiled code (including compiled code which is abolished during recompilation) is now reclaimed on return to the top level. (Space is not reclaimed on return to a break level.) This can reduce memory use — especially during development, when recompilations might be frequent. In earlier releases of Quintus Prolog, space for abolished static compiled code was not reclaimed.

4.3.2. The Code Space Congealer

In Quintus Prolog, the code space is the area which contains all compiled and consulted code as well as space allocated by calls to malloc(3) from users' C code. This space is managed by a system of multiple free lists. In previous releases, excessive fragmentation sometimes arose, particularly when the program had the characteristic that it was repeatedly allocating and then freeing larger and larger blocks of memory. This problem has been solved in Release 2.0 by the provision of a congealing routine which, from time to time, searches out adjacent free blocks and "congeals" them into a single block.

4.4. New Built-in Predicates

These are all explained in more detail in the Reference Manual. This is a summary.

abolish(*+Predicates*)
: This is like **abolish**(*Name,Arity*) except that the predicate to be abolished is specified as a single argument in the form *Name/Arity*. You can also specify a list of predicates to be abolished.

retractall(*+Head*) Erases every clause in the data base whose head matches *Head*. The argument *Head* must be instantiated to a term corresponding to a dynamic predicate.

retractall/1 used to be defined in the library package library(update).

retractall/1 is useful for erasing all the clauses of a dynamic predicate without forgetting that it is dynamic; **abolish**/1 will indeed erase all the clauses, but it will also forget absolutely everything about the predicate. **retractall**/1 just erases the clauses.

As **retractall**/1 erases all the clauses whose heads match *Head*, it is determinate. If there are no such clauses, it succeeds trivially.

portray_clause(*+Clause*)
: Writes the clause *Clause* to the current output stream. **portray_clause**/1 is the operation used by **listing**/0 and **listing**/1. The clause is written to the current output stream in exactly the format in which **listing**/1 would have written it, including a final period and newline.

If you want to print a clause, this is almost certainly the command you want. By design, none of the other term output commands puts a period after the written term. If you are writing a file of facts to be loaded by **consult**/1 or **compile**/1, use **portray_clause**/1, as it tries to ensure that the clauses it writes out can be read in again as clauses.

atom_chars(*?Atom,?Chars*)
: *Chars* is the list of ASCII character codes comprising the printed representation of the atom *Atom*. This is like **name**/2 except that the first argument may not be a number, and in the cases such as

```
| ?- atom_chars(X,"1.2").

X = '1.2' ,
```

an atom rather than a number is returned.

number_chars(*?Number,?Chars*)
: *Chars* is the list of ASCII character codes comprising the printed representation of the number *Number*. This is like **name**/2 except that the first argument may not be an atom, and the second argument may not be bound to a list of characters which do not represent a number.

copy_term(*+Term,-Copy*)
: This is a meta-logical predicate which makes a new *Copy* of its *Term* argument in which all variables have been replaced by new variables which occur nowhere else.

prolog_flag(*?FlagName,?Value*)
: This allows you to find the current value of a flag. It is like **prolog_flag**(*+FlagName*, Value, Value) except that with this predicate it is permissible to give a variable as the *FlagName* argument and have it backtrack through all the flags.

stream_position(*?Stream,?Position*)
: is true when *Stream* is a stream object representing an open stream, *Pos* is a stream position object, and the current position of *Stream* is *Pos*.

This operation makes sense on any stream at all: streams which are connected to disk files, streams which are connected to the terminal, and even streams defined using QP_make_stream().

garbage_collect This predicate forces an immediate garbage collection. There should normally be no need to call **garbage_collect**/0, because the garbage collector is automatically invoked whenever the system would otherwise require more space.

Module System predicates

The following new built-in predicates concern the module system. They are fully described in the manual chapter on modules.

use_module (*+FileSpecList*)
use_module (*+FileSpec, +ImportList*
module (*+TypeInModule*)
current_module (*?ModuleName*)
current_module (*?ModuleName, ?SourceFileName*)
: (*+Module, +Goal*) **or** *+Module*:*+Goal*

The following two functors are not built-in predicates, but their names are reserved since they are used as declarations.

module (*ModuleName, PublicPredList*)
meta_predicate (*Term*)

4.5. Changes to System Limitations

1. An internal limitation on the number of clauses in an index of a static compiled procedure has been removed. In earlier releases of Quintus Prolog, an index could overflow on as few as 2^13 clauses. (To reach this limit, a compiled procedure would need at least 2^13 clauses with nonvariable first arguments.) There is also no limit to the number of clauses in an index for any interpreted procedure.
2. Previously there was an internal limitation (see the *Quintus Prolog System-Dependent Features Manual*, Appendix II) that compiled clauses could contain no more than 255 temporary variables. The limit has been raised to 512 temporary variables.

5. Changes to the Library

The most important change to the library is that several files are now supported. Many new files have been added. Some old files have been withdrawn and their contents moved into the system or distributed among other files.

5.1. On-line documentation

The library directory contains two files which describe the contents of the directory. They are called Contents and Index.

The Index file contains one line for each exported predicate in the library. The predicates are listed in standard order, ignoring module. A typical entry looks like this:

```
list_to_binary/4    flatten    ./flatten.pl
```

This means that there is a predicate called list_to_binary/4 in the library, that it lives in a module called 'flatten', and that the file which contains it is 'flatten.pl' in the same directory as the Index.

If you have set up an "environment variable" QL holding the name of the Quintus library directory, you could ask "what predicates are there to deal with files?" by issuing the UNIX command

```
% egrep files $QL/Index
```

The Contents file is organised by library files rather than by predicates. A typical entry in this file is a block of lines like this:

```
basics  + is supported
basics  % the basic list processing predicates
basics  - ./basics.pl
basics  : append/3
basics  : member/2
basics  : memberchk/2
basics  : nonmember/2
```

The first line means that library(basics) is part of the supported library. The second line is a short description of the contents. The third line says which file contains library(basics), in this case it is 'basics.pl' in the same directory as Contents. The remaining lines list the predicates exported by library(basics). You could obtain this information by issuing the UNIX command

```
% egrep '^basics' $QL/Contents
```

These files are provided as a convenience, and do not have the same authority as the printed manual.

5.2. For C Programmers

The library includes a header file, "quintus.h" which provides external declarations for all the variables and functions which the Quintus Prolog system exports to user foreign code. Do not look for these functions in the library, they are in the system. This file is supported.

The Prolog Lint checker automatically includes this header file into the scratch file it generates.

5.3. Supported Library Files

All the files described here have fuller descriptions in the Quintus Prolog Library Manual.

arg.pl — library(arg) defines several generalisations of **arg**/3. The predicates args/3, args0/3, and path_arg/3 are new in this release. project/3 was moved here from the defunct library(project).

ask.pl — library(ask) is for asking questions of the user. Most of the commands take a prompt and a specification of what sort of answer is wanted and read a one-character answer from the terminal. ask_chars/4 is new in this release.

basics.pl — library(basics) is the basic list-processing routines. You will almost always want these.

changearg.pl — The predicates change_arg/[4,5] and swap_args/[4,5,6] have been split out of library(arg), and the two new predicates change_path_arg/[4,5] have been added. The new package is library(change_arg). There is a new matching path_arg/3 predicate in library(arg). With the new predicates, you can find a path to a subterm and then replace it; for example,

```
if Term = ((a+b)*(c+d))/((e+f)*(g+h)), then
    path_arg(Path, Term, c)
binds Path = [1,2,1], and then
    change_path_arg(Path, Term, NewTerm, z)
binds NewTerm = ((a+b)*(z+d))/((e+f)*(g+h)).
```

continued.pl — library(continued) is a new package for reading continued lines (such as TERMCAP entries) from files.

ctypes.pl — library(ctypes) provides a kit of predicates for classifying characters. For the most part, these predicates are very closely modelled on the character classification macros provided in the C header file <ctype.h>. Check "isalpha" in your C manual. These

predicates (though not the code in any particular version) are portable between ASCII and EBCDIC versions of Quintus Prolog. The predicates is_csym/1, is_csymf/1, and is_digit/3 are new in this release.

directory.{pl,c} library(directory) is specific to UNIX, and will not be available for VMS or for Xerox Quintus Prolog.

The predicates

```
file_member_of_directory(
            [Directory, [Pattern, ]]Name, Full)
directory_member_of_directory(
            [Directory, [Pattern, ]]Name, Full)
```

enumerate the file (directory) *Names* in the given *Directory* [default = '.'] which match the given *Pattern* [default = '*']. The *Name* argument is unified with a simple file name (that is, just the name and extension), and the *Full* argument is unified with a full path name including the directory prefix. The predicates

```
file_property(File, Property[, Value])
directory_property(Directory, Property[, Value])
```

check whether a *File* or *Directory* has a certain boolean property (such as being readable), or find the value of a *File* or *Directory*'s *Property* (such as its owner).

See the Library Manual to find out which properties can be tested.

files.{pl,c} library(files) provides commands for renaming, deleting, and opening files, for testing whether a file could be opened, for enumerating DEC-10-compatible streams, and for closing all open streams. The predicates current_dec10_stream/2 and rename/2 are new in this release.

lineio.pl library(lineio) provides commands for reading lines from files and writing lines to files as lists of character codes. The predicate get_line/3 is new in this release.

lists.pl library(lists) is a collection of list-processing utilities which are not used quite as often as the ubiquitous library(basics). The predicates append/5 and same_length/3 are new in this release. numlist/3 has moved into library(between) {not supported yet}. Several other predicates in this file have been generalised.

math.{pl,c} library(math) is an example of the use of the foreign code interface. It provides an interface to the C "math" library. The predicates abs/2, max/3, and min/3 are new in this release.

not.pl library(not) provides some sound, and some unsound, negation predicates. If you are worried about the "unsoundness" of '\+'/1, not/1 may be what you want. The predicate once/1 is new in this release.

ordsets.pl library(ordsets) is a collection of operations on sets represented as lists in standard order. With this representation, set operations can be more efficient than with the unordered list representation used by library(sets). The predicates ord_intersection/2, ord_intersection/3, and ord_union/2 are new in this release.

prompt.pl library(prompt) provides prompted input from the terminal.

readconst.pl library(read_const) provides Pascal-style input from files or from the terminal. The command read_constant(X) acts much like read(X) would in Pascal. That is, it skips layout in the current input stream, reads a "token", and unifies that with X. "..." is read as a list of character codes, '...' as an atom. Other input is terminated by a layout character. %-comments are allowed, and behave like layout characters. The command read_constants([X1,...,Xn]) acts much like read(X1, ..., Xn) would in Pascal. The commands prompted_constant/2 and prompted_constants/2, which read from the terminal, act like readln(). See also library(lineio).

readin.pl library(read_in) is for reading sentences.

readsent.pl library(read_sent) is another package for reading sentences.

samefunctor.pl — library(same_functor) is a new package defining some predicates which test whether two terms have the same functor. These operations make it easier to write "reversible" meta-logical predicates.

sets.pl — library(sets) is a collection of operations on sets represented as lists with no duplicate elements but in no particular order. If you can ensure that the lists are in standard order (as the result of **setof**/3 or **sort**/2 is), you will find library(ordsets) more efficient. The predicates intersection/2, intersection/3, and union/2 are new in this release.

strings.{pl,c} — library(strings) is a collection of operations on character sequences represented as atoms (or, in Xerox Quintus Prolog, as Lisp strings). The predicates atom_chars/2 and number_chars/2 which used to be defined here are now part of the Quintus Prolog system. The predicates compare_strings/3, compare_strings/4, concat_atom/2, concat_chars/2, midstring/[3,4,5,6], nth_char/3, span_left/[3,4,5], span_right/3,4,5], span_trim/[2,3,5], string_length/2, string_search/3, subchars/[4,5], and substring/5 are new in this release.

subsumes.pl — library(subsumes) provides predicates for testing whether one term subsumes another. copy_term/2 is now part of the Quintus Prolog system.

unify.pl — library(unify) implements sound unification (that is, unification with the occurs check).

5.4. The Unsupported Library

Many new files have been added to the unsupported library. Additions and improvements have been made to most of the others, and a few files have been withdrawn and their contents distributed among other files.

The withdrawn files are

contains.pl — renamed to occurs.pl because the argument order of its predicates was changed. library(occurs) is supported.

portray_string.pl — renamed to printchars.pl

project.pl — contents distributed between library(arg) and library(lists), both of which are supported.

streampos.pl — this used to define stream_position/2, which is now a built-in predicate.

The new and changed files are

aggregate.pl — library(aggregate) defines aggregate/3, an operation similar to bagof/3 which lets you calculate sums. For example, given a table pupil(Name,Class,Age), to calculate the average age of the pupils in each class, one would write

```
| ?- aggregate( sum(Age)/sum(1),
                 Name^pupil(Class,Name,Age),
                 Expr),
     call(Average_Age is Expr).
```

aropen.{pl,c} — library(aropen) lets you open a member of a UNIX archive file (see ar(1) in the UNIX manual) without having to extract the member. You cannot compile or consult such a file, but you can read from it. This may be useful as an example of defining Prolog streams from C.

arrays.pl — library(arrays) provides constant-time access and update to arrays. It involves a fairly unpleasant hack. You would be better off using library(logarr) or library(trees).

assoc.pl — library(assoc) was present in 1.6. The predicates get_next_assoc/4, get_prev_assoc/4, max_assoc/3, and min_assoc/3 are new in this release.

between.pl — library(between) was present in 1.6. The predicate gen_arg/3 has moved to library(arg) which is supported. The predicate numlist/3 moved here from library(lists).

big_text.{pl,c} library(big_text) defines a 'big_text' data type and several operations on it. The point of this module is that when writing an interactive program you often want to display to (or acquire from) the user large amounts of text. It would be inadvisable (although possible) to store the text in Prolog's data base. With this package you can store text in a file, copy text to a stream, acquire new text from a stream, and/or have Emacs edit a big text file.

See the file big_text.txt in the library area for more details.

call.pl library(call) was present in 1.6. The predicates call/6 and call/7 are new in this release. It works with the module system. This is actually a very important library file.

caseconv.pl library(caseconv) is mainly intended as an example of the use of library(ctypes). (Note that the order of the arguments to the 'ctypes' predicate to_upper changed in Release 1.6.) Here you'll find predicates to test whether text is in all lowercase, all uppercase, or mixed case, and to convert from one case to another.

charsio.{pl,c} library(charsio) lets you open a list of character codes for input as a Prolog stream and, having written to a Prolog stream, collect the output as a list of character codes. There are three things you can do with library(charsio):

1. You can open an input stream reading from a (ground) list of characters. This is the predicate chars_to_stream.
2. You can run a particular goal with input coming from a (ground) list of characters. The predicates with_input_from_chars/[2,3] do this.
3. You can run a particular goal with output going to a list of characters (the unification is done after the goal has finished). The predicates with_output_to_chars/[2,3] do this.

crypt.{pl,c} and encrypt.c

library(crypt) defines two operations similar to **open**/3: crypt_open(*+FileName[, +Password], +Mode, -Stream*)

If you do not supply a *Password*, crypt_open/3 will prompt you for it. Note that the password will be echoed. If there is demand, this can be changed. The *Stream* will be clear text as far as Prolog is concerned, yet encrypted as far as the file system is concerned.

encrypt.c is a stand-alone program (which is designed to have its object code linked to 3 names: encrypt, decrypt, and recrypt), and can be used to read and write files using this encryption method.

This encryption method was designed, and the code was published, in Edinburgh, so it is available outside the USA.

decons.pl library(decons) provides a set of routines for recognising and building Prolog control structures. It is derived from the Dec-10 Prolog library file of the same name. The only predicate which is likely to be useful to you is prolog_clause*(Clause, Head, Body).*

environ.pl library(environ) provides access to the UNIX "environment variables". (See sh(1), csh(1), getenv(3), and environ(5) in the UNIX manual.) environ(?Varname, ?Value) is a genuine relation. Note that if you include this file in a saved state, the values of environment variables are those current when the saved state was run, not when it was saved. There is also an argv/1 in this file, which is superseded by the supported feature unix(argv(_)).

foreach.pl library(foreach) defines two iteration forms.

```
forall(Generator, Test)
```

is the standard double-negation "there is no proof of *Generator* for which *Test* is not provable", coded as "\+ (*Generator*, \+ *Test*)."

```
foreach(Generator, Test)
```

works in two phases: first each provable instance of *Generator* is found, then each corresponding instance of *Test* is collected in a conjunction, and finally the conjunction is executed.

If, by the time a *Test* is called, it is always ground — apart from explicitly existentially quantified variables — the two forms of iteration are equivalent, and forall/2 is cheaper. But if you want *Test* to bind some variables, you must use foreach/2 .

A foreach(*Variables, Generator, Test*) predicate is planned but was not ready in time for the beta release.

freevars.pl — This is an internal support package that was split out when modules were added to Quintus Prolog. Users will probably have no use for its predicates.

fromonto.pl — library(fromonto) defines some "pretty" operators for input/output redirection. Examples:

```
| ?- (repeat, read(X), process(X)) from_file 'fred.dat'.

| ?- read(X) from_chars "example. ".
X = example

| ?- write(273.4000) onto_chars X.
X = "273.4"
```

BEWARE: you will have to correct this file. The directive

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

was accidentally omitted, and you will have to supply it.

knuth_b_1.pl — library(knuth_b_1) is a table of constants taken from appendix B1 of D.E.Knuth's "The Art of Computer Programming", Volume 1. The point is not to provide the constants — you could have calculated them yourselves easily enough — but to illustrate the recommended way of building such constants into your programs.

logarr.pl — library(logarr) is an implementation of "arrays" as 4-way trees. See also library(trees).

long.pl — This is a rational arithmetic package. Users of the DEC-10 library file LONG.PL will miss the "trig" functions and will welcome the fact that exponentiation works!

rational*(N)* recognises arbitrary precision rational numbers; this includes integers, 'infinity', 'neginfinity', & 'undefined'. whole*(N)* recognises arbitrary precision integers. eval*(Expr, Result)* evaluates an expression using arbitrary precision rational arithmetic; it does not accept floats at all. {eq,ge,gt,le,lt,ne}/2 are infix predicates like < that compare rationals (or integers, not expressions). succ/2, plus/3, and times/3 are relational forms of arithmetic which work on rational numbers (not floats). To have rational numbers printed nicely, put the command

```
:- assert((portray(X) :- portray_number(X)))
```

in your code. See long.doc and the comments in the long.pl

maplist.pl — library(maplist) was present in release 1.6. It is built on top of library(call), and provides a collection of meta-predicates for applying predicates to elements of lists. The predicate include/3 is new in this release.

maps.pl — library(maps) was present in release 1.6. It implements functions over finite domains, which functions are represented by an explict data structure. The predicate map_intersection/3 is new in this release.

menu.pl — library(menu) illustrates how to drive the Emacs interface from Prolog. The sample application involves choosing items from a menu. See also menuexample.pl in the demo subdirectory of the installation directory.

multil.pl — has been converted from the DEC-10 Prolog library and debugged, but we don't recommend that you use it.

note.pl — The built-in predicates and commands pertaining to the "recorded" (or "internal" data base) have an argument called the "key". All that matters about this key is its principal functor. That is, fred(a,b) and fred(97,46) are regarded as the same key. Library(note) defines a complete set of storing, fetching, and deleting commands where the "key" is a ground term all of which is significant, using the existing recorded data base. Note that this package is no better indexed than the existing recorded data base.

ordered.pl — library(ordered) is a collection of predicates for doing things with a list and an ordering predicate. The predicates select_max/[3,4] and select_min/[3,4] are new in this release. See also library(ordprefix), library(ordsets), and library(samsort).

ordprefix.pl — library(ordprefix) is for extracting initial runs from lists, perhaps with a user-supplied ordering predicate. See also library(ordered).

pptree.pl — This file defines pretty-printers for (parse) trees represented in the form

```
<tree> --> <node label>/[<son>,...<son>]
         |  <leaf label>          -- anything else
```

Two forms of output are provided: a human-readable form and a Prolog term form for reading back into Prolog.

```
pp_tree(+Tree)
```

prints the version intended for human consumption, and

```
pp_term(+Tree)
```

prints the Prolog-readable version. There is a new command ps_tree/1 which prints trees represented in the form

```
<tree> --> <node label>(<son>,...,<son>)
         |  <leaf>                -- constants
```

The output of ps_tree/1 is readable by Prolog and people both. You may find it useful for other things than parse trees.

printchars.pl — library(print_chars) was present in release 1.6. The name of the file has been changed several times because of restrictions in VMS file names. (These restrictions no longer exist.) This is not a module, and it would be pointless to load it into any module but 'user'. It defines portray/1 so that lists of character codes are written by print/1, by the top level, and by the debugger, between double quotes.

```
| ?- X = "fred".
X = [102,114,101,100]

| ?- ensure_loaded(library(print_chars)),
|    X = "fred".
X = "fred"
```

qsort.pl — NOTE: quicksort is not a good sorting method for a language like Prolog. If you want a good sorting method, see library(samsort), which was described briefly in a recent newsletter. This is what a good Prolog quicksort really looks like. Read it carefully to see why this version is stable and the usual version isn't!

random.{pl,c} — library(random) provides a random number generator and several handy interface routines. This was present in release 1.6, but the heart of it was rewritten in C for speed. The random number generators in UNIX 4.2, System V release 2, VMS, and Interlisp are all different. It is useful to have a random number generator which will give the same results in all versions of Quintus Prolog, and this is the one. The random number generator was recommended in a Statistics journal, so should be good, but if anyone knows or can show that it is not, please tell us. You might care to consult the book "Numerical Recipes", by Press, Flannery, Teukolsky, and Vetterling, published by Cambridge University Press.

retract.pl — The names and arguments of the built-in predicates for asserting, retracting, and

accessing clauses of dynamic predicates are somewhat haphazard. This file adds more, so that there are two complete families of commands for clauses and a complete family for the recorded data base. The built-in predicate **retract**/1 will backtrack through a predicate, expunging each matching clause until the caller is satisfied. This is not a bug. That is the way **retract**/1 is supposed to work. But it has long been felt to be a problem. library(retract) defines, among many other commands, retract_first/1, which is identical to **retract**/1 except that it expunges only the first matching clause, and if asked for another solution, fails.

samsort.pl — library(samsort) provides a stable sorting routine which exploits existing order, both ascending and descending. (It is a generalisation of the natural merge.) samsort(Raw,Sorted) is like **sort**(Raw,Sorted) except that it does not discard duplicate elements. samsort(Order,Raw,Sorted) lets you specify your own comparison predicate, which the built-in sorting predicates **sort**/2 and **keysort**/2 do not. This file also exports two predicates for merging already-sorted lists: merge/3 and merge/4. See also library(ordered) and library(qsort).

setof.pl — library(setof) was present in release 1.6. The predicates bag_of_all/3 and set_of_all/3 are new in this release. Note that the built-in predicates **bagof**/3 and **setof**/3 were improved in this release, and are much more efficient than the predicates in this file. See also library(findall).

show.pl — The built-in command **listing**/1 displays dynamic predicates. But there is no built-in command for displaying the terms recorded under a given key. library(show) defines two predicates: show(Key) displays all the terms recorded under the given Key, and show/0 displays all the Keys and terms in the recorded data base.

showmodule.pl — library(show_module) provides a command for displaying information about a loaded module. show_module(Module) prints a description of the Module, what it exports, and what it imports. The command

```
| ?- show_module(_), fail ; true.
```

will print a description of every loaded module.

terms.{pl,c,h} — The Quintus Prolog foreign code interface provides means of passing constants between Prolog and C, Fortran, Pascal, etc. It is sometimes thought that it would be useful to pass general terms between Prolog and some other language. Passing actual terms across the foreign interface would not be a good idea at all. In fact, it would be a very silly thing to do, the foreign code could corrupt the system in all sorts of ways, and it would not be in your interests, because it would mean that we couldn't change the representation we use for terms. Which would mean that some of the good things coming couldn't come. But there is no harm in passing copies of terms across the interface. library(terms) lets you pass copies of terms from Prolog to C, and receive copies of terms from C. For example, the new built-in predicate **copy_term**/2 could have been defined this way, but wasn't:

```
'copy term'(Term, Copy) :-
        prolog_to_c(Term, Pointer_to_C_version),
        c_to_prolog(Pointer_to_C_version, Temp),
        erase_c_term(Pointer_to_C_version),
        Copy = Temp.
```

The C code in terms.c is just as much a part of this package as the Prolog code. In particular, the comments in that file describe the representation used on the C side of the interface and there are routines and macros (see terms.h) for accessing terms-in-C.

trees.pl — library(trees) is an implementation of arrays as binary trees.

types.pl — This file is support for the rest of the library, and is not really meant for general use. The type tests it defines are almost certain to remain in the library or to migrate to the system. The error checking and reporting code is certain to change. must_be_compound/3, must_be_proper_list/3, must_be_var/3, and proper_list/1 are

new in this release.

unix.{pl,c} library(unix) was present in release 1.6. For this release, it was split into library(files) — which is supported under VMS too — and library(unix).

vectors.{pl,c} The Quintus Prolog foreign code interface provides means of passing scalars between Prolog and C, Fortran, Pascal, etc. That isn't always enough. library(vectors) provides routines you can use to pass one-dimensional numeric arrays between Prolog and C, Pascal, or Fortran. See the comments in the code. Briefly,

list_to_vector(*+ListOfNumbers, +Type, -Vector*)

creates a vector, which you can pass to C. C will declare the argument as *Type**, and Prolog will declare the argument as +address(Type). Fortran will declare the argument as an array of Type.

make_vector(*+Size, +Type, -Vector*)

creates a vector which the foreign routine is to fill in. C will declare the argument as *Type**, and Prolog will declare the argument as +address(Type). Fortran will declare the argument as an array of Type.

vector_to_list(*+Vector, ?List*)

extracts the elements of the *Vector* as a list of numbers; if the *Vector* contains chars or ints, the *List* will contain integers, otherwise it will contain floating point numbers.

kill_vector(*+Vector*)

frees a vector. Don't forget to do this! You can still call vector_to_list/2 on a dead vector, until the next time memory is allocated. All that you can really rely on is that it is safe to create some vectors, call a C routine, kill all the vectors, and then extract the contents of the interesting ones before doing anything else.

6. Performance Measurements

The following table gives the results of running a suite of benchmarks written by Fernando Pereira. The two columns are the times taken, in milliseconds, to run each benchmark on a Sun 3/50 workstation under releases 1.6 and 2.0 respectively. These benchmarks have been designed to each measure a particular aspect of a Prolog implementation. In order to give accurate timings, most of the tests are run many times and the cost of the loop overhead for doing this is factored out.

The main things to note about these figures are that, because of the indexing of dynamic code, assert is slower but accessing an asserted clause can be many times faster. Also note that the speed of setof and bagof has been substantially improved.

	Quintus Prolog Release	
Benchmark	1.6	2.0
tail_call_atom_atom	1.58	1.37
binary_call_atom_atom	2.05	2.30
cons_list	2.42	2.77
walk_list	1.50	1.29
walk_list_rec	1.27	1.14
args(1)	1.23	1.11
args(2)	1.74	1.68
args(4)	3.41	2.78
args(8)	6.89	5.80
args(16)	13.24	11.55
cons_term	2.95	2.60

walk_term	2.30	1.64
walk_term_rec	2.08	1.61
shallow_backtracking	1.59	1.47
deep_backtracking	3.22	3.07
choice_point	3.35	3.07
trail_variables	3.62	3.70
medium_unify	1.81	1.69
deep_unify	118.67	109.83
integer_add	1.95	2.05
floating_add	9.17	8.95
arg(1)	5.50	5.02
arg(2)	5.49	5.00
arg(4)	5.48	5.01
arg(8)	5.49	5.01
arg(16)	5.49	5.01
index	2.29	2.16
assert_unit	1933.00	3717.00
access_unit	1051.33	39.50
slow_access_unit	1211.70	1171.80
setof	596.60	268.40
pair_setof	613.30	360.00
double_setof	1786.60	815.00
bagof	479.90	154.90

To complement these figures for small benchmarks, here are some figures for loading and running CHAT, a program of some 5000 lines of Prolog code which can be found in the demo directory of the Quintus Prolog installation directory. These figures were obtained on a Sun 3/110 which is a bit faster than a 3/50.

	Quintus Prolog Release	
Benchmark	1.6	2.0
compiling CHAT:	159.1 sec	160.9 sec
code space used:	170,156 bytes	173,900 bytes
time hi(questions):	19.4 sec	13.7 sec
stack space used:	26,928 bytes	24,932 bytes
heap space used:	136,068 bytes	129,676 bytes
consulting CHAT:	74.3 sec	67.4 sec
code space used:	195,416 bytes	279,180 bytes
time hi(questions):	242.7 sec	67.1 sec
stack space used:	197,288 bytes	32,356 bytes
heap space used:	481,404 bytes	422,876 bytes

The release 2.0 figures were obtained with the default garbage collection. The garbage collector only in fact ran during interpreted hi(questions) when it took 4.9 seconds which is included in the 67.1 seconds. The stack and heap usage figures are maxima during the run of hi(questions), and they were obtained by means of specially built versions of the Prolog releases.

Table of Contents