SHORT GUIDE TO UNIX PROLOG IMPLEMENTATION SHORT TO UNI X PROLOG IMPLEMENTATION

C.S.Mellish,
25 October 1979

1. Introduction

This is a very short note to help anybody who might wish to alter or make additions to UNIX PROLOG. A fuller description may appear later, but for now this summary is all that is available. The description applies to version NU4 of the system, although many details are more or less independent of the version. It is suggested that the reader have available the code of the system and use this document primarily as a way of getting a global picture of the implementation and of understanding whatever obscurities the code and its comments may have.

Some of the design decisions now look rather strange in hindsight, but this is not the place for justifications or suggestions for improvements.

2. Storage of Programs

The PROLOG system is an interpreter, and so the storage of user programs involves little more than representing in a linear form the syntactic structure of the clauses.

A fundamental component of the representation system is the representation of atoms. Whenever an atom is read (through DOREAD), its characters are stored and used to produce a hash number (this happens in GETATOM). This number is then used as an index to look up a word in the hash table (which extends from HASHSTART to HASHEND). Each word in the hash table contains either zero or the address of a dictionary entry for an atom. If the word that is obtained contains zero, this means that the atom has not been previously encountered. In this case a new dictionary entry is created and the word in the table made to point to that. On the other hand, if the word obtained is non-zero, it is necessary to check whether the dictionary entry pointed to is the one for the atom read. (This is done by comparing the characters that were read with the characters stored in the dictionary entry) If the entry is for some other atom, the system looks a fixed number of words (31) further on in the table to try and get some success there. This process continues until the system has finally located a word in the table that points to the dictionary entry for the atom (a new entry being created if ever a zero word is encountered). The position of the word in the table ((its addr - HASHSTART)/ 2) is called the ATOM NUMBER and is used in various places to characterise that atom.

The format of the dictionary entry for an atom is shown in fig 1. The first word of the entry contains either zero or the address of the first clause with that atom as its main predicate. That clause points to the second clause, and

```
            |----------------------|
Low addr:   | Addr of first clause |
            |----------------------|
            |  After   |  Before   |
            |Precedence|Precedence |
            |----------------------|
            | 1st char |  Length   |
            |----------------------|
            | 3rd char | 2nd char  |
            |----------------------|
            | .....       .....    |
            |----------------------|
High addr:  |     0    | nth char  |
            |__________|___________|
```

Figure 1: Dictionary Entry for Atom

Figure

so on until a zero pointer is found. The next two bytes give precedence information about how the atom behaves as an operator. They usually contain zero, but can be filled by the routine OP. The 'length' stored in a dictionary entry is the number of characters in the name (n, say) +1, all rounded up to a multiple of 2. If n is even, the dictionary entry is padded out with a null byte at the end.

The representation of a clause is illustrated in figure 2.

```
            |----------------------|
Low addr:   | Addr of next clause  |
            |----------------------|
            | Number of Variables  |
            |----------------------|
            |  ......              |
            |  Head of clause      |
            |             ......   |
            |----------------------|
            |  ......              |
            |  1st goal            |
            |             ......   |
            |----------------------|
            |  ......              |
            |             ......   |
            |----------------------|
            |  ......              |
High addr:  |  nth goal            |
            |             ......   |
            |______________________|
```

Figure 2: Clause Representation

Figure

The main components of the representations of clauses are the representations of the head term and the goals in the body of the clause. Each one of these may occupy several words. Its extent is well defined because of the way terms are represented. The representation of any goal in a clause happens to begin with a word containing an odd number whereas each atom entry or clause begins with an even value. Hence the end of a clause does not have to be specially marked.

The representations of the terms that are the head and the goals of clauses are called SKELETONS and are formed by the following rules:

- For an integer - the representation takes the form of 1 word in the format (14 bits)01, where the 14 bits give the value of the integer.
- For a variable - the representation takes the form of 1 word containing the value 2*n, where n is the number of the variable in the clause (1 <= n <= number of vars).
- For an anonymous variable (a variable that only occurs once in a clause and hence does not have to be allocated any space) - a single word containing 0.
- For an atom - the representation is a single word with the format (4 bits)(10 bits)11, where the first 4 bits are zero and the 10 bits are the atom number.
- For a complex term - this is represented in a sequence of words, giving the prefix Polish structure of the term. Thus it begins with a word representing the functor of the term and continues with the representations of the arguments in turn. (The arguments may, of course, themselves be complex terms). The format of a word representing a functor is (4 bits)(10 bits)11, where the 4 bits give the arity and the 10 bits give the atom number of the functor.

An example of the representation of a term is given in figure 3.

```
                  |----------------|
Low addr:         |0011001101000111|  ( 3, f)
                  |----------------|
                  |0000000000000101|  (1)
                  |----------------|
                  |0000000000000010|  (var 1)
                  |----------------|
                  |0010010001011111|  (2, g)
                  |----------------|
                  |0000000000000010|  (var 1)
                  |----------------|
High addr:        |0000000000000000|  (anon)
                  |________________|
```

Figure 3: Skeleton for f(1,X,g(X,Y))

Examples of atom entries and clauses can be seen at the end of the PROLOG system code. Even where they have special system-defined properties (ie when

they correspond to evaluable predicates) all atoms and clauses must be represented as above. If a predicate is associated with a system function, the address of the routine must be placed in the word before the start of a dummy clause for the predicate. The clause must also appear before the label SFSTART (and only such clauses can appear there). Then when the head of the clause is matched, instead of looking at the body of the clause the system calls the special routine.

3. Storage of Data in Execution

When the user runs a program, he assigns values to variables corresponding to invocations of clauses. When a clause is invoked, the system assigns a block of storage with 1 word for each variable of the clause. The initial value of each variable is 'undefined', which is represented by the value 0. During the execution, the value may change to one of the following:

- The address of another variable. This happens when two uninstantiated variables are unified. The later instantiation of the second one must result with the same value being associated with the first. The presence of an address where the value of a variable should be says effectively "To get the value, look in this other place".

- An atomic item (integer or atom). In this case the representation of the item (being the same as in a skeleton) is put in the word.

- A complex term. Complex terms created during execution (DS's) are placed in the global stack. If a variable is given such a value, a pointer to the structure is placed in the variable's word. A DS is effectively a copy of a skeleton with values like those we are currently discussing substituted for the variables that occur in it.

4. Core Allocation

The organisation of core allocation has been designed to make the most of the (rather inappropriate) facilities that UNIX offers in this area. The system attempts never to demand more from the operating system than it is likely to need. First of all, the overall layout of core is illustrated in figure 4. At any point, the PROLOG system must be able to access the area between the lowest address and GTOP, as well as the area between LTOP and TOP. The former of these is in general allocated as UNIX data area, with DATLIM giving the address of the highest location that can currently be accessed. The latter is in general allocated as UNIX stack area, with STLIM giving the address of the next word below this area. When the stack area attempts to expand over 4K words further expansion is prevented and the data area expands instead to meet it. The flag CONNECTED indicates whether this condition holds. Although the PROLOG system relinquishes control of data space that it does not immediately need (through TRIMCORE), it is unfortunately not possible to do this with stack

Lowest Address:	System code " " "
HASHSTART:	Hash table for atoms

```
                                  "    "    "
        HASHEND:                  System clauses and atoms
                                  "    "    "    "    "
        FREESTART:                User's clauses and atoms (heap)
                                  "    "    "    "    "    "    "
        HTOP-->                   Free area above heap
                                  "    "    "    "    "
        HLIMIT-->                 Previous global stack areas
                                  "    "    "    "    "    "
        GSTART-->                 Current goal clause and DS
                                  "    "    "    "    "    "
        GBOT-->                   Current global stack
                                  "    "    "    "    "
        GTOP-->                   Free area between stacks
                                  "    "    "    "    "    "
        LTOP-->                   Current local stack
                                  "    "    "    "    "
        LBOT-->                   Previous local stack areas
                                  "    "    "    "    "    "    "
        TOP-->                    Top of core
```

Figure 4: Core Layout

space. Since it is useful to keep the SP register to handle subroutine calls within the interpreter, SP is not used as a pointer to the stacks at the top of core except when more space needs to be allocated there. In general, SP points inside a special workspace area extending downwards from WSTART.

Because of the facility to do nested 'consult's and 'breaks', it is possible that the user may have several different execution environments active at the same time. The main parameters of the current environment are to be found between ENVSTART and ENVEND. The parameters for previous environments are kept on the global stack.

Within the space that the system uses, there are basically three main data areas:

1. The 'heap'. This is not really a heap, because it is not garbage collected. Instead it is a stack containing all the permanent clauses and atom dictionary entries. This area starts at FREESTART and its top is given by HTOP. The expansion is limited to below HLIMIT, which is where the global stacks start. If expansion further is attempted, the global stacks are automatically moved up to make space. Anything placed on the heap must start with a word with an even value (because there are no explicit clause terminators). Note that the function to check the availability of space on the heap (HALLOC) does not update HTOP. This is so that it is possible to put values there but to postpone making them permanent until the last

   minute. In this way one can avoid permanently messing up the heap if an error occurs half way through putting something on.

2. The global stack areas. There is one global stack for each active user execution. The first one's start is given by HLIMIT and the

current one's by GBOT. The top of the current one is given by GTOP. The global stack is used to store terms constructed during the execution (referred to in many comments as DS's). In addition, the DS and clause associated with the current user goal are kept just below where GBOT points, pointed to by GSTART. A second function of the global stack is to contain the trail - the list of variables whose values must be reset on backtracking. Trail entries are interspersed with DS's on the global stack; they can be told apart because a trail entry is a single address (even value) whereas a DS starts with an odd value. If the stack is moved up, the pointers GBOT, GSTART, OPSTART, HLIMIT and GTOP are updated, as are any pointers into the global stack from itself or the local stack. No other pointers are updated. Putting anything on the global stack must be preceded by an appropriate call of GALLOC.

3. The local stack areas. There is one local stack for each active user execution. The local stacks expand downwards, whereas the global stacks and heaps expand upwards. The first local stack begins at the address given in TOP. The current one begins at the value of LBOT and extends to the value of LTOP. Local stacks are used to contain administrative information about the user's execution as well as cells for variable values. These are allocated in groups called stack frames. Putting anything on the local stack must be preceded by an appropriate call of LALLOC. Note that the part of the stack used during the satisfaction of a goal will be reclaimed if this succeeds determinately.

5. Keeping track of Execution

The code for the main loop of the interpreter (from CONTINUE to GOON) is fairly well commented, but the main structure can be summarised here.

Given the skeleton for a goal to be satisfied (at GOTCALL) the system obtains the atom which acts as its functor and finds the first clause with that atom as predicate (now at GOTCLAUSE). It must then allocate a stack frame on the local stack for a new invocation of the clause. It starts off by recording administrative information and then (at ENTER) allocates space for the variables of the clause. It then invokes the unification routine (UNIFY). One argument to this is a pointer to the skeleton that mentioned this goal, together with a pointer to the stack frame which gives the values of the variables in the skeleton. The other argument is a pointer to the skeleton at the head of the clause, together with the address of the new stack frame (giving the values of the variables for the new invoocation).

If unification succeeds, (now at CONTINUE) it must first be checked to see if a system clause has been activated. If not, the search for the next goal can take place immediately. The first place to look is after the head of the new clause (now at TRYCALL). If there is a goal there, that is fine; otherwise we have completed the current goal. In this case (now at RETURN) we must look back to the goal that this was a subgoal of, and so on until a remaining unsatisfied goal can be found. If this process goes back right to the original user goal, we have been successful (SUCCESS).

If unification fails, or there are no clauses for the predicate, backtracking takes place (FAIL). After resetting the values of the variables given in the trail, the system looks for the last choice point (The address of its stack frame is given in NB). If there is none, the user goal cannot be satisfied (FAILURE). Otherwise NB is reset to the previous choice point and a new clause is picked for the place where the choice was made. (Now back to GOTCLAUSE).

How does the execution process start (at EXECUTE)? The goal that the user gives is 'asserted' as the body of a clause for the predicate '' (the atom with an empty name). That is, the atom entry for '' (at EDUMMY) is made to point to a dummy clause at the start of the global stack. The system then gets the address (DCALL) of a dummy goal of '' and starts the interpreter cycle at GOTCALL.

The details of the interpreter loop can only really be understood if the format of stack frames is explained. Figure 5 does this.

6. Unification

During the execution of a user's program, there are two ways in which

instantiations of the skeletons in his clauses can be characterised. Firstly, giving the address of a skeleton plus the address of a stack frame (normally the 'base' address) gives enough information to specify the term denoted, because the skeleton gives the basic structure and the stack frame gives the values of the variables that fit into that structure. Secondly, giving the address of a concrete DS is also a complete specification. The basic unification routines make use of the similarities in the representation of skeletons and DS's and work for both types of representation. The only difference is that if the number of a variable is encountered (which only happens in a skeleton) the appropriate environment (stack frame) must be available, so that the value of the variable can be looked up. The locations E, E1 and E2 are used to hold currently used environment pointers.

The routine IDENT is fundamental for unification. Its function is to interpret the data representations. IDENT is called to identify the type of a term and also to return an appropriate value result. It can be applied to a variable cell or to part of a skeleton or DS. If it is used on a skeleton, the environment pointer E must be set appropriately. The possible results that IDENT can return are given in fig 6. Most of what IDENT does is just following pointers, decoding the last two bits of words and looking up the values of

```
                    ¦-------------------¦
Low addr:           ¦ Cell for var n    ¦
                    ¦-------------------¦
                    ¦  ......           ¦
                    ¦           ......  ¦
                    ¦-------------------¦
                    ¦ Cell for var 1    ¦
                    ¦-------------------¦
                    ¦ Addr of clause    ¦
                    ¦ being used        ¦
```

```
                   |-------------------|
                   | Addr of call of   |
                   | this goal         |
                   |-------------------|
                   | Base of frame for |   <<== Base of stack frame (FRAME)
                   | parent goal       |
                   |-------------------|
                   | Base of frame for |   } Only
                   | last choice point |   } present
                   |-------------------|   } for
                   | Value of GTOP
```

- For an anonymous variable (a variable that only occurs once in a clause and hence does not have to be allocated any space) - a single word containing 0.

- For an atom - the representation is a single word with the format <4 bits><10 bits>11, where the first 4 bits are zero and the 10 bits are the atom number.

- For a complex term - this is represented in a sequence of words, giving the prefix Polish structure of the term. Thus it begins with a word representing the functor of the term and continues with the representations of the arguments in turn. (The arguments may, of course, themselves be complex terms). The format of a word representing a functor is <4 bits><10 bits>11, where the 4 bits give the arity and the 10 bits give the atom number of the functor.

An example of the representation of a term is given in figure 3.

```
                |----------------|
Low addr:       |0011001101000111|  ( 3, f)
                |----------------|
                |0000000000000101|  (1)
                |----------------|
                |0000000000000010|  (var 1)
                |----------------|
                |0010010001011111|  (2, g)
                |----------------|
                |0000000000000010|  (var 1)
                |----------------|
High addr:      |0000000000000000|  (anon)
                |________________|
```

Figure 3: Skeleton for f(1,X,g(X,Y)) Figure

Examples of atom entries and clauses can be seen at the end of the PROLOG system code. Even where they have special system-defined properties (ie when

they correspond to evaluable predicates) all atoms and clauses must be represented as above. If a predicate is associated with a system function, the address of the routine must be placed in the word before the start of a dummy clause for the predicate. The clause must also appear before the label SFSTART (and only such clauses can appear there). Then when the head of the clause is matched, instead of looking at the body of the clause the system calls the special routine.

3. Storage of Data in Execution

When the user runs a program, he assigns values to variables corresponding to invocations of clauses. When a clause is invoked, the system assigns a block of storage with 1 word for each variable of the clause. The initial value of each variable is 'undefined', which is represented by the value 0. During the execution, the value may change to one of the following:

- The address of another variable. This happens when two uninstantiated variables are unified. The later instantiation of the second one must result with the same value being associated with the first. The presence of an address where the value of a variable should be says effectively "To get the value, look in this other place".

- An atomic item (integer or atom). In this case the representation of the item (being the same as in a skeleton) is put in the word.

- A complex term. Complex terms created during execution (DS's) are placed in the global stack. If a variable is given such a value, a pointer to the structure is placed in the variable's word. A DS is effectively a copy of a skeleton with values like those we are currently discussing substituted for the variables that occur in it.

4. Core Allocation

The organisation of core allocation has been designed to make the most of the (rather inappropriate) facilities that UNIX offers in this area. The system attempts never to demand more from the operating system than it is likely to need. First of all, the overall layout of core is illustrated in figure 4. At any point, the PROLOG system must be able to access the area between the lowest address and GTOP, as well as the area between LTOP and TOP. The former of these is in general allocated as UNIX data area, with DATLIM giving the address of the highest location that can currently be accessed. The latter is in general allocated as UNIX stack area, with STLIM giving the address of the next word below this area. When the stack area attempts to expand over 4K words further expansion is prevented and the data area expands instead to meet it. The flag CONNECTED indicates whether this condition holds. Although the PROLOG system relinquishes control of data space that it does not immediately need (through TRIMCORE), it is unfortunately not possible to do this with stack

Lowest Address:	System code
	" " "
HASHSTART:	Hash table for atoms
	" " " "
HASHEND:	System clauses and atoms
	" " " " " "
FREESTART:	User's clauses and atoms (heap)
	" " " " " " "

```
HTOP-->                Free area above heap
                       "    "    "     "
HLIMIT-->              Previous global stack areas
                       "        "      "     "
GSTART-->              Current goal clause and DS
                       "       "    "      "   "
GBOT-->                Current global stack
                       "       "      "
GTOP-->                Free area between stacks
                       "    "    "       "
LTOP-->                Current local stack
                       "       "     "
LBOT-->                Previous local stack areas
                       "        "     "     "
TOP-->                 Top of core
```

Figure 4: Core Layout

space. Since it is useful to keep the SP register to handle subroutine calls within the interpreter, SP is not used as a pointer to the stacks at the top of core except when more space needs to be allocated there. In general, SP points inside a special workspace area extending downwards from WSTART.

Because of the facility to do nested 'consult's and 'breaks', it is possible that the user may have several different execution environments active at the same time. The main parameters of the current environment are to be found between ENVSTART and ENVEND. The parameters for previous environments are kept on the global stack.

Within the space that the system uses, there are basically three main data areas:

1. The 'heap'. This is not really a heap, because it is not garbage collected. Instead it is a stack containing all the permanent clauses and atom dictionary entries. This area starts at FREESTART and its top is given by HTOP. The expansion is limited to below HLIMIT, which is where the global stacks start. If expansion further is attempted, the global stacks are automatically moved up to make space. Anything placed on the heap must start with a word with an even value (because there are no explicit clause terminators). Note that the function to check the availability of space on the heap (HALLOC) does not update HTOP. This is so that it is possible to put values there but to postpone making them permanent until the last

minute. In this way one can avoid permanently messing up the heap if an error occurs half way through putting something on.

2. The global stack areas. There is one global stack for each active user execution. The first one's start is given by HLIMIT and the current one's by GBOT. The top of the current one is given by GTOP. The global stack is used to store terms constructed during the execution (referred to in many comments as DS's). In addition, the DS and clause associated with the current user goal are kept just below where GBOT points, pointed to by GSTART. A second function of the

global stack is to contain the trail - the list of variables whose values must be reset on backtracking. Trail entries are interspersed with DS's on the global stack; they can be told apart because a trail entry is a single address (even value) whereas a DS starts with an odd value. If the stack is moved up, the pointers GBOT, GSTART, OPSTART, HLIMIT and GTOP are updated, as are any pointers into the global stack from itself or the local stack. No other pointers are updated. Putting anything on the global stack must be preceded by an appropriate call of GALLOC.

3. The local stack areas. There is one local stack for each active user execution. The local stacks expand downwards, whereas the global stacks and heaps expand upwards. The first local stack begins at the address given in TOP. The current one begins at the value of LBOT and extends to the value of LTOP. Local stacks are used to contain administrative information about the user's execution as well as cells for variable values. These are allocated in groups called stack frames. Putting anything on the local stack must be preceded by an appropriate call of LALLOC. Note that the part of the stack used during the satisfaction of a goal will be reclaimed if this succeeds determinately.

5. Keeping track of Execution

The code for the main loop of the interpreter (from CONTINUE to GOON) is fairly well commented, but the main structure can be summarised here.

Given the skeleton for a goal to be satisfied (at GOTCALL) the system obtains the atom which acts as its functor and finds the first clause with that atom as predicate (now at GOTCLAUSE). It must then allocate a stack frame on the local stack for a new invocation of the clause. It starts off by recording administrative information and then (at ENTER) allocates space for the variables of the clause. It then invokes the unification routine (UNIFY). One argument to this is a pointer to the skeleton that mentioned this goal, together with a pointer to the stack frame which gives the values of the variables in the skeleton. The other argument is a pointer to the skeleton at the head of the clause, together with the address of the new stack frame (giving the values of the variables for the new invoocation).

If unification succeeds, (now at CONTINUE) it must first be checked to see if a system clause has been activated. If not, the search for the next goal can take place immediately. The first place to look is after the head of the new clause (now at TRYCALL). If there is a goal there, that is fine; otherwise we have completed the current goal. In this case (now at RETURN) we must look back to the goal that this was a subgoal of, and so on until a remaining unsatisfied goal can be found. If this process goes back right to the original user goal, we have been successful (SUCCESS).

If unification fails, or there are no clauses for the predicate, backtracking takes place (FAIL). After resetting the values of the variables given in the trail, the system looks for the last choice point (The address of its stack frame is given in NB). If there is none, the user goal cannot be satisfied

(FAILURE). Otherwise NB is reset to the previous choice point and a new clause is picked for the place where the choice was made. (Now back to GOTCLAUSE).

How does the execution process start (at EXECUTE)? The goal that the user gives is 'asserted' as the body of a clause for the predicate '' (the atom with an empty name). That is, the atom entry for '' (at EDUMMY) is made to point to a dummy clause at the start of the global stack. The system then gets the address (DCALL) of a dummy goal of '' and starts the interpreter cycle at GOTCALL.

The details of the interpreter loop can only really be understood if the format of stack frames is explained. Figure 5 does this.

6. Unification

During the execution of a user's program, there are two ways in which instantiations of the skeletons in his clauses can be characterised. Firstly, giving the address of a skeleton plus the address of a stack frame (normally the 'base' address) gives enough information to specify the term denoted, because the skeleton gives the basic structure and the stack frame gives the values of the variables that fit into that structure. Secondly, giving the address of a concrete DS is also a complete specification. The basic unification routines make use of the similarities in the representation of skeletons and DS's and work for both types of representation. The only difference is that if the number of a variable is encountered (which only happens in a skeleton) the appropriate environment (stack frame) must be available, so that the value of the variable can be looked up. The locations E, E1 and E2 are used to hold currently used environment pointers.

The routine IDENT is fundamental for unification. Its function is to interpret the data representations. IDENT is called to identify the type of a term and also to return an appropriate value result. It can be applied to a variable cell or to part of a skeleton or DS. If it is used on a skeleton, the environment pointer E must be set appropriately. The possible results that IDENT can return are given in fig 6. Most of what IDENT does is just following pointers, decoding the last two bits of words and looking up the values of

```
                     |-------------------|
Low addr:            | Cell for var n    |
                     |-------------------|
                     |  ......           |
                     |           ......  |
                     |-------------------|
                     | Cell for var 1    |
                     |-------------------|
                     | Addr of clause    |
                     | being used        |
                     |-------------------|
                     | Addr of call of   |
                     | this goal         |
                     |-------------------|
                     | Base of frame for |  <<== Base of stack frame (FRAME)
```

```
                   | parent goal         |
                   |---------------------|
                   | Base of frame for   |   } Only
                   | last choice point   |   } present
                   |---------------------|   } for
                   | Value of GTOP       |   } Choice
High addr:         | when frame created  |   } Points
                   |_____________________|
```

Figure 5: Stack Frame Representation

Type result (in R0)	Value result (in R1)
0	The representation of an integer
2	The address of an undefined variable
4	The representation of an atom
6	The address of a complex skeleton/DS
8	The location of an anonymous variable

Figure 6: Possible Results of IDENT

variables in environments.

The main unification routine UNIFY is normally called with two skeletons and environment pointers. These correspond to the call of the current goal in another clause and the head of the new clause that is being tried. In fact it will work with any two things that IDENT can interpret. The process of unification involves following the tree structure of the two terms, checking that functors are equal, and then doing special actions whenever a leaf of a tree is encountered. If UNIFY performs a normal subroutine return, unification has been successful and all the necessary substitutions have taken place. The only other possible action is for a jump to FAIL to take place.

UNIFY starts by calling IDENT for each argument, and then jumps to an appropriate routine according to the combination of types. Some routines just

involve comparing two values (TESTEQ), some mean immediate failure (UNIFAIL), and some involve simply assigning a value to a variable (VTO1, VTO2, ASS2V). If one variable is made to point to another, it must be ensured that pointers go in such a direction that if pieces of stack are reclaimed there will be no pointers into limbo. The rules for ASS2V make sure of this (In this context, a 'local' variable cell is a cell in the local stack whereas a 'global' cell is one that appears as part of a DS on the global stack). Moreover, whenever any value is assigned to a variable, the address of that variable must be put on the trail if the value has to be reset on backtracking (Routine TRAIL).

The more difficult cases occur when complex terms are involved. If two complex terms are being unified (SA12) a certain amount of manoeuvring is necessary to take account of whether the terms are part of larger terms in prefix format or not. Then unification is called recursively on corresponding arguments (assuming identity of functors). If a complex term is unified with an uninstantiated variable, two possible cases arise, according to whether the term is in the form of a skeleton (when it appears below GBOT) or a DS. If it is a DS, it is a simple matter to put its address in the variable cell. Otherwise a copy of the skeleton is put on the global stack (routine FLESH), with appropriate values substituted for the variables. (This is where DS's are created). The variable can then point to this.

7. Routines for Evaluable Predicates

The way in which special routines are invoked for evaluable predicates was discussed in earlier sections. This section gives some guidelines for writing such routines.

When a routine for an evaluable predicate is called, R1 points to the 'variable 1' slot of the current stack frame. The variable ARGPTR also holds this address. The normal procedure is to start by checking the types of the arguments, through IDENT, storing the values as necessary (All the registers except SP are available in these routines). The the work for the predicate is done, and a normal subroutine exit indicates successful satisfaction of the goal.

If the goal is not satisfiable, it suffices for the routine to cause a jump to FAIL. FAIL will clear up various bits and pieces, and does not require any registers to be set. Alternatively, if a serious error occurs, the routine ERROR should be called. This must be a proper subroutine call, even though the call will never return. The number of the error should be placed in the word after the subroutine call, so that it can appear in error messages.

General points to note when writing evaluable predicate routines are the following. Firstly, if a value is assigned to any variable cell, TRAIL must be called. Secondly, any additions to the stacks or heap should be preceded by calls of the appropriate ALLOC functions and must take into account how items in these areas are interpreted. Thirdly, if HALLOC is called, the global stacks may have shifted by the time it returns - so if one has pointers into the global stack in such situations it is necessary to use OPSTART or keep addresses relative to some pointer that will be updated (eg GBOT). Finally, the fact that the user can specify 'abort' to happen immediately after an interrupt means that evaluable predicate routines must be prepared to be discontinued at any point. This means that permanent additions to the heap or changes to system variables must be carefully timed or interrupts disallowed for their duration (through INTROFF and INTRON).

Finally there is the task of producing dummy clauses and atom entries for new evaluable predicates, and in particular of obtaining the atom number of a new predicate. Unfortunately there is no way of doing this properly (yet). The best way to discover what the atom number of an atom is seems to be to run PROLOG, type in a simple clause with the atom as predicate and then look at the core image (with DB, DDT or whatever) to interpret what has appeared after FREESTART.