[PRODUCT-SPECS.NOTE   5:08pm Thursday, 5 July 1984]

PRODUCT SPECIFICATIONS

by David Warren

This document contains the preliminary specification of the Mark 1 version of Quintus Prolog, that will be ported to the Sun, Megaframe, and VAX, and that will constitute the final deliverable for the Burroughs contract. The current goal of Engineering is to produce software that precisely meets this specification, and nothing more. However it is proposed to review the specifications on October 1 to see whether any improvements are desirable and possible within timescales agreed between Engineering and Marketing.

The specification of the Mark 1 Quintus Prolog product is that its external behavior will be identical to Prolog-20, as defined in the Prolog-20 manual, with the following improvements, deletions, and aspects still to be determined.

Improvements

The following improvements will be made. Each is fully described in a separate document appended to this document (authors in parentheses):

(DW) o Static and dynamic procedures.

(WK) o Floating point numbers.

(WK) o Text editor (EMACS) interface.

(WK) o Improved input-output predicates.

(WK) o Style warnings.

(LB) o Improved debugger.

(LB) o C interface.

Deletions

The following items will be deleted from Prolog-20. Predicates marked with (**) will produce an aplogy message if called; predicates marked with (*) will succeed but will have no effect.

- o current_functor (**)
- o LC, NOLC (**)
- o Garbage collector: gc, nogc, gcguide (*)
- o log, nolog (**)
- o rename (**)
- o revive (**)
- o Mode declarations (will be ignored)
- o Public declarations (will be ignored: effectively all user predicates will be public)

To Be Defined

The following items have still to be defined. The person(s) chiefly responsible for producing a definition are shown in parentheses where known.

(WK,LB) o Variable names are (or may be) kept for debugging interpreted clauses.

(WK,LB) o Error messages and error handling.

(LB) o 'statistics' predicate

(WK, LB) o Installation procedures for Quintus Prolog

[MILESTONES.NOTE 6:10pm Thursday, 5 July 1984]

MILESTONES

by David Warren

Apr 1	Prototype achieves 5,000 lips on Sun-1.
May 1	Chat natural language demonstration program is running on Sun-2.
May 15	We achieve 13,500 lips on Sun-2 with preliminary indexing.
June 24	System ported to the Vax achieving 19,000+ lips on a 780.
Jul 6	System capable of being self-supporting for future development.

Aug 1	Megaframe arrives.
Sep 1	Beta-test systems available on Sun and Vax.
Oct 1	Initial Megaframe system delivered to SDC (Burroughs).
Nov 1	Specifications for Mark 1 System frozen. Product Announcement.
Nov 15	Code for Mark 1 System frozen on Sun and Vax.
Dec 1	Prototype interface to Ingres (or other relational database) on Sun.
1985	
Jan 1	Release of Mark 1 Product on Sun and Vax.
Jan 15	Final Megaframe system delivered to SDC (Burroughs).
Jul 1	Release of Mark 2 Product.

Engineering Roadmap through 31August

Suspend

Person	Finished	Task
✓ DB	9Jul	Sun and Vax systems brought into sync
LB	9Jul	single-character action mode, IO buffer flushing no (not portable)
BK	15Jul	fix read/write/tokens bugs, preliminary tuning
FP	15Jul	critical regions spec implementation
BK	20Jul	name special-casing integers
FP	22Jul	grammar rules tuned
VP	23Jul	find_definition in Emacs interface and Prolog hook
DB&DW	23Jul	fully working fast-assert, clause
LB	23Jul	fully functional 4-port interpreter/debugger
BK&VP	30Jul	toplevel exactly as will be delivered
LB&DW	30Jul	Interrupts
VP	2Aug	modifiable modeline from Emacs
BK	2Aug	dynamic procedure abstraction incorporated
DB&DW	3Aug	retract reclaiming space
FP	5Aug	fully operational and tuned Chat demo
LB&DW	6Aug	Implementation of critical regions
BK&DW	8Aug	setof tuned
LB	12Aug	tuned interpreter/debugger
DB	12Aug	proposal for comprehensive shakedown of system (20-31 Aug)
DW	12Aug	ruggedization of compiler
VP&BK	12Aug	Split-screen Emacs abstraction fully debugged

Continuing tasks

DB - system integration, source code management
BK - internal training program
LB - keeping Evan busy

13Aug - FEATURE FREEZE. No changes past this point except bug fixes.

13Aug-17Aug - Beta-release Documentation Week

Beta-test documentation (to go to Patti and Jonathan)

DB	incompatibilities with DEC10 Prolog
BK	Mark I features not included in beta-release
LB	designed-in limits
DW	known problems, bugs
VP&PK	brief description of editor interface

20Aug-31Aug

* System shakedown and debugging according to plan developed by DB
* Engineering planning for post-beta-release period

31Aug - Beta-release tape produced

STATIC-DYNAMIC.NOTE   5:07pm Thursday, 5 July 1984

SPECIFICATION FOR STATIC AND DYNAMIC PROCEDURES

by David Warren

Predicates can be declared to be either static or dynamic, eg:

```
        :- static foo/3, baz/1.
        :- dynamic toto/2.
```

If a procedure is static, the user is forbidden to modify or inspect it using assert, retract, clause, or listing. Attempts to do so will fail (giving an error message).

The default is static. However, if a declaration has not been given, and no procedure currently exists, a (user) call to assert a clause for that predicate will be treated as an implicit dynamic declaration.

Procedures can be created by consulting or compiling files. These are the only ways to create static procedures. Compiling a file produces faster code, but is slower and the code is not subject to the full inspection and debugging facilities. If a procedure is dynamic, the clauses will always simply be asserted, allowing them to be subsequently modified. Static procedures will be "compiled". [N.B. If the file is consulted, all the clauses will actually be stored in an interpreted form, which can be inspected by listing, etc. However, the user will still only be allowed to modify the dynamic procedures.]

Both 'compile' and 'consult' delete any previously defined procedures which are redefined in the file. Thus 'compile' behaves as on Prolog-20, while 'consult' is *NOT* the same as on Prolog-20, but instead behaves as Prolog-20's 'reconsult'. 'reconsult' will be defined as a synonym for 'consult' in a compatibility package.

Apart from the above, there is *NO* difference between the behavior of static and dynamic procedures. They can call each other interchangeably without any extra effort on the part of the user.

Quintus Prolog Floating point specification (Mark I)

Bill Kornfeld

Syntax:

```
    <float>:  <mantissa><E><exponent> | <integer>.<integer>

        <E>: e | E

 <mantissa>: +<mantissa1> | -<mantissa1> | <mantissa1>

<mantissa1>:  <integer>.<integer>

 <exponent>: +<integer> | -<integer> | <integer>
```

Examples: 1.0 0.54 123.0e45 0.35E-27

Semantics:

o All arithmetic operations with the exception of /, given two integers as arguments, behave exactly as they do in the DEC10 system.

o / always returns a float.

o A new function, //, is defined to act as / does now.

o The following arithmetic operations fail (or fault) if given one or more floating arguments:
 /\, \/, \, <<, >>, mod

o The operations: +, -, and * behave as follows. If given two floating arguments they return a float. If given one floating argument, and one integer argument, the integer is coerced to float, the operation performend, and a float returned.

o The comparison operations do coercion to float of integer arguments if one is a float. Two floats are =:= each other iff they are the same bit pattern.

o inside is: float(X) converts an integer to a float. integer(X) converts a float to an integer by truncating digits after the decimal point.

o number(+object), and float(+object) are additional type-checking predicates. (Presumably floats will be assigned their own tag).

o Two floats unify iff they have exactly the same bit pattern. Floats never unify with any other type.

```
o Writing floats obey the following rules:
  o All significant digits are shown.  Trailing zeros are deleted.
  o Non-exponential notation is used unless printing would entail
    three insignificant digits before the decimal point or three
    insignificant digits after the decimal point.
  o Non-exponential notation obeys the above grammar.
  o Exponential notation expresses the mantissa as a number between
    .1 and 1.

o name(Float,Chars) with Float instantiated to a float will succeed with
  Chars instantiated to the list of characters that would have been
  printed if the float had been printed.
```

Quintus Prolog Emacs Interface (Mark I)

William Kornfeld

Running Prolog

M-X Split-Screen

Puts Emacs into a two-window state, with Prolog running in the upper window and a text buffer in the lower window. All file operations apply to the lower window regardless of which one is selected when the operation is done. In other words, while in Split-Screen mode the Prolog window stays fixed in the upper window.

M-X No-Split-Screen

Puts Emacs back into the normal mode.

M-X Run-Prolog

Starts a Prolog running in an Emacs buffer. All typein and typeout happens through Emacs, making use of any text-editor commands desired.

Interrupting Prolog

M-X Send-Interrupt-Character (^X^C)

Sends a ^C interrupt to the Prolog process.

M-X Send-EOF-Character (^X^D)

Sends a ^D end-of-file signal to the process.

Adding to the database from Prolog

M-X Compile-Region

Takes the text between point and mark in the currently selected buffer and has Prolog compile it. Prolog will insert comments into its associated buffer indicating which procedures it has just compiled.

M-X Consult-Region

Takes the text between point and mark in the currently selected buffer and has Prolog consult it. Prolog will insert comments into its associated buffer indicating which procedures it has just consulted.

M-X Compile-Procedure

M-X Consult-Procedure

Behave like the above, except that the text sent to Prolog consists of the procedure that the cursor is currently sitting in. Emacs uses certain heuristics to find the bounds of the current procedure which work pretty well, but are not fool-proof due to the complexities of Prolog syntax.

Finding Definitions

M-X Find-Definition (M-.)

Takes the procedure whose name is next to the cursor, loads the appropriate file into an Emacs buffer (or selects the buffer if already loaded), and positions the cursor at the definition of the procedure.

Prolog Mode

When typing in the Prolog window, and in buffers containing Prolog source files, certain help is provided with the syntax. Matching parentheses and brackets will be shown as one types, also the following commands:

<tab>

Indents for Prolog. If after a :-, it indents three spaces. Otherwise it indents as per previous line.

<linefeed>

Does a newline followed by <tab>.

Facilities available to system programmers

emacs_find_definition(+functor,+arity,+module).

Switches to a buffer containing the source file for the specified procedure and positions the procedure definition on the screen.

```
emacs_display_message(+atom)
----------------------------
Causes Emacs to display the given message in the bottom line.
```

Mark I I/O enhancements

open(+Filename,+Type,-Stream)

Opens the given filename for the type of I/O specified by Type
(which must be one of input, output, and append). A stream
(atomic as far as the user is concerned) is returned.

The following predicates are upward compatible from the DEC10.
If given a file name they work the same way. If given a stream
argument they perform their operation on a stream, except that
see and tell do not (re)open the file, merely select it as standard
input or output: see, tell, seeing, telling, seen, told, close.

The I/O predicates read, write, writeq, print, get, put, get0, and
nl have been extended to take an additional, optional, argument.
The extra argument is a stream to which the I/O goes to instead
of the default stream.

What should seeing & telling do?
new seeing-stream and telling-stream?
[illegible]

Style Warnings

Bill Kornfeld

Defaultly consult, reconsult, and compile will call the style warnings facility. At the end of the outermost file operation the warnings, if any, will be displayed. There are three classes of style warning:

(1) Single variable occurence: A named variable has been used only once in a clause. (Variables beginning with "_" however are ignored by this check).

(2) Undefined procedure: A procedure call is present for a procedure that has no current definition, and is not defined in the file being read.

(3) Multiply defined procedure: The same procedure has clauses in more than one file.

User settable flags:

```
assert(no_style_check(all)).
    Turns off all style checking.

assert(no_style_check(undefined)).
    Turns off checking for undefined procedures.

assert(no_style_check(multiple)).
    Turns off checking for multiply-defined procedures.

assert(no_style_check(single_var)).
    Turns off checking for single variable occurences
```

Mark I Error handling

Mark I error handling will be in two areas, undefined procedure and I/O errors.

Undefined procedures:
There will be two modes, in the first mode, Prolog will print an error message and throw the user into the debugger. In the second mode, Prolog will fail.

I/O errors:
When an I/O error happens, the user will be given three choices as to course:

1. Retry I/O operation.
2. Change the file name and retry.
3. fail

no, too complicated

What about eof & other i/o errors not tied to open?

Quintus Specification
Lawrence
July 3rd 1984
CONFIDENTIAL, Proprietary to Quintus Computer Systems, Inc.

Quintus Prolog Debugger

0) Contents

1) Overview

The debugger has three aspects:

(A) The model it uses for presenting the execution of the Prolog program.
(B) The global operations which determine what the debugger will show and when and how it will do it.
(C) The way information is presented and the actions that can be performed during debugging.

Prolog-20 defines a solution to all of these. Restricted versions of the Prolog-20 debugger are present in PDP-11 Prolog and in C-Prolog. The description of debugging facilities in "Programming in Prolog" by Clocksin & Mellish follows the structure established by the Prolog-20 debugger. There is a published paper which specifies the execution model in more detail (Lawrence Byrd, Debrecen workshop 1980).

The initial Quintus Prolog debugger follows the Prolog-20 debugger except as described below.

2) Caveat

The user-ergonomics of the debugger are an important part of our product's image. How the concepts are decribed and the names that are used (such as "creep" and "leap") may need rethinking. I see the need for feedback from our test sites, and for an active role by the documentation staff. Difficulties in explaining the debugger or Prolog execution should force improvements in the terminology and/or the facilities themselves.

3) Execution model (A)

The debugger is a 4-port debugger as in Prolog-20. The execution model (1) will be improved by making the standard backtrack behaviour of the debugger follow the strategy used by the 'x' option in the Prolog-20 debugger. This only shows 'Redo' and 'Fail's for invocations which really have a choice point.

4) Global operations (B)

The global operations will be improved over Prolog-20 to make using the debugger less confusing as follows:

* The debugger will be in one of three states: off, on and start creeping, on and start leaping. Which state the debugger is in will be displayed to the user. The debugger will stay in one of these states until it is explicitly changed.

* Turning debugging off will not remove spypoints, although no infomation will be shown abourt these spypoints while debugging is off. A new predicate is provided:

nospyall	remove all spypoints

5) Presentation and actions (C)

The information presentation (3) will look the same as Prolog-20 but the options available will differ as follows:

* There will be additional options:

+	Place a spypoint on the procedure currently being looked at
-	Remove the spypoint on the procedure currently being looked at
=	SHow the debugging state ('debugging')

Quintus Specification
Lawrence
July 5th 1984
CONFIDENTIAL, Proprietary to Quintus Computer Systems, Inc.

Quintus Prolog C Interface under Unix

0) Contents

1) Overview

Quintus Prolog provides tools for integrating Prolog programs with programs written in C. This may be desirable for several reasons:

1) To speed up certain critical operations by writing them in the lower level language C.
2) To interface with the operating system and other libraries and programs.
3) To integrate an already existing C program with Prolog so that some of programming can be done in Prolog. An example here might be the use of Prolog to write the user interface and reasoning component of a design system in which graphics and numerical algorithms are written in C.

C programs are integrated with Prolog by compiling them and then linking these together with the supplied Prolog "object" files to produce new executable systems. This linking uses the standard Unix program 'ld'. C procedures can be called directly from Prolog in such composite systems. When building an integrated system the user specifies which C procedures should be callable from Prolog and how their arguments should be passed. The interface allows the passing and returning of Prolog's simple (atomic) data types. Complex data structures, such as lists and trees, cannot be passed directly between Prolog and C. However, complex data structures can be passed by unpacking them in Prolog and passing their atomic components.

Quintus provides a default definition of the main() procedure. This can be redefined by the user in which case the C program becomes responsible for starting the Prolog system. Prolog automatically initializes when it is first invoked.

Prolog manages its own working storage in sophisticated ways. It is important to be aware of this when intregrating C programs with Prolog.

2) Building a composite Prolog system

Quintus supplies two Unix object files which are used when building composite systems. These are:

prolog.o	The Quintus Prolog system
qpmain.o	Default definition of the main() procedure

Composite Prolog and C programs are built by linking together these files with the object files of the C program which have been generated by the C compiler. The standard Unix link editor 'ld' is used for this. The source code for qpmain.o is also provided. Section 5 describes how the user can change this file.

In addition to the object files supplied by Quintus, the program must also include some linkage information that helps Prolog and C communicate. This information describes which C procedures are to be accessible from Prolog, and how their arguments are to be passed. The information is prepared as a Prolog file which contains a list of facts describing the desired interfaces. The format of these facts is described is Section 3. The Prolog system predicate make_c_link(File) is then used to generate a C file which is compiled and loaded with the new system. This file defines data structures which are used by Prolog to link Prolog predicates to C procedures. Thus, the following files will be produced by the user:

```
        qplink.o        Table used to link C procedures into Prolog
        abc.o           }
        jkl.o           } The users compiled program (examples)
        pqr.o           }
```

The following example shows how these should all be prepared and linked together:

For this example we need to assume that some directory on your system has been set aside for the Prolog library where the supplied object files are kept. Let us set the csh variable $q to be an example library directory.

```
        % set q = "/usr/lib/prolog"
```

The user should already have produced some C files and also the file which describes the C procedures that will be accessible from Prolog. This is a Prolog file of a format described later.

```
        % ls
        abc.c   jkl.c   pqr.c   qplink.pl
```

The first step is to run Prolog and generate a C file which will handle the Prolog to C linkages. This is generated from the Prolog specification in qplink.pl.

```
        % prolog
        | ?- make_c_link(qplink).
        Generated Prolog to C link file: qplink.c

        | ?- halt.
```

All of the users C files need to be compiled, and also the qplink.c file. All of the following operations can be more easily achieved using the Unix 'make' facility. Refer to the make documentation for further details. This example shows all the steps explicitly.

```
        % cc -o abc.o  abc.c
        % cc -o jkl.o  jkl.c
        % cc -o pqr.o  pqr.c
        % cc -o qplink.o  qplink.c
```

The objects files are now linked together to produce a composite system.

```
        % ld $q/prolog.o $q/qpmain.o qplink.o abc.o jkl.o pqr.o
        % mv a.out newprolog
```

It is important that the supplied prolog.o file be the first file specified in the 'ld' command. This is so that the Prolog system starts at as low address as possible in the program's address space. This is necessary for reasons internal to the Prolog implementation.

[[Missing from specification: the bootstrap process which gets the constructed 'newprolog' up and running with all the Prolog system code loaded.]]

3) Calling C procedures from Prolog

C procedures are linked to Prolog predicates using facts which decribe the procedures and their argument patterns. As described in the previous section, these facts are pre-processed using the Prolog system predicate make_c_link(File) to produce a C file which forms part of the composite system.

The File used with make_c_link(File) should contain c_procedure fact for each Prolog predicate that is to be attached to a C procedure. These are of the form:

```
          c_procedure( <Pattern> ).
 or       c_procedure( <Pattern>, <CName> ).
```

where <Pattern> is:

```
          <predicate name>(<arg spec>,<arg spec>,...)
```

and <CName> is, optionally, the name of the C procedure (an atom),

and <predicate name> is the name of the Prolog predicate (an atom),

and there is an <arg spec> for each argument of the predicate, and <arg spec> is one of:

```
        +integer     +atom    +string
        -integer     -atom    -string
        [-integer]  [-atom]  [-string]
```

[[See cproc.eg1 and cproc.eg2 for examples]]

The interface allows the simple Prolog data types, atoms and integers, to be passed to C procedures and returned from C procedures. Prolog checks the types of the arguments it passes to C and the call will fail if any argument is not the right type. Prolog assumes that C will return results of the specified type.

The interface is responsible for all the data conversions between Prolog's internal representation and C's internal representation. The C program does not need to know how Prolog represents atoms and integers in order to interface with Prolog. This feature simplifies the integration of C and Prolog, and allows for compatibility across later versions of Quintus Prolog and versions of Quintus Prolog running on other hardware. In particular, this feature make it easier to interface directly with already written C procedures in libraries and so forth.

The <arg spec> specifications have the following meanings:

```
Prolog: +integer
C:      long int
```

The argument must be instantiated to an integer. The Prolog integer is converted to C integer and passed to the C procedure.

```
Prolog: +atom
C:      long unsigned
```

The argument must be instantiated to an atom, otherwise the call fails. A canonical representation of the Prolog atom is passed to the C procedure as an unsigned integer.

```
Prolog: +string
C:      char *
```

The argument must be instantiated to an atom, otherwise the call fails. A pointer to a null terminated string of characters containing the printed representation of the atom is passed to the C procedure. It is essential that this string is NOT overwritten by C.

```
Prolog: -integer
C:      long int *
```

A pointer to an integer is passed to the C procedure. It is assumed that C will overwrite this integer with the result it wishes to return. When the C procedure returns the pointed to integer is converted to a Prolog integer and unified with the provided Prolog argument. The argument can be of any type; if it cannot be unified with the returned integer then the call will fail. If the C procedure does not overwrite the integer then 0 will be returned. However, such uses should be considered an error and this particular behaviour SHOULD NOT be relied upon by programs.

Prolog: -atom
C: long unsigned *

A pointer to an unsigned integer is passed to the C procedure. It is assumed that C will overwrite this unsigned integer with the result it wishes to return. This result should be a canonical representation of an atom already obtained by C from Prolog. Returning an arbitrary integer will have undefined results. When the C procedure returns, the atom represented by the pointed to unsigned integer is unified with the provided Prolog argument. The argument can be of any type; if it cannot be unified with the returned atom then the call will fail. If the C procedure does not overwrite the unsigned integer then the atom 'bad_atom_from_c' will be returned. However, such uses should be considered an error and this particular behaviour SHOULD NOT be relied upon by programs.

Prolog: -string
C char * *

A pointer to a character pointer is passed to the C procedure. It is assumed that C will overwrite this character pointer with the result it wishes to return. This result should be a pointer to a null terminated string of characters. When the C procedure returns the atom which has the printed representation specified by the string is unified with the provided Prolog argument. The argument can be of any type; if it cannot be unified with the returned atom then the call will fail. Prolog copies the string if required, so that it is not necessary for C to worry about retaining it. If the C procedure does not overwrite the character pointer then the atom 'bad_atom_from_c' will be returned. However, such uses should be considered an error and this particular behaviour SHOULD NOT be relied upon by programs.

Prolog: [-integer]
C: return (long int)

No argument is passed to C. The return value from the C procedure is assumed to be an integer. It is converted to a Prolog integer and unified with the provided Prolog argument. The argument can be of any type; if it cannot be unified with the returned integer then the call will fail.

Prolog: [-atom]
C: return (long unsigned)

No argument is passed to C. The return value from the C procedure is assumed to be an unsigned integer which should be a canonical representation of an atom already obtained by C from Prolog. Returning an arbitrary integer will have undefined results. The atom represented by the unsigned integer is unified with the provided Prolog argument. The argument can be of any type; if it cannot be unified with the returned atom then the call will fail.

Prolog: [-string]

C   return (char *)

No argument is passed to C. The return value from the C procedure is assumed to be an character pointer pointing to a null terminated string of characters. The atom which has the printed representation specified by the string is unified with the provided Prolog argument. The argument can be of any type; if it cannot be unified with the returned atom then the call will fail. Prolog copies the string if required, so that it is not necessary for C to worry about retaining it.

Arguments are passed to C in the same order as they appear in the Prolog call. Only one "return value" argument can be specified; that is, there can be only one [-integer], [-atom] or [-string] specification. There need not be any "return value" argument in which case the value returned by the C procedure is ignored. Note that both input and output arguments are passed to C, except of course for the "return value" argument if present. Each input argument is appropriately converted and passed, each output argument is passed as a pointer through which C will send back the result.

Note that Prolog predicates attached to C procedures are always determinate. They may fail if their input arguments are of the wrong type, or if an output returned from C cannot be unified with the provided Prolog argument.

Prolog integers have a different precision than C integers. In the current Sun and Vax implementations Prolog integers are 29bit integers, whereas C long ints are 32bit integers. When integers are returned to Prolog from C they will be reduced to 29bit integers.

4) Access to Prolog atoms from C

The C interface allows Prolog atoms to be passed to C either in a canonical form as unsigned integers, or as pointers to character strings.

C programs can store canonical atoms in data structures and pass them around and back to Prolog, but they should not attempt to construct or decompose them. For each atom there is a single canonical representation. C programs can rely on this identity property. Note however, that the canonical form of atoms are NOT necessarily identical across different invocations of the program. This means that canonical atom representations should not be used in files or inter-program communication. For these purposes use strings.

Strings passed from Prolog to C should NOT be overwritten. Strings passed back from C to Prolog are automatically copied by Prolog if necessary. Thus the C program does not have to retain them and can reuse their storage space as desired.

In addition to obtaining and returning atoms through the interface Prolog provides two C procedures for converting back and forth between canonical atoms and strings.

```
        long unsigned QP_StrAtom(string)
```

```
                char *string;
```

Returns the canonical representation of the atom whose printed representation is string. The string is copied and the C routine can reuse the string and it's space.

```
        char * QP_AtomStr(atom)
                long unsigned atom;
```

Returns the string of characters for the canonical atoms printed representation. This string should NOT be overwritten by C.

Canonical atoms are particularly useful as constants to be used in passing back results from C procedures. The above routines can be used to initialize tables of such constants.

5) Changing start-up behaviour: redifining main()

[[See the example file qpmain.c]]

The supplied object file qpmain.o provides a default definition for the program's main() procedure. The source code is also supplied in qpmain.c. The definition of main() can thus be easily extended or replaced.

The default definition of main() uses the following routines:

```
        QP_CmdLine(argc,argv,env)
                int argc;
                char *argv[];
                char *env[];
```

This is the Prolog command line parser. In its normal use it parses the command line typed when Prolog is run. If the C program that is being integrated with Prolog also wishes to scan the command line then this action should be intercepted. It is then up to the C program to call QP_CmdLine with its required options, or options passed on from the users command line. QP_CmdLine should also be passed the programs environment table (env) so that this can be made available to Prolog.

```
        QP_Prolog(goal)
                char *goal;
```

This runs the Prolog system with an initial goal. goal should be a pointer to a string of characters. The initial goal will be the corresponding predicate with no arguments (arity of 0). When the goal terminates, by either succeeding or failing, then QP_Prolog() will return.

6) Prolog's storage management assumptions

The file qpmain.c also defines the variables:

QP_Strategy	storage management strategy flag
QP_Space	Default initial size of Prolog memory space

When QP_Prolog() is first called Prolog initializes itself and requests a large block of storage from Unix using the sbrk() system call. The initial size of this block is determined by the value of QP_Space. Prolog provides its own definition of the malloc() and related C library routines. Different storage allocation behaviours are specified by the value of QP_Strategy:

QP_Strategy == 1
  Prolog allocates space when QP_Prolog() is first called.
  Prolog will try and extend its space (with sbrk()) when it needs to.
  Space for malloc() is allocated from the Prolog storage area.

QP_Strategy == 2
  Prolog allocates space when QP_Prolog() is first called.
  Prolog does NOT extend its space. Overflows cause errors.
  Space for malloc() is allocated using sbrk() and is NOT taken from the Prolog storage area.

Strategy 1 is the standard stategy in which Prolog assumes control of memory allocation and is able to incrementally grow the amount of space it uses. Under strategy 1 the user C program should not use the brk() or sbrk() system calls. Strategy 2 is useful for C programs which do wish to make use of brk() or sbrk(), or programs which make considerable use of malloc(). Under strategy 2 there are no overheads involving movement of Prolog data areas which will be incurred by malloc() from time to time under strategy 1. However under strategy 2 Prolog uses a fixed size data area which cannot be expanded. This may be reasonable, given an adaquate initial size, for composite systems which wish to assume control for managing the rest of memory outside the area used by Prolog.

7) Prolog's input/output assumptions

Prolog uses the standard C I/O library for input and output, except for saving and restoring save states which use the Unix system calls open(2), read(2) and write(2) directly.

[[The Prolog I/O routines are not available to the user. This facility could be provided in later versions.]]

8) Debugging C procedures in composite systems

[[Documentation is required on how this is done. Some explanation of how to avoid wading through bits of the Prolog system itself would be useful. I am not currently sure if DBX will be useable with composite systems. This would be a loss if so.]]

```
/* QPMAIN.C : Default definition of main() for composite systems

Author:  Lawrence Byrd
Date:    June 1984
Version: 1

Supplied by Quintus Computer Systems, Inc.

*/

/* ----------------------------------------------------------------------
        Setting the default size for the Prolog memory space
   ---------------------------------------------------------------------- */

&define K 1024

long int QP_Strategy = 1;        /* storage management flag */

long int QP_Space = 1024*K;      /* default size of Prolog memory space */


/* ----------------------------------------------------------------------
        Default main procedure for composite Prolog systems
   ---------------------------------------------------------------------- */

main(argc,argv,env)
    int argc;
    char *argv[];
    char *env[];
    {
        QP_CmdLine(argc,argv,env);

        QP_Prolog("prolog_top_level");
    }
```

```
/* CPROC.EG1 : Some examples of C procedure interfaces

Author:   Lawrence Byrd
Date:     June 1984
Version:  1

Supplied by Quintus Computer Systems, Inc.

This is a set of examples to show the argument passing mechanism in the
Prolog to C interface. The examples shown do not perfrom any interesting
operations.

*/


/* ----------------------------------------------------------------------
        Prolog facts describing the interface
   ---------------------------------------------------------------------- */

c_procedure( p1(+integer,-integer) ).
c_procedure( p2(+integer,[-integer]) ).
c_procedure( p3(+atom,+string,+integer,-atom,-string,-integer,[-string]) ).

c_procedure( p4(+integer,-integer),  'FooBaz' ).


/* ----------------------------------------------------------------------
        Examples of how these procedures might be called from Prolog
   ---------------------------------------------------------------------- */

?- p1(42,X).
?- p1(42,36).                   /* fails if 36 is not returned */
?- p2(42,X).
?- p2(42,36).                   /* fails if 36 is not returned */

?- p1(A,B).                     /* fails because A is not an integer */
?- p2(hello,X).                 /* fails because 'hello' is not an integer*/

?- p3(england,john_brown,35,A,B,C,D).

?- p4(42,X).


/* ----------------------------------------------------------------------
        Some example C code
        This is to show the argument types - the operations are arbitrary
   ---------------------------------------------------------------------- */

void p1(a,b)
    long int a;
    long int *b;
    {
        *b = a+a+a;
    }
```

```
int p2(a)
    long int a;
    {
        return a+a+a;
    }


char *p3(atom,string,num,atomout,stringout,numout)
    long unsigned atom;
    char *        string;
    long int      num;
    long unsigned *atomout;
    char *        *stringout;
    long int      *numout
    {
        static long unsigned previous;
        static char buffer[123];

        *atomout = previous;
        sprintf(buffer,"%s for president",string);
        *stringout = buffer;
        *numout = num+num+num;

        previous = atom;

        return "yes";
    }


void FooBaz(a,b)
    long int a;
    long int *b;
    {
        *b = a+a+a;
    }


/* ---------------------------------------------------------------------
        Example of what the QPLINK.C file would look like given the
        interface described above.

        The exact details of this file are subject to change and should not
        concern the user anyway.
   --------------------------------------------------------------------- */

/* Automatically generated Prolog to C linkage file  <date> */

long int QP_Count = 4;

char *QP_Name[] =
    {
        "p1",
        "p2",
        "p3",
        "p4"
    };
```

```
long int QP_Arity[] =
    {
        2,
        2,
        7,
        2
    };

char QP_Args[] =
    {
        7, 8,
        7, 9,
        1, 4, 7, 2, 5, 8, 6
        7, 8,
    };

extern p1();
extern p2();
extern p3();
extern FooBaz();

 (*(QP_Proc[]))() =
    {
        p1,
        p2,
        p3,
        FooBaz
    };
```

```
/* CPROC.EG2 : Another example

Author:   Lawrence Byrd
Date:     June 1984
Version:  1

Supplied by Quintus Computer Systems, Inc.

*/
```

Imagine we have a list of pairs of numbers and names. We want to do some complex calculation on the numbers and then sort the list according to the the numbers from this calculation. The result we want is the list of names in the order of this new list. Let us assume that the calculation is so aweful that we must do it in C. To do this we move the data into a C array, call a C procedure to do the work and sort this array, then we read the data back out from the array to get a new Prolog structure.

This example shows a lot of data transfer. For many straight forward interfaces from Prolog to general capabilities (e.g. Unix calls) there is never that much data to transfer. Where there is, the user must determine if the increased speed of C over Prolog is worth the data transfer costs.

Interface:

```
        c_procedure( store_data(+integer,+integer,+atom) ).
        c_procedure( fetch_data(+integer,-atom) ).
        c_procedure( do_calculation_and_sort ).
```

Example goal:

```
        ?- do_calculation([ data(4,bob), data(5,jane),....], Answer).

        Answer = [jane,bob,...]
```

Prolog program:

```
        do_calculation(InList,OutList) :-
                store(InList,0,Count),
                do_calculation_and_sort,
                fetch(0,Count,OutList).

        store([],N,N).
        store([data(Number,Name)|Rest],N,Count) :-
                store_data(N,Number,Name),
                N1 is N+1,
                store(Rest,N1,Count).

        fetch(N,Count,[]) :- N >= Count, !.
        fetch(N,Count,[Name|Rest]) :-
                fetch_data(N,Name),
                N1 is N+1,
                fetch(N1,Count,Rest).
```

C code:

```
typedef long unsigned ATOM;

struct { int numfield; ATOM namefield; } array[SOMESIZE];

store_data(n,num,name)
  int n, num; ATOM name;
  {
    array[n]->numfield = num;
    array[n]->namefield = name;
  }

do_calculation_and_sort()
  {
    /* Who knows? Sorts array[] though */
  }

fetch_data(n,name)
  int n; ATOM *name;
  {
    *name = array[n]->namefield;
  }
```