CHAPTER 1

INTRODUCTION

NU-Prolog is intended to be a successor to the current generation of Prolog systems. Programs can be written more logically in NU-Prolog than is possible with other versions of Prolog; NU-Prolog is a step towards purely declarative logic programming. The language includes quantification and if-then-else; logical connectives such as implication; logical equivalents of the unsafe constructs which were necessary in earlier versions of Prolog; and **`when`** declarations, which delay the execution of predicates until their arguments are sufficiently instantiated, and can be used to produce code that is more logical and efficient. These features not only simplify programming, but improve readability and bring NU-Prolog programs closer to the ideal of programs written in pure logic. In particular, they make most uses of **`cut`** unnecessary.

Although the syntax and system predicates of NU-Prolog are essentially a superset of those available in DEC-10 Prolog [Bowe82], MU-Prolog [], and Quintus Prolog [Inc86][], NU-Prolog makes most of the uses of the non-logical or unsound aspects of these systems obsolete. A few DEC-10 and some Quintus Prolog predicates are not available, and some have slightly different effects, although most DEC-10 and Quintus Prolog programs should run with few alterations. However, NU-Prolog is a new generation Prolog system.

Prolog has long been recognised as a suitable language for writing database applications [Gall78]. NU-Prolog incorporates an external database facility which permits the storage of predicates in which the arguments may be arbitrary terms under a partial-match indexing scheme. This scheme, however, does not permit databases to grow. An alternative indexing scheme under which ground terms may be stored in databases which can grow indefinitely is also provided. NU-Prolog can also access and modify data stored in UNIFY† databases via an interface which transforms Prolog goals into SQL queries.

Chapter 2 introduces the principal software tools that comprise the NU-Prolog system. These are the NU-Prolog compiler; an interpreter for testing and debugging programs and for handling interactive database queries; a program checker which scans NU-Prolog source for probable errors; a preprocessor for adding control to NU-Prolog programs; and an incremental program revision facility.

The syntax of NU-Prolog is described in Chapter 3. Chapter 4 describes the semantics of the principal constructs of NU-Prolog This chapter describes many of the novel features of the language, including advanced negation facilities and extended syntax. It is recommended that this chapter be read by all NU-Prolog programmers.

Chapter 5 describes the NU-Prolog system predicates. These includes the NU-Prolog external database schemes, debugging facilities, and a substantial number of predicates used for handling I/O, lists, terms and the internal NU-Prolog database. NU-Prolog libraries are described in Chapter 6. These include a library for the porting of Quintus Prolog and DEC-10 Prolog programs.

Appendix 1 contains programs illustrating NU-Prolog and demonstrating the use of **`when`** declarations and declarative aggregate constructs. The definite clause grammar notation is described in Appendix 2. Appendix 3 gives an example of the use of the NU-Prolog compiler.

---

† UNIFY is a trademark of the UNIFY Corporation.

Appendix 4 describes the principle changes required in porting MU-Prolog programs to NU-Prolog. Appendix 5 gives an example of the creation and use of a dsimc database. Appendix 6 lists predicate names that have been reserved for future use. Appendix 7 defines the classes of ASCII characters used by NU-Prolog. Appendix 8 gives the operator declarations used by NU-Prolog. Signals are listed in Appendix 9. Appendix 10 outlines some bugs and limitations of NU-Prolog.

This document is intended to be a reference manual, not a text on writing Prolog programs. Both novice and experienced Prolog programmers should find this manual an essential and convenient reference for programming in NU-Prolog.

CHAPTER 2

USING NU-PROLOG

NU-Prolog is a compiled Prolog system. §2.2 describes *nc*, the NU-Prolog compiler; its behaviour is analogous to *cc* (1), the compiler for the *C* programming language, and it has similar options. An interpreter-like environment, *np*, is provided for NU-Prolog as an aid in debugging and development, as described in §2.3. The NU-Prolog system also includes *nit*, a program checker which identifies probable bugs in NU-Prolog programs. This is described in §2.4. The preprocessor *nac*, described in §2.5, is used to add control to NU-Prolog programs. The other NU-Prolog facility described in this chapter is *revise*, an incremental program revision facility, described in §2.6.

In NU-Prolog, the program which is being executed is essentially fixed at compile time. It is translated into machine code for an abstract Prolog machine based on the abstract machine described in [Warr83]. Unlike an interpreter-based system, the compiled predicates are not stored as Prolog structures in the internal database. Since predicates are not stored in the same format as other data structures, the programmer does not have the same freedom to manipulate and define predicates and clauses as in interpreted Prolog systems. If NU-Prolog clauses are to be added or deleted at run time, the predicates of which they are part should be declared as **`dynamic/1`**. However, a dynamic predicate is not executed as efficiently as the compiled equivalent. Alternatively, facts and rules may be stored using the external database facilities.

2.1. Getting Started

Developing a program using the NU-Prolog system usually involves the following steps.

(1) Editing a file, say *prog.nl*, containing the source of the program. The source might be:

```
parent(fred, mary).
parent(fred, bill).
parent(mary, sally).

ancestor(X, Y) :-
      parent(X, Y).
ancestor(X, Y) :-
      parent(X, Z),
      ancestor(Z, Y).
```

(2) Compiling the program using *nc* with the *−c* option:

```
nc -c prog.nl
```

which will generate two files:

prog.no – abstract machine object code.
prog.ns – assembler source for the abstract machine.

(3) To use the program, type:

```
np
```

to invoke the interpreter. The user will be given the prompt:

```
1?-
```

and can load the compiled program with the command

```
[prog].
```

This could have been done in one step with **np prog.no**. After the message

```
loading prog.no
```

the system will give the prompt

```
2?-
```

to which the user can type queries such as

```
ancestor(fred, X).
```

or

```
X : ancestor(fred, X).
```

The former finds the first *X* for which **fred** is an ancestor, and finds further values for *X* on request; the latter will retrieve all such *X*.

NU-Prolog programs may contain a predicate **main/1** to be invoked when the program is executed from shell. Any command line arguments are given to **main/1** as a list of atoms, just as command line arguments are given to the *C* language function *main* as an array of strings. For example, suppose that the UNIX directory /**db** contained student records of the form **student(LoginID, Surname)**. The command

```
nc -o names names.nl
```

could then be used to compile the following program into the executable file *names*.

```
?- dbCons('/db').

main(_.Surnames) :-
      solutions(ID, (member(SN, Surnames), student(ID, SN)), IDs ),
      printList(IDs).
printList([]).
printList(A.B) :-
      writeln(A),
      printList(B).
```

The program will print every *ID* where **member(SN,Surnames),student(ID,SN)** is true, and the goal **?- dbCons('/db')** tells NU-Prolog which database to check to find **student/2** records. So, the command

```
names Smith Jones Bloggs
```

would print the login ID of every student whose surname is Smith, Jones or Bloggs.

All the descriptions of predicates and connectives in this manual are available in the predicate **man/1**. It is described in §5.3.

The remaining sections in this chapter describe some of the UNIX commands which comprise the NU-Prolog system. These commands are *nc* (described in §2.2), *np* (§2.3), *nit* (§2.4), *nac* (§2.5), *revise* (§2.6) and *initdb* (§5.8).

2.2. *nc* – NU-Prolog Compiler

The NU-Prolog compiler is *nc*, which accepts several types of source files as input. Arguments whose names end with **.nl** are assumed to be NU-Prolog source programs. Each **.nl** file is compiled into an assembler file, whose name is that of the source with **.ns** substituted for **.nl**. Any **.ns** files are assembled to produce object files, whose suffix is **.no**. Most of the compile time is spent producing **.ns** files, so if space is to be saved it is cheapest to remove **.no** files.

If a program is to be executed from shell, compilation produces an executable image – **a.out** by default – which incorporates the predicate definitions from all of the specified **.no** files. If a predicate definition is split across a number of files, a warning will be printed and the only part of the definition to be used will be that which occurs in the last file to be loaded. An ‘‘entry predicate’’ with arity of one can be defined in one of the **.no** files, The name of the entry predicate can be specified with the **–e** option; by default, it is **main**/**1**. This will be used as the entry point into the executable file. That is, when **a.out** is executed, the program will be run with the goal **main**/**1**, where **main** is the name of the entry predicate. The argument to **main**/**1** will be a list of atoms representing the command line arguments; the first element in the list is always the name of the executable file. There is a default definition of **main**/**1** which enters the top level of the interpreter when the program is invoked.

NU-Prolog executes all goals in **.nl** files twice, once at compile time and once at load time. Goals are executed by the compiler when encountered in the program source, and are also placed in a initialization predicate which is invoked immediately before the executable file is created, that is, at load time. This is because some goals – such as **useIf**/**1** and **dynamic**/**1** – are used during compilation, and others – such as user-defined predicates – when the executable file is created.

NU-Prolog starts with default amounts of memory pre-allocated for its various data areas. These can be expanded at run-time, but expansion of the permanent data-area usually blocks further expansion of the others. Larger, or smaller, initial allocations can be made with the **–u**, **–v**, **–w**, and **–x** options.

nc may be used as follows.

> **nc [-c] [-D] [-S] [-U] [-o** *objfile* **] [-e** *entry_pred* **] [-F** *flag* **]**
> **[-u** *n* **] [-v** *n* **] [-w** *n* **] [-x** *n* **]**
> **[-l** *library* **] *.nl *.ns *.no**

The options are used as follows.

–c Compile only, do not link. That is, stop the compilation when the **.no** files have been created; this is used for programs which are to be executed from the interpreter.

–D Apply the definite clause grammar preprocessor to the given files and omit the processed code on standard output. Definite clause grammars are described in Appendix 2.

–e *entry_predicate*
Specify the name of the predicate with arity of one to be used as the goal when the executable file is run. This defaults to **main**/**1**. It is an error for the entry predicate to be absent from the program.

–F *flag*
Make **option(flag)** true during compile time. Useful for conditional compilation of clauses with **useIf**/**1**.

–l *library*
Specify a library to be loaded with the executable image. The predicates defined in the library will be incorporated into the program. Since using this option is quicker it should be used in preference to embedding **?– lib library** goals in the **.nl** files.

-o *object_file*
Place the executable image in *object_file* rather than the default **a.out**.

-S Compile the named NU-Prolog programs and leave the assembly-language output on corresponding files suffixed **.ns**. No **.no** files are produced.

-U Translate calls to UNIFY databases into a more efficient form before compilation. Described in more detail in §5.8.

-u *n*
Ensure that there is at least *n* kwords of space available for permanent data items when the compiled program is loaded. These include compiled and interpreted code, properties, IO and database buffers, and the symbol table. A word will be 4 bytes on most systems, but will store a similar amount of data on all. Filling this area during compilation is likely to prevent further expansion of the Prolog stacks. This option is meaningful only when an executable image is being produced.

-v *n*
Ensure that there is at least *n* kwords of space available on the Prolog heap when the compiled program is loaded. This contains Prolog's global execution data – mainly holding terms, floats, and information about delayed calls. A word will be 4 bytes on most systems, but will store a similar amount of data on all. Prolog can usually expand this if necessary, unless the permanent data area has overflowed its initial allocation. This option is meaningful only when an executable image is being produced.

-w *n*
Ensure that there is at least *n* kwords of space available on the Prolog local stack when the compiled program is loaded. This contains choice-points and environments. A word will be 4 bytes on most systems, but will store a similar amount of data on all. Prolog can usually expand this if necessary, unless the permanent data area has overflowed its initial allocation. This option is meaningful only when an executable image is being produced.

-x *n*
Ensure that there is at least *n* kwords of space available on the Prolog trail when the compiled program is loaded. This contains information on variable bindings to be reset on backtracking. A word will be 4 bytes on most systems, but will store a similar amount of data on all. Prolog can usually expand this if necessary, unless the permanent data area has overflowed its initial allocation. This option is meaningful only when an executable image is being produced.

An example of the use of *nc* is given in Appendix 3.

2.3. *np* − NU-Prolog Interpreter

Compiled NU-Prolog programs may be run by generating executable files or by using *np*, the NU-Prolog interpreter. To use *np*, type:

np [*object_file* **...]**

where *object_file* is an object file, that is, a file with **.no** suffix, which will be loaded into *np*. Object files may also be loaded with **load/1** and similar predicates. *Np* can also interpret NU-Prolog programs directly without compiling them first. This is much slower and uses far more memory than executing compiled code, but it may shorten the edit-compile-run cycle. To load a file for interpreted execution, **consult/1** it. Except when loaded with **consult/1**, NU-Prolog will choose the compiled **.no** version of a file in preference to the **.nl** source.

Np provides an interactive system which reads and calls one goal at a time. A session with *np* is terminated either by end-of-file (type ^**D** as the first character on the line) or by a call to the **exit/1** predicate.

If a goal read by *np* contains a syntax error, an error message will be printed and *np* will reprompt for another goal to be entered. *Np* uses a mechanism similar to **read1/1** to fetch goals, which it reads as NU-Prolog terms and thus requires that commands be terminated with a '.' and whitespace.

Np incorporates a history mechanism. Each '**?-**' goal, whether syntactically correct or not, is numbered, and may be repeated, listed or modified with the following history list commands, which are not themselves added to the history list.

?− N

Repeat the goal numbered *N*. If −*N* is chosen, the goal given by subtracting *N* from the current prompt number is chosen.

Only available at the top level of the interpreter.

?− e
?− e N

Load the most recent (or N^{th}) goal in the history list into the editor specified by the environment variable VISUAL (or EDITOR or *vi* in line mode). If *N* is negative, the goal given by adding *N* to the current prompt number is chosen. When the editor exits, the new goal is added to the history list and executed.

Only available at the top level of the interpreter.

?− h

Print history list.

Only available at the top level of the interpreter.

?− r Hist

Hist is the UNIX filename of a saved history list to be loaded into *np*. If the current prompt number is *n* and there are *m* commands in *Hist*, the commands in *Hist* are stored in the history list against the numbers n to $n+m-1$.

Only available in at the top-level of the interpreter.

?− s Hist

Hist is the UNIX filename into which the saved history list is to be saved.

Only available in at the top-level of the interpreter.

The following commands will be executed by *np* and stored in the history list. Of particular use in making queries on databases are the constructs described in §4.2, such as **all/2**, **some/2**, **count/3**, **max/3**, **min/3** and **sum/4**.

?− Goal

The formula *Goal* is interpreted as an ordinary NU-Prolog goal, When *Goal* is solved, any bindings made to free variables in *Goal* are displayed. To get alternative solutions to *Goal*, type '**;**' (semi-colon); otherwise type 'return' to display the prompt. If there were no free variables in *Goal* and the goal succeeds **true** will be printed. If *Goal* fails, or there are no further solutions **fail** will be printed.

?− : Goal
?− Vars : Goal
?− Vars sorted : Goal
?− Vars sorted Keys ... : Goal

Goal is pure (see **pure/1**) and therefore may only contain logical predicates and connectives. All solutions for the variables in the term *Vars* are printed out. If *Vars* is absent, either **Yes** (there is an answer) or **No** (no answers) is printed.

If the keyword **sorted** is included, the solutions are sorted using **sort/2**. If any optional *Keys* are specified, the solutions are sorted with **multiKeySort/4**.

Only available at the top level of the interpreter.

?− delete Predicates where Goal

Goal is pure (see **pure/1**) and therefore may only contain logical predicates and connectives. For each solution to the conjunction of *Goal* with the elements of *Predicates*, every clause matching an element of *Predicates* is retracted. *Predicates* is one of a single term, (**Predicate**), a list of terms (**[**$\textbf{Predicate}_1$, **...]**), or a comma separated list of terms ($\textbf{Predicate}_1$, · · ·). The conjunction of *Goal* with the elements of *Predicates* must ground each element of *Predicates*.

Only available at the top level of the interpreter. Some examples are given in Appendix 1.

?− insert Predicates where Goal

Goal is pure (see **pure/1**) and therefore may only contain logical predicates and connectives. For each solution to the conjunction of *Goal* with the elements of *Predicates*, each element of *Predicates* is asserted. *Predicates* is one of a single term, (**Predicate**), a list of terms (**[**$\textbf{Predicate}_1$, **...]**), or a comma separated list of terms ($\textbf{Predicate}_1$, · · ·). The conjunction of *Goal* with the elements of *Predicates* must ground each element of *Predicates*.

Only available at the top level of the interpreter. Some examples are given in Appendix 1.

?− update Var$_1$ to Term$_1$, ... in Predicates where Goal

Goal is pure (see **`pure/1`**) and therefore may only contain logical predicates and connectives. For each solution to the conjunction of *Goal* with the elements of *Predicates*, every clause matching an element of *Predicates* is retracted, *Var*$_1$,... is replaced by *Term*$_1$,... and the resulting predicates are asserted. *Predicates* is one of a single term, (**`Predicate`**), a list of terms (**`[Predicate`**$_1$**`, ...]`**), or a comma separated list of terms (**`Predicate`**$_1$**`, ...`**). The conjunction of *Goal* with the elements of *Predicates* must ground *Var*$_1$,..., *Term*$_1$,... and each element of *Predicates*.

Only available at the top level of the interpreter. Some examples are given in Appendix 1.

2.4. *nit* – NU-Prolog Program Checker

Nit † is a program which identifies probable bugs in NU-Prolog programs; it is analogous to *lint* (1). *Nit* will check for the following classes of (potential) errors, predicates with the same functor and different arity, that all defined predicates are called and all called predicates defined, that pure predicates (see **pure/1**) contain no non-logical subgoals, that variables are only used within their defined scope, that variables whose names don't begin with '_' occur more than once, that variables do not have confusing names, for clauses in which variables in negations cannot be ground, and for programs which are not stratified. These classes are described later in this section.

Goals of the form **?- dynamic(Functor/Arity)** tell *nit* that *Functor* with *Arity* is defined. If the goal **?- dbCons(DB)** occurs, *nit* will consider all predicates in external database *DB* to be defined. The goals **?- useIf Goal**, **?- useElse** and **?- useEnd** may be used to mark conditional examination of code. *Nit* also will recognise goals of the form **?- lib lib_name** as being equivalent to **-l lib_name** being specified as an argument to *nit*. *Nit* will decompose '**?-**' goals over '**,**' and '**;**', but does not attempt to parse more complex structures.

Several types of arguments are accepted by *nit*. Arguments which are not an option as described below are assumed to be NU-Prolog source programs, where the argument '**-**' (hyphen) represents standard input (**user**). If no NU-Prolog source files are given, *nit* will read from standard input. *Nit* has the following usage.

nit [-apsvx] [-d *Functor/Arity* **] [-e** *Functor/Arity* **] [-F** *flag* **]**
[-l *library* **] [-u** *Functor/Arity* **]** *files* **...**

The options are used as follows.

-a Turn off checking for predicates with the same functor and different arity.

-d *Functor/Arity*
Functor with *Arity* is considered to be defined (although its definition may be absent from the source).

-e *entry_predicate*
(As in *nc*). Specify the name of the single-argument entry predicate in the NU-Prolog source. This defaults to the predicate **main/1**.

-F *flag*
(As in *nc*). Specify that **option(flag)** be true. Useful in conjunction with **useIf/1**.

-l *library*
(As in *nc*). Specify the name of a NU-Prolog library which will be loaded with the NU-Prolog executable image. In effect, this turns off ''undefined predicate'' errors for library predicates. *Nit* will recognise goals of the form **?- lib lib_name** in NU-Prolog source as being equivalent to **-l lib_name** being specified.

-p Turn off checking pure predicates with non-logical features.

-s Turn off checking for stratifiability of programs.

-u *Functor/Arity*
Functor with *Arity* is considered to be called, although there may be no call to this predicate in the source.

-v Turn off checks on variables. These include checks on naming, scope and use of variables.

-x Turn off checking for predicates which are used but not defined, or defined but not used.

† *nit* is an acronym for ''NU-Prolog incompetence tester''

The errors reported by *nit* fall into two groups, those which are part of the program as a whole, and errors within an individual clause; in this case, the error message is printed with the clause to which it belongs. The error messages, and the classes to which they belong, are described below.

Arity check

Nit will report when two predicates have the same functor and different arity. Although this may not be an error, such code can be obscure and should be avoided, particularly if both arities are large.

`Functor has arities X and Y.`
Two predicates have the same functor and the given arities.

Cross-checking of predicate definitions

Nit will report when predicates are called but not defined, or defined but not called.

`Functor/Arity is modified but not declared dynamic.`
`Functor`/**`Arity`** should be declared as **`dynamic`**/**`1`**. If there are some initial clauses for **`Functor`**/**`Arity`**, the behaviour of the program is not defined.

`Functor/Arity is modified but not called.`
The predicate with **`Functor`** and **`Arity`** is modified, via **`assert`**/**`1`** or a similar predicate, but never called.

`Functor/Arity is not called.`
The predicate with **`Functor`** and **`Arity`** is defined but never called.

`Functor/Arity is not defined.`
The given clause calls a predicate with *Functor* and *Arity* which is not defined.

Pure predicate definitions with non-logical features

Predicates declared to be **`pure`**/**`1`** should not call system predicates which are non-logical.

`Nonlogical predicate Functor/Arity called by pure predicate.`
The given clause uses the non-logical construct or predicate with *Functor* and *Arity*.

Scope of variables

If the same variable name occurs in separate scopes in a clause, it refers to different variables, and therefore should be given different names. Scopes are defined by quantifiers.

`V occurs in distinct scopes.`
The variable *V* occurs in separate scopes in the given clause, and NU-Prolog will treat it as two separate variables. For clarity, different names should be given to variables in distinct scopes.

Uniquely occurring variables

If a variable occurs once only in a clause, its name should (for clarity) be prefixed with '_' (underscore). Conversely, variables whose names are prefixed with '_' should occur once only in a clause.

$[V_1, \ldots, V_n]$ `should occur once only.`
The names of the variables $V_1, \ldots, V_n$ are prefixed with '_' (underscore), but these variables occur more than once in the given clause.

`The variables` $[V_1, \ldots, V_n]$ `occur once only.`
The variables $V_1, \ldots, V_n$ occur once only in the given clause, and should therefore

have their names prefixed with '_' (underscore).

Naming of variables

When the names of two variables in a clause are identical except for the case of some characters and the placing of underscores, the clause may not only be obscure, but usually contains a spelling error.

V_i and V_j are suspiciously similar.

The variables V_i and V_j have names which suggest that they are intended to be identical.

Clauses in which variables in negations cannot be ground

A clause is unsafe or will flounder if some variables occur more than once and do not occur positively. (Variables which only occur once in a clause are always locally existentially quantified.) Non-positive occurrences of variables are within the condition of **if** or **->**, **not/1**, **\+**, the left hand side of **=>**, the right hand side of **<=**, or either side of **<=>**.

V cannot be ground.

V cannot be ground by the given clause, so that the clause is unsafe or will flounder. If *V* only occurs once in the clause, its name should be prefixed with '_' (underscore); if it occurs more than once, it should be given a positive occurrence.

Nonstratifiability of programs

A program is stratified when, in each instance of recursion, including any mutual recursion, there are no non-positive or all-solutions predicates. This condition is described as ''negative mutual recursion''. It is, in general, safer for a program to be stratified. Non-positive predicates include **not/1**, **\+**, the left hand side of **=>** or the right hand side of **<=**, either side of **<=>**, or the condition of **if** or **->**. All-solutions predicates include **all/2**, **gAll/2**, **solutions/3**, **setof/3**, **bagof/3**, **count/3**, **min/3**, **max/3** and **sum/4**.

[F_1/A_1, . . . ,F_n/A_n] contains negative recursion.

The predicates with functors F_1 to F_n and arities A_1 to A_n respectively have mutual negative recursion, and therefore the program is not stratified.

2.5. *nac* – NU-Prolog control preprocessor

nac† is a preprocessor for NU-Prolog programs which adds control information [Nais85]. This can be very useful for programs which are written purely declaratively. Declarative programming is simpler, since the programmer needs only to consider the logic, but with naive control tends to result in inefficiency and infinite loops. Many predicates work correctly for only some input/output modes. By using *nac* to automatically add control information, program development time can be reduced.

nac is designed for code which manipulates recursive data structures, such as lists and trees. It does not work as well for database and numerical programming. To avoid overriding user-defined control, *nac* only adds control to predicates which are declared **pure** and do not already have **when** declarations. As well as adding control information, *nac* produces some comments at the start of the output, to indicate what has been done and warn of possible errors. Unfortunately, other comments in the input are stripped. *nac* also expands DCG grammar rules (see appendix 2).

The control additions made by *nac* fall into two categories. The first is the insertion of **when** declarations. These are used to delay calls to predicates until they are sufficiently instantiated to run correctly. The algorithm for generating when declarations is based mainly on prevention of infinite loops. Potential infinite recursions in the program are analysed and **when** declarations are added to prevent them. If there is a potential loop that cannot be prevented by addition of **when** declarations, a warning is printed. If a program has an infinite loop, there is a very good chance that *nac* will be able to eliminate or at least identify it. The **when** declarations also make many predicates work correctly for more input/output modes, where this is applicable.

The second way *nac* adds control information is the reordering of calls in the bodies of clauses. The main rules for reordering are to call tests and deterministic procedures early and avoid left recursion (which often results in infinite loops). Calling tests before predicates which generate bindings for variables can increase efficiency by coroutining. Failure is detected as soon as possible, so unnecessary computation is avoided. Most system predicates are deterministic and many are tests. User-defined predicates which are deterministic or likely to be tests can be detected by analysing **when** declarations.

Example

Suppose that the file queenl.nl contained the following program, which solves eight queens problem:

```
        % logic of 8-queens problem

?- pure queen/1.
queen(X) :- permute(1.2.3.4.5.6.7.8.[], X), safe(X).

?- pure permute/2.
permute(X.Y, U.V) :- permute(Z, V), delete1(U, X.Y, Z).
permute([], []).

?- pure delete1/3.
delete1(A, A.L, L).
delete1(X, A.B.L, A.R) :- delete1(X, B.L, R).

?- pure safe/1.
```

† Could be an acronym for ‘‘not another command’’, but some people think it stands for ‘‘NU-Prolog addition of control’’.

```
safe([]).
safe(N.L) :- safe(L), nodiag(N, 1, L).

?- pure nodiag/3.
nodiag(_, _, []).
nodiag(B, D, N.L) :-
      D =\= N-B,
      D =\= B-N,
      D1 is D + 1,
      nodiag(B, D1, L).
```

If this program is run as it stands, the first call to **perm/2** results in an infinite loop. Even if this problem were avoided, the program would be very inefficient; an entire permutation of the numbers 1 to 8 is generated before it is tested for safety. After typing the command:

```
nac < queenl.nl > queenc.nl
```

the file queenc.nl would contain (approximately) the following:

```
% the following code has been nac'ed
%
% procedure queen/1 is locally deterministic
% procedure permute/2 is locally deterministic
% procedure safe/1 doesnt construct some arg(s)
% procedure safe/1 is locally deterministic
% procedure nodiag/3 doesnt construct some arg(s)
% procedure nodiag/3 is locally deterministic
% procedure safe/1 is deterministic
% procedure nodiag/3 is deterministic
% clause altered: queen(A) :-  . . .
% clause altered: permute(A.B, C.D) :-  . . .
% clause altered: safe(A.B) :-  . . .

?- pure queen/1.
queen(A) :-
      safe(A),
      permute(1.2.3.4.5.6.7.8.[], A).

?- pure permute/2.
?- permute(A, B) when A or B.
permute(A.B, C.D) :-
      delete1(C, A.B, E),
      permute(E, D).
permute([], []).

?- pure delete1/3.
?- delete1(A, B.C, D) when C or D.
delete1(A, A.B, B).
delete1(A, B.C.D, B.E) :-
      delete1(A, C.D, E).

?- pure safe/1.
?- safe(A) when A.
safe([]).
```

```
safe(A.B) :-
      nodiag(A, 1, B),
      safe(B).

?- pure nodiag/3.
?- nodiag(A, B, C) when C.
nodiag(A, B, []).
nodiag(A, B, C.D) :-
      B =\= C - A,
      B =\= A - C,
      E is B + 1,
      nodiag(A, E, D).
```

The **when** declarations make calls delay where they might start an infinite loop. This also allows **safe**/**1** to be called before **perm**/**2**, resulting in coroutining between the test and generator. Initially, **safe**/**1** delays, but whenever **perm**/**2** and **delete**/**2** further instantiate the list of queen positions, **safe**/**1** and **nodiag**/**3** are resumed and check that the new position is safe.

The comments produced indicate that **queen**/**1**, **perm**/**2** and **nodiag**/**3** are locally nondeterministic (calls to these procedures can only match with one clause); some argument(s) to **safe**/**1** and **nodiag**/**3** are never constructed (so they must be input); **safe**/**1** and **nodiag**/**3** are deterministic (the entire computation of these calls does not create any choice points); and the bodies of the rules for **queen**/**1**, **perm**/**2** and **safe**/**1** have been reordered.

2.6. *revise* – Incremental Program Revision Facility

Revise is an incremental program revision facility for NU-Prolog programs, available from UNIX and from within NU-Prolog. The precise usage is given below.

During a *revise* session, only the parts of the program that have been changed are recompiled. That is, new **`*.ns`** files are created from the **`*.nl`** source, and the **`*.no`** files – which are relatively cheap to recreate – are removed; if *revise* is used from within NU-Prolog, the **`*.no`** file representing the changed predicates is loaded into NU-Prolog when *revise* exits.

Revise recognises changed code by comparing the new source file **`*.nl`** with a backup copy of the source file **`*.nl@`**. Each source file is divided by delimiters into blocks; when any part of a block is changed, the whole block is recompiled, along with any code occurring before the first delimiter in that file. Any operator declarations or other goals needed at compile time should therefore be placed before the first delimiter, so that they are used with each recompiled block.

Predicate definitions must not be split across blocks; otherwise only part of the predicate definition will be used if it is modified. Goals (other than when declarations) are treated as part of a special predicate, so *revise* will not include them in the modified **`*.no`** file. If goals needed at load time are changed, the whole file should be recompiled with *nc* in the standard way or the **`*.ns`** file should be removed (forcing complete recompilation).

The default block delimiter is a blank line. If you like to have blank lines between clauses of a predicate, you may change the delimiter by making

```
%tag <delim>
```

the first line of a source file. **`%<delim>`** is then taken as the delimiter throughout that file. The sequence **`%<delim>`** must occur at the start of the line and be followed by white space. If the delimiter is not a blank line, blocks may be named:[1]

```
%<delim> <block> comment ...
```

`<block>` is the name of the block, and anything following the string is considered a comment. Block names can be used as **`vi`** tags.

The revise system consists of three commands: *redit*, *rnc* and *revise*. Running *redit* is the same as running *vi* (or whatever editor is specified in the environment variable VISUAL or EDITOR), except that before a **`*.nl`** file is edited, it is copied to a backup **`*.nl@`** file if a backup does not already exist. It is also possible to create the backup files once using *cp*, then use your normal editor directly.

The *rnc* command is essentially the same as the *nc* command, but only the modified parts of files are recompiled if the appropriate **`*.nl@`** and **`*.ns`** files exist, and the backup **`*.nl@`** files are updated (assuming compilation is sucessful). The **`-F`**, **`-D`** and **`-U`** options are not supported by *rnc*. If a **`*.nl@`** file is an argument *rnc*, the compiled form of the modified blocks is placed in **`*@.no`** as well as **`*.no`** being updated.

The *revise* command is the same as *redit* followed by *rnc -c*, with **`*.nl@`** files specified. For example, suppose you had the files **`f.nl`**, **`f.ns`** and **`f.no`**, and ran the command **`revise f.nl`**. The first step would be the execution of **`redit f.nl`**, which would copy **`f.nl`** to **`f.nl@`** then run the editor. After changing the file and quitting the editor, **`rnc -c f.nl@`** would be run. This compares **`f.nl`** and **`f.nl@`** to find what blocks have been modified. These blocks, and the first block of the file, are compiled (they are put in file **`f@.nl`**) to produce assembly code (in **`f@.ns`**) and **`f@.no`**. The new assembly code is used to update **`f.ns`** which is then assembled to get the updated **`f.no`**. Finally, **`f.nl`** is copied to **`f.nl@`**.

---

[1]This is known as the ''. . . but never jam today feature.''

From within NU-Prolog, the revise system can be used as follows:

revise
revise(File)
revise(Pred)

(Non-logical).
Edit and recompile part of program. Invokes the user's preferred text editor (*vi* by default) to allow modification of the source file for a loaded file. If *File* (a term of the form **`F.nl`**) is specified, the editor is invoked with that file. If *Pred* (an atom) is specified, the editor is invoked with the file containing the definition of the predicate **`Pred`**/**`N`**, for some **`N`**. If **`revise`**/**`0`** is used, the most recently loaded (or revised) file is edited.

When the editor is exited, code in any modified blocks of the file will be recompiled, the **`.ns`**, **`.no`** and backup **`.nl@`** files updated, and the modified code reloaded. See §2.6.

CHAPTER 3

SYNTAX OF NU-PROLOG

This chapter provides a description of the syntax of NU-Prolog and definitions for the terminology used throughout this manual.

3.1. Variables

A **variable** is a sequence of characters, which may be alphanumeric, '**$**' or '_' (underscore), beginning with an upper-case letter or '_'. Case is significant in variable names. Classes of characters are described in Appendix 7.

The variable named '_' is used to identify anonymous variables; all instances of '_' denote distinct anonymous variables. Variables whose names begin with '_' (and contain more than one character) are not anonymous, but *nit* treats such variables occurring more than once in a clause as an error. These variables are used as place holders when the bindings in that position are not of interest. The normal scope of a variable is over the clause containing that variable; **all**/**2**, **some**/**2**, **count**/**3**, **max**/**3**, **min**/**3**, **solutions**/**3** and **sum**/**4** may be used to restrict scope. If the same variable name occurs in distinct scopes, it refers to distinct variables. For example, in the clause fragment

```
... p(X), some X q(X), r(X),...
```

the occurrences of *X* in **some X q(X)** are a distinct variable from the other occurrences.

3.2. Constants

A **constant** is an atom, an integer, a float, or any of several internal types. Other types may be added in the future.

3.2.1. Atoms

An **atom** is:

- a sequence of characters, which may be alphanumeric, '**$**' or '_', beginning with a lower-case letter or '**$**';
- or a sequence of symbol characters
- or any of one of '**!**', '**,**', '**;**', '**[]**', or '**{}**';
- or any sequence of permitted characters enclosed in single quotes. Two adjacent single quotes inside a quoted atom are transformed into a single quote, as is the escape sequence **\'**.

Characters classes are described in Appendix 7.

3.2.2. Integers

An **integer** is either:

(1) A sequence of decimal digits.

(2) An expression of the form *B*'*N* where *B* is a decimal base in the range 1 to 36, and *N* is a sequence of the digits which may appear in base *B*. If *B* is greater than or equal to 10, the letters *A* to *Z* (or *a* to *z*) are used to represent the base. If *B* is less than 10, the base is represented by a decimal digit.

(3) An expression of the form **0'Char**, which is equivalent to giving the ASCII code for *Char*, where *Char* is alphanumeric, a symbol character or an escape sequence, as defined in Appendix 7. For the purposes of input and output, characters are represented in NU-Prolog by their ASCII code.

An integer may be immediately preceded by a '+' or a '**-**'. Note that **-1** and **- 1** are different; the first is the integer negative one and the second is the atom **-** followed by the integer **1**. Depending on context and operator declarations (see §3.8) this is likely to be parsed as the complex term (see §3.3) with functor minus and argument the integer one.

The range of representable integers is implementation dependent, but is guaranteed to include **- 2^25** and **2^25 - 1**. Releases of NU-Prolog later than 1.5 support 32 bit integers, but, like floats, these take up heap space. The maximum and minimum integers supported can be found with **X is maxint** and **X is minint**.

3.2.3. Floats

A floating-point number (**float**) is either:

(1) A sequence of decimal digits followed by a '**.**' and another sequence of decimal digits. Both sequences must be non-empty. Optionally, this form of floating-point representation may be followed by an '**e**' or '**E**' and an optionally signed decimal exponent.

(2) A sequence of decimal digits followed by an '**e**' or '**E**' and an optionally signed decimal exponent.

A float may be immediately preceded by a '+' or a '**-**' in the same way as an integer.

Floating point numbers are stored in the same heap as terms.

NU-Prolog uses the floating-point representation of the machine it is running on. It is worth noting that some machines' floating-point arithmetics reserve some values to indicate various error and exception conditions. A common example is the IEEE 754 floating-point standard. Generally, NU-Prolog cannot accept these special values as input, but may print them as output. The textual representation of any special floating-point values should be listed in the machine’s manual entry for the C stdio routine printf().

3.3. Terms

A **term** may be defined recursively as follows:

(1) A variable is a term.

(2) A constant is a term.

(3) If **f** is an atom and $t_1,...,t_n$ are terms, then $f(t_1,...,t_n)$ is a term.

The **arity** of $f(t_1,...,t_n)$ is **n**; **f** is its **functor**. The maximum possible arity is system dependent, but is guaranteed to be at least 63. In version 1.4 and later the maximum is determined at installation time and will be at least 1024.†

On input, there can be no white space between the functor and the opening parenthesis of a term. '**.**' (period) followed by white space is always interpreted as the end of a term.

3.4. Clauses

A **clause** is an expression of the form

$$A \;{:}{-}\; B_1,...,B_n.$$

where **A** and all B_j are terms which are not integers. **A** is the **head** and $B_1,...,B_n$ is the **body**. If **f** and **n** are the functor and arity of **A**, then **A :- ...** is a clause **about f/n**. A **unit clause** is a clause with an empty body, and is written

A.

A **fact** is a ground unit clause. A **rule** is a clause with non-empty head and non-empty body. A **goal** is a clause with an empty head, and is written

$$?{-}\; B_1,...,B_n.$$

'**?-**' is omitted from goals which occur within a clause, and is provided at the command level of *np*. A goal is a statement of what is to be proved.

3.5. Predicates

A **predicate** definition is a set of clauses with heads with the same functor, **f**, and arity, **n**, and is referred to as **f**/**n**. The maximum arity for predicates is the same as that of functors. The maximum arity of predicates stored in an external database is 32. The functors of many system predicates contain the character '**$**', and this character should be avoided in the functors of user-defined predicates. Some predicate names which have been reserved for future use are listed in Appendix 6.

---

†The compiler limits the arity of predicates to about 250 and may also be unable to compile clauses with terms of arity greater than this. This limit is due to the availability of registers and is unlikely to change.

3.6. Lists

There are two separate notations for representing lists. The first uses '**.**' (period) as the list construction operator (pronounced **cons**). In the term **Head.Tail** (equivalent to **.(Head, Tail)**, or **[Head|Tail]** in the second notation), **Head** refers to the first element in the list and **Tail** the remainder of the list.

The character **.** is used for a number of syntactic purposes. These are sometimes incompatible with its use as an operator. On input, there may be no white space after '**.**' (cons) because this sequence is interpreted as the end of the term being read; the characters **1.2.3** will parse as the cons of the float **1.2** and the integer **3**; and +**.**+ parses as one atom with the print name ''+.+''.

For example, a list consisting of the constants **a**, **b** and **c** would be **a.b.c.[]** (or **[a, b, c]**), where the atom **[]** (pronounced **nil**) is used to represent the empty list. Similarly, a list of indeterminate length beginning with the constants **a** and **b** would be represented by **a.b.Y** (or **[a, b| Y]**).

3.7. Strings

A **string** is a sequence of permitted characters enclosed by double quotes. It is syntactic sugar for a list of the ASCII codes of the sequence of characters. The ASCII code for a character *Char* is an integer (see section 3.2.2.) Two double quotes inside a string, or the escape sequence **\"**, are converted into one double quote. Permitted characters are listed in Appendix 7. Lists are usually printed as strings if every element of the list is an integer for which **isPrint**/**1** is true.

Note that the empty string is the empty list, that is, **[]**.

Internally, strings may be stored in two ways, either in a compact form, or as an ordinary list of ASCII codes. The difference between these two forms is invisible to the user, because a string and a list consisting of the same character sequence will unify. For example, the string

```
"Hello"
```

is equivalent to the list

```
0'H.0'e.0'l.0'l.0'o.[]
```

or

```
[72, 101, 108, 108, 111]
```

3.8. Operators

Operators provide an alternative and convenient syntax for terms. An operator is a functor with one or two arguments, which can appear in any of prefix, infix or postfix position relative to its arguments. For example, **x + y** is equivalent to (and more readable than) the term +**(x, y)**.

Unary operators can be postfix (type **xf** or **yf**) or prefix (**fx** or **fy**). Binary operators can be infix (**xfx**, **xfy**, **yfx** or **yfy**) or prefix (**fxx**, **fxy**, **fyx** or **fyy**). It is not possible to unambiguously parse a language with both postfix and prefix binary operators, so postfix binary operators are not provided. In operator types, **f** represents the position of the operator, and **x** and **y** represent the associativity of the arguments. Assuming no brackets, an **x** indicates that the top-level operator in that argument must be of a strictly lower precedence than the operator, whereas a **y** indicates that the top-level operator in that argument must be of either the same or lower precedence. Precedences are between 1 and 1200; the lower the numerical value of a precedence, the higher the precedence of the operator and the tighter it binds its arguments. It is possible to declare a functor as only one unary and/or one binary operator. If a functor is declared as both a unary and binary operator, both must have the same precedence.

The standard operator declarations – which are nearly all as in DEC-10 Prolog – are listed in Appendix 8. Parentheses can be used to override precedence of operators. If an operator is to be used as an atom, it may need to be enclosed in parentheses.

For example, given the standard operator declarations,

```
X = Y :- unify(X, Y).
```

is a legal clause, and

```
?- write(a + b.[]), writeln(not p(X., Y)).
```

is a legal goal.

A predicate which has been defined as an operator may also be used in **prefix** format, so that for example **=** may be used as **=(X, Y)** or **X = Y**, and **all/2** may be used as **all Var Goal** or **all(Var, Goal)**. However, this implies that in terms such as **a / (b+c)** there must be space between the / and the opening parenthesis, or / will be treated as the functor of **(b+c)** rather than as an operator.

3.9. Comments

`Comments` in NU-Prolog are treated as if they were white space, and therefore may not appear within an atom, variable, integer, float or string, and between the functor and the opening parenthesis of a term. Comments are either:

(1) The character **%** followed by any sequence of characters up to a newline.

(2) The sequence /***** followed by any sequence of characters up to the sequence *****/. This type of comment does not nest.

CHAPTER 4

SEMANTICS OF NU-PROLOG

This chapter is organized as follows. §4.1 of this chapter defines declarative and operational semantics. §4.2 gives the declarative and operational semantics of declarative constructs and describes **when** declarations. Users should use the declarative semantics for writing programs, and refer to the operational semantics only when a specific order execution is desired. §4.3 gives the operational semantics of some non-declarative constructs; these constructs should be avoided wherever possible.

4.1. Semantics

Predicates and constructs in NU-Prolog may have two types of semantics, a declarative semantics and an operational semantics. On this basis, predicates and constructs may be divided into two classes.

The first class is based on extended syntax [Lloy84b] and advanced negation facilities [Nais86], and consists of predicates and constructs which have a declarative semantics. It includes such logical facilities as implication, if-then-else, an all-solutions predicate, negation and quantification. The predicates and constructs in this class may also have operational semantics. However, the user may use **pure/1** to declare a predicate to be purely declarative if its definition only uses predicates and constructs which have a declarative semantics.

The compiler will optimise user-defined predicates which are purely declarative. The order of evaluation is not specified and subexpressions may be evaluated repeatedly. If such predicates are not declared as **pure/1**, they have both declarative and operational semantics. A **pure/1** declaration should be omitted from declarative code only when a particular operational semantics is required; the compiler will in this case make no attempt to optimise the predicate in any way which alters its operational behaviour.

The second class consists of those predicates and constructs which are purely operational, and have no declarative semantics. It includes non-logical quantifiers, which highlight non-logical code, and provide greater flexibility than non-logical constructs such as unsafe negation (**\+**) and unsafe if (**–>**). These are described in §4.3.3.

Good programming style dictates the use of predicates and constructs from the former class wherever possible and that predicates from the latter class should generally be avoided. The behaviour of clauses which mix the two is unpredictable.

?– pure Functor/Arity

The predicate definition with *Functor* and *Arity* is purely declarative. No particular operational semantics is intended, and limitations imposed by the operational semantics may not apply. Predicates definitions which are declared **pure/1** may use any mixture of logical constructs and predicates, but should not contain predicates or constructs which are defined to be non-logical. Some transformations and optimisations are applied to pure predicates to execute them more efficiently.

Not meaningful at the command level of the interpreter.

4.1.1. Declarative Semantics

The **declarative** semantics of a NU-Prolog program is given by the meaning of the clauses in the program as formulas of first order logic. The order in which subgoals are called is not relevant to the declarative semantics of a program, and the NU-Prolog system is free to manipulate the program into a more efficient but logically equivalent form. As long as the subgoals of each clause are **logical**, such manipulation cannot affect the bindings returned to a top-level goal. A logical predicate is one with a well-defined declarative semantics. A detailed treatment of this subject is given in [Lloy84a].

4.1.2. Operational Semantics

The **operational** semantics of a NU-Prolog program is given by regarding each clause

$$\mathbf{A\ :\!-\ B_1, \ldots, B_n.}$$

as a predicate definition. If

$$\mathbf{?\!-\ C_1, \ldots, C_k.}$$

is a goal, then each $\mathbf{C_j}$ is regarded as a predicate call. A program is run by giving it an initial goal. A step in the computation process involves the **unification** of some $\mathbf{C_j}$ in the current goal with the head **A** of a program clause, thus reducing the current goal to

$$\mathbf{?\!-\ (C_1, \ldots, C_{j-1}, B_1, \ldots, B_n, C_{j+1}, \ldots, C_k)}\theta$$

where θ is the most general unifier of $\mathbf{C_j}$ and **A**. Unification is a uniform mechanism for parameter passing, data selection and data construction. The computation terminates when the empty goal is produced.

When a variable unifies with a term, the variable is **bound** to that term and the two are indistinguishable. A term is **ground** if it contains no variables. A term which is not a variable is **instantiated**. If this term is not ground, it is **partially instantiated**.

The **computation rule** determines which subgoal $\mathbf{C_j}$ is selected. Like other Prolog systems, NU-Prolog normally selects the leftmost subgoal. However, NU-Prolog is able to **delay** subgoals, that is, call other subgoals until the delayed subgoal is further instantiated. Many system predicates delay until they are **sufficiently instantiated**, that is, until the call is **safe**. A call is **unsafe** if it fails (or loops infinitely) but further instantiation could make it succeed (or terminate). The implementation of negation in NU-Prolog guarantees safety by delaying until globals variables are ground (see §4.2.3). **when** declarations may also be used to make user-defined predicates safe by causing the predicate to delay until some arguments are nonvariable or ground. A goal has **floundered** if all of its unproved subgoals are delayed and none can be woken.

If a subgoal fails, NU-Prolog **backtracks** to the previous subgoal to find a different solution, in an attempt to change the bindings which caused the current subgoal to fail. If the previous subgoal has many solutions, but does not bind the variables which caused failure in the current subgoal, the goal may be very expensive. Programmers should avoid this kind of structure in programs.

If a predicate definition has more than one clause, the clauses are considered in the order in which they appear, so that if the given subgoal cannot be proved with the first clause of a predicate, then NU-Prolog will attempt to prove it using the second clause, and so on.

Negated subgoals are handled with the **negation–as–failure** rule (a weaker form of the closed world assumption). When a negated subgoal **not G** is selected, **G** is called; if **G** fails finitely, **not G** succeeds, and if **G** succeeds **not G** will fail. Negated calls may be nested. Unlike most Prolog systems, NU-Prolog provides a safe implementation of negation.

4.2. Declarative Constructs

4.2.1. Standard Constructs

Head :– Body

Declaratively: clause definition; *Head* is true if *Body* is true. *Head* and *Body* are terms. The implementation of negation assumes that if a call is true then there exists a clause for which the call matches *Head* and, after the substitution is applied, *Body* is true.

Operationally: if *Head* is unified with the current subgoal, NU-Prolog attempts to prove *Body*, which may bind variables in *Head*.

?– Formula

Declaratively: *Formula* is a goal to be proved. True if *Formula* is true.

Operationally: attempt to solve *Formula*. If some subgoals of *Formula* are delayed when all other subgoals have been solved, *Formula* has floundered.

Goals in source programs are executed twice, at compile time and when the object files are loaded together.

Nonterminal – –> Expression

Statement in a definite clause grammar, as explained in Appendix 2. Equivalent to a clause about **f**/**(n**+**2)**, where *Nonterminal* is **f**/**n**.

Formula
call(Formula)

Declaratively: true if the subgoal given by the binding of *Formula* is true. This subgoal may only include constructs and predicates available at the command level of the interpreter. There is no implicit quantification inside *Formula*.

Operationally: delays until *Formula* is instantiated. If *Formula* contains '**!**' (cut), the scope of '**!**' is local to *Formula*.

$Formula_1$, $Formula_2$

Declaratively: $Formula_1$ and $Formula_2$. True if both $Formula_1$ and $Formula_2$ are true.

Operationally: $Formula_1$ is called, and if $Formula_1$ succeeds or delays then $Formula_2$ is called. If $Formula_1$ fails then '**,**' fails; if $Formula_2$ fails then NU-Prolog backtracks and an alternative solution for $Formula_1$ is sought before $Formula_2$ is tried again.

$Formula_1$; $Formula_2$

Declaratively: $Formula_1$ or $Formula_2$. True if either $Formula_1$ or $Formula_2$ is true.

Operationally: $Formula_1$ is called, and if $Formula_1$ succeeds then '**;**' succeeds. On backtracking, further solutions for $Formula_1$ are sought; only when $Formula_1$ fails is $Formula_2$ called.

4.2.2. Quantifiers

NU-Prolog supports both existential and universal quantification. Each quantifier has the form **`Quantifier Variables Formula`**. In logical quantification, any variables which occur syntactically in the term *Variables*, or are quantified inside *Formula*, are considered to be local to *Formula*. If the same variable name occurs in distinct scopes it refers to distinct variables. As well as explicit quantification some variables are implicitly quantified. Variables which only occur once in a clause, and are not in the head, are locally quantified existentially, except in ~**=**, where the implicit quantification is universal. *nit* expects unique variables to be prefixed with '_'. There is no implicit quantification in code inside **`dynamic`** predicates and **`call`**/**`1`**.

NU-Prolog provides operational semantics for existential quantifiers in any situation but universal quantifiers are only useful in conjunction with certain other connectives.

all Vars Formula

> *Declaratively*: universal quantification. All variables in the term *Vars* are universally quantified in *Formula* and the scope of these variables is restricted to *Formula*. True if, for all values of *Vars*, *Formula* is true.
>
> *Operationally*: the operational semantics of **`all`** in conjuntion with the connectives **`not`**, ~**=**, **=**>, <**=** and <**=**> are described in the sections relating to these constructs. All other uses of **`all`**/**`2`** will result in floundering.

some Vars Formula

> *Declaratively*: existential quantification. All variables in the term *Vars* are existentially quantified in *Formula*, and the scope of these variables is restricted to *Formula*. True if, for some value of *Vars*, *Formula* is true.
>
> *Operationally*: if all non-local variables in *Formula* are ground, only one solution to *Formula* is sought; if any non-local variable in *Formula* is not ground, **`some Vars Formula`** is equivalent to **`Formula`**. Special cases of **`some`**/**`2`** are described in **`not`**/**`1`**, **`if-then`** and **`solutions`**/**`3`**.

4.2.3. Negation, Implication, and Related Constructs

The constructs in this section provide safe negation and selection, and should be used instead of the unsafe **\+**, **–**> and **\=**.

not Formula
all Vars not Formula
not (some Vars Formula)

> *Declaratively*: the negation of *Formula*. True if **`some Locals Formula`** is false, where *Locals* is *Vars* plus any variables quantified – implicitly or otherwise – inside *Formula*.
>
> *Operationally*: the call to **`not`**/**`1`** delays until all variables other than *Locals* are ground. *Formula* is then called and if it succeeds, **`not`**/**`1`** fails; otherwise, **`not`**/**`1`** succeeds.

$Term_1$ ~= $Term_2$
all Vars $Term_1$ ~= $Term_2$

Declaratively: same as **`not Term`$_1$ `= Term`$_2$** and **`not some Vars Term`$_1$ `= Term`$_2$** respectively. True if, for all bindings of quantified variables, *$Term_1$* is distinct from *$Term_2$*

Operationally: if *$Term_1$* and *$Term_2$* won't unify then ~**=** succeeds. If *$Term_1$* and *$Term_2$* unify without binding any non-local variables, ~**=** fails; otherwise, the call delays until *$Term_1$* or *$Term_2$* is further instantiated.

$Formula_1$ => $Formula_2$
all Vars $Formula_1$ => $Formula_2$

Declaratively: *$Formula_1$* implies *$Formula_2$*. True if, for all bindings of local variables, *$Formula_1$* is false or *$Formula_2$* is true.

Operationally: the same as **`not Formula`$_1$ `; Formula`$_2$** and **`not some Vars (Formula`$_1$`, not Formula`$_2$`)`** respectively.

$Formula_2$ <= $Formula_1$
all Vars $Formula_2$ <= $Formula_1$

Declaratively and *operationally*: the same as **`Formula`$_1$ `=> Formula`$_2$** and **`all Vars Formula`$_1$ `=> Formula`$_2$** respectively.

$Formula_1$ <=> $Formula_2$
all Vars $Formula_1$ <=> $Formula_2$

Declaratively: *$Formula_1$* and *$Formula_2$* are equivalent. True if, for all bindings of local variables, both *$Formula_1$* and *$Formula_2$* are true, or if both *$Formula_1$* and *$Formula_2$* are false.

Operationally: the same as

`(Formula`$_1$ `=> Formula`$_2$`), (Formula`$_1$ `<= Formula`$_2$`)`

and

`all Vars Formula`$_1$ `=> Formula`$_2$`, all Vars Formula`$_1$ `<= Formula`$_2$

respectively, except the duplication of *$Formula_1$* and *$Formula_2$* does not affect implicit quantification.

if Cond then Formula
if some Vars Cond then Formula

Declaratively: the same as **`(Cond, Formula) ; not Cond`** and **`some Vars (Cond, Formula) ; not some Vars Cond`** respectively, except the duplication of *Cond* does not affect implicit quantification. True if, for some binding of local variables, *Cond* is false or *Formula* is true.

Operationally: the call delays until the non-local variables in *Cond* are sufficiently instantiated. Unless *Cond* is **`=`**, the call delays until the non-local variables in *Cond* are ground; otherwise, the call delays until these variables are sufficiently instantiated. *Cond* is then called, and if it fails, **`if-then`** succeeds. If *Cond* succeeds, *Formula* is called. If no variables in *Vars* are in *Formula* and *Formula* fails, **`if-then`** also fails; otherwise (some variables in *Vars* are in *Formula*), if *Formula* fails, alternate solutions for *Cond* are tried. Only when *Cond* fails does **`if-then`** fail.

if Cond then Formula$_1$ else Formula$_2$
if some Vars Cond then Formula$_1$ else Formula$_2$

Declaratively: the same as

(Cond, Formula$_1$) ; (not Cond, Formula$_2$)

and

some Vars (Cond, Formula$_1$) ; (not (some Vars Cond), Formula$_2$)

respectively, except the duplication of *Cond* does not affect implicit quantification. True if, for some binding of local variables, *Cond* and *Formula*$_1$ are true, or *Cond* is false and *Formula*$_2$ is true.

Operationally: the call delays until the non-local variables in *Cond* are sufficiently instantiated. Unless *Cond* is an **`=`**, the call delays until the non-local variables in *Cond* are ground; otherwise, the call delays until these variables are sufficiently instantiated. *Cond* is called, and if it succeeds, *Formula*$_1$ is called. If *Cond* succeeded and no variables in *Vars* are in *Formula*$_1$ and *Formula*$_1$ fails, **`if-then-else`** also fails; otherwise (some variables in *Vars* are in *Formula*$_1$) if *Formula*$_1$ fails, alternate solutions for *Cond* are tried. If *Cond* has no solutions, *Formula*$_2$ is tried, and only if *Formula*$_2$ fails does **`if-then-else`** fail.

4.2.4. Aggregate Functions

Some examples of the following constructs are given in Appendix 1.

solutions(Term, Formula, Set)
solutions(Term, some Vars Formula, Set)

Declaratively: *Set* is the list of instances of *Term* for which *Formula* is true. The scope of the variables in *Term* is the call to **`solutions/3`**, so that they are considered local. Same as

```
all X (some Term (Formula, X = Term) <=> member(X, Set)),
                                    sorted(Set)
```

and

```
all X (some Term.Vars (Formula, X = Term) <=> member(X, Set)),
                                    sorted(Set)
```

respectively, where **`member/2`** is the standard list membership predicate and **`sorted(Set)`** is true if *Set* is a sorted list without duplicates.

Operationally: all answers to *Formula* are found using backtracking. For each answer, the bindings of *Term* and all non-local variables are saved. For each distinct binding of non-local variables, an answer is returned. The list *Set* is the sorted (by **`termCompare/3`**, with duplicates removed) set of instances of *Term* corresponding to these bindings. It is an error for the binding to *Term* to contain any local variables. Some answers may have delayed calls to ~**`=`**. If the binding to *Term* contains global variables there can also be delayed calls to **`termCompare/3`** and **`sort/2`**.

count(Term, Formula, Result)

Declaratively: same as

`solutions(Term, Formula, Set), length(Set, Result)`.

The scope of variables in *Term* is the call to **`count/3`**. *Result* is the number of distinct instantiations of *Term* by *Formula*.

Operationally: **`count/3`** delays until non-local variables in *Term* and *Formula* are ground. Each instantiation of *Term* must be ground (to restrict *Result* to finite values), or **`count/3`** will fail with an error message.

max(Term, Formula, Result)

Declaratively: same as

`solutions(Term, Formula, Set), maximum(Set, Result)`, where **`maximum/2`** finds the maximum element of *Set*.

The scope of variables in *Term* is the call to **`max/3`**. *Result* is the maximum (via the non-logical standard ordering) instantiation of *Term* by *Formula*.

Operationally: **`max/3`** delays until global variables in *Formula* are ground. Each instantiation of *Term* may contain local variables (this is non-logical).

min(Term, Formula, Result)

Declaratively: same as

`solutions(Term, Formula, Set), minimum(Set, Result)`, where **`minimum`**/**2** finds the minimum element of *Set*.

The scope of variables in *Term* is the call to **`min`**/**3**. *Result* is the minimum (via the non-logical standard ordering) instantiation of *Term* by *Formula*.

Operationally: **`min`**/**3** delays until global variables in *Formula* are ground. Each instantiation of *Term* may contain local variables (this is non-logical).

sum(Summand, Term, Formula, Result)

Declaratively: same as

`solutions(Summand-Term, Formula, Set), sumOfKeys(Set, Result)`. where **`sumOfKeys`**/**2** sums the first arguments of of each **`-`**/**2** pair in *Set*.

The scope of variables in *Term* and *Summand* is the call to **`sum`**/**4**. *Result* is the sum of the instantiation of *Summand* for each distinct instantiation of *Term* and *Summand* by *Formula*. If *Formula* fails then *Result* is **`0`**.

Operationally: **`sum`**/**4** delays until global variables in *Formula* are ground. *Summand* may be any arithmetically evaluable term. Each instantiation of *Summand* and *Term* must be ground or **`sum`**/**4** will fail with an error message.

4.2.5. When Declarations

Predicates without **`when`** declarations behave like conventional Prolog predicates – calls to them either succeed or fail. Calls to predicates which have **`when`** declarations may also delay, resuming execution when their arguments are further instantiated. Making some calls delay is essential for ensuring safeness (particularly of negation), termination (for example, of predicates which are called in several ways), and efficiency (especially generate-and-test algorithms). **`When`** declarations specify that a predicate should only proceed with a call when certain variables are partially or totally instantiated.

If a predicate has **`when`** declarations then, in the present implementation, its clauses are indexed. In a future release NU-Prolog may have an explicit clause indexing facility for rapid access to the appropriate clause in predicates with a large number of constituent clauses.

Appendix 1 contains some examples of predicate definitions with **`when`** declarations.

?- Head when Body

Operationally: calls to a predicate with **`when`** declarations delay until one of the declarations is satisfied. A **`when`** declaration can only appear when a predicate is being compiled. A **`when`** declaration is satisfied when the call to the predicate is an instance of some *Head* and, when the call and *Head* have been unified, *Body* is satisfied. The set of **`when`** declarations for a predicate is also satisfied if the call does not unify with any *Head*. *Body* may contain:

(1) The single atom **`ever`**, which is always satisfied.
(2) Variables, which are satisfied when instantiated.
(3) Terms of the form **`ground(Var)`**, which are satisfied when *Var* is ground.
(4) **`A and B`**, which is satisfied when *A* is satisfied and *B* is satisfied; where *A* and *B* are of form (2) to (6).
(5) **`A or B`**, which is satisfied when *A* is satisfied or *B* is satisfied; where *A* and *B* are of form (2) to (6). This binds less strongly than **`and`**.
(6) **`(A)`** where *A* is of form (2) to (6).

Some sets of **`when`** declarations with complex *Heads* cannot be compiled, in which case an error message is printed. The following conditions must be observed:

(1) The heads of all **`when`** declarations should be mutually non-unifiable.
(2) There cannot be any repeated variables in the head of a **`when`** declaration.

Variable Formula
freeze(Variable, Formula)

Declaratively: true if the subgoal given by the binding of *Formula* is true. This subgoal may only include constructs and predicates available at the command level of the interpreter. There is no implicit quantification inside *Formula*.

Operationally: delays until *Var* and *Formula* are instantiated. If *Formula* contains '**`!`**' (cut), the scope of '**`!`**' is local to *Formula*.

4.3. Non-declarative Constructs

4.3.1. Non-logical Quantifiers

The non-logical quantifiers **`gSome`/`2`** and **`gAll`/`2`** may be used wherever **`some`/`2`** or **`all`/`2`** may be used with operational semantics. The ‘g’ prefix stands for **`global`**.

gAll Vars Goal

(Non-logical).
Operationally: same as **`all Vars Goal`**, except that the scope of variables in *Vars* is not restricted to *Goal*.

gSome Vars Goal

(Non-logical).
Operationally: same as **`some Vars Goal`**, except that the scope of the variables in *Vars* is not restricted to *Goal*.

4.3.2. Cut

`Cut` is used to restrict the search tree. **`Cut`** is highly dangerous, and should be avoided wherever possible; the constructs described in this chapter supersede most of the current uses of **`cut`**.

`!`

(Non-logical).
Operationally: the **`cut`** operation. It succeeds, but on backtracking it fails, and everything fails up to and including the most recent ancestor which is not **`,`**, **`;`** or **`->`**; that is, no subgoals to the left of **`!`** are retried. If the **`!`** is within an **`;`**, no further branches within **`;`** are tried and **`;`** fails. If the **`!`** is within **`call`/`1`** then **`call`/`1`** fails. Otherwise, no more clauses of the predicate containing **`!`** are tried, and the predicate fails. Other uses of **`!`** are undefined, the behaviour of **`!`** is not defined if it occurs in the condition of **`->`**, and **`!`** should not be used in conjunction with quantifiers (§4.2.2), negation, implication and related constructs (§4.2.3), aggregates (§4.2.4), or predicates which may delay.

Cut can be used to implement many of the unsafe features of Prolog such as **`\+`**, **`\=`**, **`==`** and **`var`/`1`**. It should be used as sparingly as possible.

4.3.3. Non-logical Constructs

The predicates described in this section are unsafe counterparts of some of the predicates described in §4.2. Although they are occasionally useful, they should be avoided.

\+ Goal

(Non-logical).
Operationally: similar to **`not`/`1`**, but there are no quantifiers and the call never delays. If *Goal* fails then **`\+ Goal`** succeeds, and if *Goal* succeeds then **`\+ Goal`** fails. The result is meaningless if *Goal* succeeds and binds any variables, or succeeds with some calls delayed.

Cond –> Goal

(Non-logical).

Operationally: similar to **`if Cond then Goal`** but (unlike **`if–then`**) there are no quantifiers, the call never delays, and *Cond* is never retried. If *Cond* succeeds then $Goal_1$ is called; otherwise, **`–>`** succeeds. Note that most other Prolog systems have **`Cond –> Then`** fail if **`Cond`** fails.[1]

Cond –> Goal$_1$; Goal$_2$

(Non-logical).

Operationally: similar to **`if Cond then Goal`$_1$ `else Goal`$_2$** but (unlike **`if–then–else`**) there are no quantifiers, the call never delays and *Cond* is never retried. If *Cond* succeeds then $Goal_1$ is called; otherwise $Goal_2$ is called.

once Goal

(Non-logical).

Operationally: finds the first solution (if any) to *Goal*. Similar to **`call((Goal, !))`**.

setof(Term, Goal, Set)

(Non-logical).

Operationally: similar to **`solutions/3`** but unsafe. The notation **`Vars^Goal`** is used instead of **`some Vars Goal`**. If there are no solutions to *Goal*, **`setof/3`** fails. Local variables are allowed in answers, which are sorted with **`compare/3`**. In the cases in which **`solutions/3`** succeeds with delayed inequalities, **`setof/3`** fails.

bagof(Term, Goal, Bag)

(Non-logical).

Operationally: same as **`setof/3`** except that *Bag* is not sorted and duplicates are not removed.

findall(Term, Goal, Set)

(Non-logical).

Similar to **`solutions/3`** but unsafe, *Set* is not sorted, duplicates are not removed, and **`Set = []`** if there are no solutions to *Goal*. There are never any delayed calls. Also similar to **`bagof/3`**, but there are no local variables.

countall(Goal, Count)

(Non-logical).

`Countall/2` is an efficient equivalent to **`findall(1,Goal,L),length(L,Count)`** and is thus similar to **`sum(1,Goal,Goal,Count)`** but unsafe. The answers to *Goal* are not sorted and duplicates are not removed. There are never any delayed calls.

---

[1] There is a justification for this otherwise weird behaviour, but it's so arcane that NU-Prolog does it the logical way.

Vars ^ Goal

Operationally: the same as **`call(Goal)`**. Used in **`setof/3`** and **`bagof/3`** to indicate existential quantification.

CHAPTER 5

SYSTEM PREDICATES

This chapter contains of descriptions of the system predicates supplied with the NU-Prolog system. Some of these descriptions include a **`when`** declaration for that predicate. However, because most predicates are defined recursively, the **`when`** declaration is only a partial indication of the instantiation required before the predicate can be tried. For example, **`isList`**/**`1`**, which has the **`when`** declaration **`?- isList(List) when List`**, will not complete until the length of the list is known; that is, until the tail of the list has been instantiated to **`[]`**.

Some system predicates of other Prolog systems are described in the compatibility library.

5.1. Predicates which Succeed or Fail

fail

Always fails.

repeat

Always succeeds, even on backtracking.

true

Succeeds. On backtracking, it fails.

5.2. Predicates for Conditional Compilation or Loading

initializing

True only at load time. During compilation, when **`initializing/0`** is false, any '**`?-`**' goals encountered in the source will be executed. These goals are also collected together into an initialization block which is executed at load time, at which point **`initializing/0`** is true.

muprolog

Fails. Useful for conditional compilation of clauses which are MU-Prolog dependent.

nuprolog

Succeeds (does nothing). Useful for conditional compilation of clauses which are NU-Prolog dependent.

option(Value)

`option(Value)` is false, except during compilation for *Values* set by the **`-F`** flag to *nc*.

pure(Functor, Arity)

True, during compile time, if the predicate with *Functor* and *Arity* is pure. See **`predicateProperty/3`**.

?- useIf Goal
?- useElse
?- useEnd

`UseIf/1`, **`useElse/0`** and **`useEnd/0`** allow conditional compilation of clauses. *Goal*, in which each subgoal must be a system predicate, is called; if it fails the clauses between the next matching **`useElse/0`** and **`useEnd/0`** are compiled instead of the clauses between the **`useIf/1`** and the **`useElse/0`**. If there is no matching **`useElse/0`**, clauses between the **`useIf/1`** and **`useEnd/0`** are only compiled if *Goal* succeeds. Note that **`useIf/1`** controls the compilation of whole clauses. These goals may be nested.

5.3. Interpreter Predicates

These predicates are usually called from the top level of the interpreter *np*, rather than being part of programs.

5.3.1. Examining the Program State

listing
listing Predicate
listing [$\mathbf{Predicate_1}$,...,$\mathbf{Predicate_n}$]

(Non-logical).
Print all dynamic predicate definitions, or print all dynamic definitions of *Predicate* (or $Predicate_1, \ldots, Predicate_n$), where *Predicate* is specified by functor or functor/arity.

man Functor
man [$\mathbf{Functor_1}$,...,$\mathbf{Functor_n}$]

(Non-logical).
Print information about all system and library predicates with *Functor*. This information is identical to that in the NU-Prolog Reference Manual; therefore, some information about individual predicates − such as that given at the start of a section − will not be displayed by this command.

5.3.2. Examining and Controlling the Execution State

abort

(Non-logical).
Aborts execution of the current goal and returns to the top level of the program. In *np* this is the top level of the interpreter. In a compiled program it is **`main`**/**`1`**, or the predicate specified by the **`-e`** option.

ancestors(Anc)

(Non-logical).
Anc is unified with a list of interpreted ancestor goals of the current clause starting with the parent goal and ending with the oldest accessible ancestor. Ancestors of compiled goals are not accessible.

break

(Non-logical)
Causes a new invocation of the top-level of the interpreter. The previous computation is resumed when this has finished. Break levels are named with ascending integers. The top-level is level 0.

breakLevel(N)

N is the current break level number. See **`break`**/**`0`**.

catch(Goal, Result)

(Non-logical).

Catch/2 calls *Goal* after arranging to receive any abnormal returns from **throw/1** via *Result*. In the absence of such a return, it is equivalent to **call(Goal)**.

commandNumber(N)

N is the current history list command number. See **h/0**.

depth(N)

(Non-logical).

N is the number of ancestors of the current call, counting only interpreted predicate calls up to and not including the first ancestor, if any, which is a compiled predicate.

maxDepth(N)

(Non-logical).

Set the maximum number of nested interpreted calls to *N*. Calls beyond this cause a trap to the debugger.

Currently only available in trace mode.

prompt

Called by the interpreter to print the interpreter prompt. Can be defined by the user. Currently defined by:

```
prompt :-
      breakLevel(B),
      commandNumber(C),
      ( B = 0,
            printf(user, ''%d?- '', [C])
      ; B > 0,
            printf(user, ''[%d] %d?- '', [B, C])
      ).
```

restart

(Non-logical).

Aborts execution of the current goal and returns to the top level of the program. In *np* this is the top level of the interpreter. In a compiled program it is **main/1**, or the predicate specified by the **-e** option. Same as **abort/0**.

subGoalOf(S)

(Non-logical).

Equivalent to **ancestors(A), member(S,A).**

As with ancestors, this only works for interpreted goals.

throw(Result)

(Non-logical).

Throw/1 cause an abnormal return from within a **catch/2**. It works backwards up the stack of ancestor goals until a **catch/2** is found, the second argument of which unifies with *Result*. This unification is performed and execution continues from the **catch/2**. If there is no matching **catch/2** then an **abort/0** is performed.

Throw/1 does not remove choice-points.

It is not a good idea to throw variables because these cannot be distinguished from a normal return.

The main use of **catch/2** and **throw/1** is in reporting errors.

5.3.3. Debugging Predicates

NU-Prolog provides a fairly standard, procedure-box-control-flow based debugger. The execution of predicates may be examined as they are called, exited, redone, or failed, and as they delay or wake.

Debugging is controlled by the debug mode which is a flag with the possible values of **off**, **debug**, or **trace**. The flag is manipulated by the **debug/0**, **nodebug/0**, **trace/0**, and **notrace/0** predicates. Along with other information, it is displayed by **debugging/0**.

In **debug** mode, only those predicates which have spypoints set on them are examined. Spypoints are set and removed by **spy/1** and **nospy/1**. **Nospyall/0** removes all spy points. The presence of a spypoint has no effect on the execution of a predicate if debugging is **off**.

In **trace** mode, the execution of all interpreted goals is examined as well as spypoints. There is no way to trace compiled code.

Debug mode is the default. Finer control of the debugger is provided by **leash/1**, **spyCondition/3**, and **leashCondition/3**.

When a predicate is being debugged, the debugger takes control at each ‘‘port’’ connecting it to its caller and displays a line of the form

XY (Number) Depth Port: Goal ?

Number is a unique integer identifying this particular call, *Depth* is the number of ancestors of the call since the last compiled predicate, *Port* is one of **call**, **delay**, **wake**, **redo**, **exit** or **fail**, and *Goal* is the call being run. *X* is the character ***** if this is a spypoint and blank otherwise. *Y* is the character > if this port was reached by skipping (see the **s** option below), and *X* otherwise.

For each goal and port reached that is not unleashed, the debugger pauses for instruction from the user. A line is read from user_input and its first character used to select one of the following actions. Some of these use the rest of the line as arguments, but most discard it.

<cr> Creep. Execution continues without debugging until the next port is reached. This is equivalent to switching temporarily to **trace** mode.

\+ Set spypoint. A spypoint is set on the current procedure.

- Remove spypoint. Any spypoint on the current procedure is removed.

@ Single command. Reads a term (terminated by a ‘**.**’) from the terminal and executes it with debugging off.

= Debugging. This calls **debugging/0**.

< *n* Set debugger print depth. The debugger prints goals only to a certain depth. The initial value is 10, but this option can be used to alter it. No value, or one of 0, removes the limit completely.

| Pipe. The debugger calls **spyHook(info(Number, Depth, Port), Goal)**, where *Number* is the current call number, *Depth* is its depth, *Port* is the port the debugger has stopped at, and *Goal* is the current goal. This is mainly useful for interfacing to user-supplied meta-debuggers.

? Help. Prints a table of these options.

a Abort. This calls **abort/0**, causing a return to the top-level.

b Break. This calls **break/0**, giving a new invocation of the top-level command interpreter. The debugger resumes control on exit from this invocation.

c Creep. Execution continues without debugging until the next port is reached. This is equivalent to switching temporarily to **trace** mode.

d Display. Writes the current goal at the terminal using **display/1**.

f Fail. Force execution to the **fail** port of the current goal.

f *n* Fail earlier call. Force execution to the **fail** port of goal number *n*. If goal number *n* is no longer accesible, execution is moved to the **fail** port of the first goal after it that is.

g Print ancestors. Prints the ancestors of the current goal using print with the current debugger depth limit. Only ancestors which were called while the debugger was on are printed and then only if they were called by interpreted code or are spypoints.

g *n* Print some ancestors. Like **g**, except that at most *n* ancestors are printed.

h Help. Prints a table of these options.

l Leap. Execution continues without debugging until the next spypoint port is reached. This is equivalent to switching temporarily to **debug** mode.

n Nodebug. Turns debugging off until execution returns to the interpreter top-level, at which time debugging mode resumes its original value. To turn off debugging permanently use **nodebug/0** at the top-level.

p Print. Writes the current goal at the terminal using **print/3**. The current debugger maximum print depth is used to truncate the goal. It can be set with the < option.

r Redo. Returns execution to the **call** or **wake** port of the current goal. At either of these it has no effect. The execution state will be similar to that when the goal was called, but side-effects such as database and i/o predicates are not undone.

r *n* Redo earlier call. Returns execution to the **call** or **wake** port of goal number *n*. If goal number *n* is no longer accesible, execution is moved to the **call** or **wake** port of the first goal after it that is.

s Skip. Execution continues without debugging until the current goal's **exit** or **fail** port is reached. This is equivalent to switching debug mode temporarily to **off**. At **delay**, **exit**, or **fail** ports, skip is equivalent to creep. Currently, skip's behavior at **call** or **wake** ports is different from that at **redo** ones. At the former ports, all debugging activity is suppressed, preventing the keeping of records of sub-goals called. At the latter, sub-goals which have already been recorded by the debugger remain, but are not displayed.

w Write. Writes the current goal at the terminal using **write/1**.

debug

(Non-logical).
Enter debug mode by setting the **debugging** flag to **debug**. This is done automatically by **spy/1** if **debugging** is **off**. See **trace/0** and **prologFlag/3**.

debugging

(Non-logical).
Lists all current spypoints and information about debugging options.

leash(Mode)

(Non-logical).
Leash/1 controls the behaviour of the debugger when it traces a goal. By default, the debugger stops at all ports it traces and asks the user to instruct it on how to continue, but a call to **leash/1** with a list of port names can be used to tell it to stop only at the listed ports. Other ports are still printed, and execution still stops at any predicates with spypoints on them. **Leash(all)** restores leashing to the default.

leashCondition(Goal, Port, Condition)

LeashCondition/3 is a hook for the user to alter the leashing behaviour of the debugger. Clauses asserted to it can be used to over-ride the leashing mode currently in force. If **leashCondition(Goal, Port, Condition)** succeeds for a particular *Goal* and *Port* being traced, then the truth or falsity of *Condition* determines whether the debugger stops after printing *Goal*.

LeashCondition/3 has no effect on spypoints.

nodebug

(Non-logical).
Set the **debugging** flag to **off**. This doesn't remove any spy points – it only causes them to be ignored. See **prologFlag/3**.

nospy Predicate
nospy [Predicate_1,...,Predicate_n]

(Non-logical).
Removes any spypoints on *Predicate* (or $Predicate_1$,...,$Predicate_n$), where *Predicate* is specified by functor or functor/arity. If no arity is specified all predicates with the given functor have their spypoints removed. See **spy/1**.

nospyall

(Non-logical).
Remove all spypoints.

notrace

(Non-logical).
Set **debugging** flag to **off**.

spy Predicate
spy [Predicate$_1$,...,Predicate$_n$]

(Non-logical).
Places a spypoint on *Predicate* (or *Predicate*$_1$,...,*Predicate*$_n$), where *Predicate* is specified by functor or functor/arity. If no arity is specified all predicates with the given functor are spied on. It is not possible to place spypoints on system predicates. See **`nospy/1`**.

`Spy/1` turns on debugging if it is currently off.

spyCondition(Goal, Port, Condition)

`SpyCondition/3` is a hook for the user to alter the behaviour of the debugger at spypoints. Its use is similar to **`leashCondition/3`**, but **`spyCondition/3`** determines whether the spypoint port reached is displayed at all. If **`spyCondition(Goal, Port, Condition)`** succeeds for a particular *Goal* and *Port* being debugged, then the truth or falsity of *Condition* determines whether the debugger displays the port.

`SpyCondition/3` has no effect on tracing.

spyHook(Info, Goal)

`SpyHook/2` is a hook to enable the user to pass a goal being debugged to a user-defined predicate. The **`|`** option to the debugger calls **`spyHook(info(Number, Depth, Port), Goal)`**, where *Number* is the unique call number, *Depth* the number of ancestor calls, *Port* is the port at which the debugger has stopped, and *Goal* is the goal being debugged. Control returns to the debugger after **`spyHook/2`** completes.

trace

(Non-logical)
Set **`debugging`** flag to **`trace`** mode.

5.4. Terms and Lists

The predicates in this section are used for examination and manipulation of terms. The non-logical predicates should be avoided where possible.

5.4.1. Manipulating Lists

The predicates in this section are defined recursively, so that they may partly proceed before delaying. They are designed so that they may be called safely with any pattern of instantiation. If there is a `when` declaration on an argument which is a list, the length of the list must be known before the predicate can be completely solved.

append(List$_1$, List$_2$, JoinList) when List$_1$ or JoinList

JoinList is $List_2$ appended to $List_1$. Fails if the first argument is not a list.

delete(Element, List, Rest) when List or Rest

Rest is *List* with the first element matching *Element* deleted. On backtracking, the next appropriate element is chosen instead. Fails if *List* or *Rest* are not lists.

isList(Term) when Term

Term is a list. Delays until *Term* is instantiated and the length of *Term* is known, that is, its tail has been instantiated to `[]`.

keySort(List, Result)
keySort(Order, List, Result)

Result is *List* sorted by *Key*, where elements of *List* are in the form *Key−Value*. *Order* is either + (ascending) or `-` (descending). `compare/3` is used for comparison of terms. These predicates do not remove duplicates.

length(List, N) when List or N

List has length *N*. Either *List* or *N* may be variables; if they are both variables, the call delays. Fails if *List* is not a list.

member(Element, List) when List

Element is a member of *List*. On backtracking, gets another member of *List* matching *Element*.

member(Element, List, SubList) when List

Element is a member of *List* and *SubList* is the tail of *List* beginning with *Element*. On backtracking, gets another member of *List*, and another *SubList*, matching *Element*. Same as

`SubList = Element._, append(_, SubList, List)`.

merge(List$_1$, List$_2$, NewList) when List$_1$ and List$_2$

Sorted *List*$_1$ and sorted *List*$_2$ are merged, removing duplicates, to give *NewList*. Undefined if either *List*$_1$ or *List*$_2$ is not sorted, and will delay until these lists are sufficiently instantiated.

multiKeySort(Keys, Term, List, SortedList)

Each element of *List* is of the form *Term*. *Keys* is a list of terms; **`multiKeySort/4`** applies a stable sort to *List* using the last element of *Keys*, then the second-last element, and so on. *Term* is successively unified with each element of *List*, binding variables in the current *Key*. *List* of *Elements* is then transformed into a list of the form *Key–Element*, which can be sorted with *keySort*/3, where order is determined by the top-level (unary) functor of each *Key*. Default is ascending, **`+(X)`** is an explicit way of saying ascending on **`X`** and **`-(X)`** is descending on **`X`**. **`MultiKeySort/4`** does not remove duplicates.

notMember(NotElement, List) when List

NotElement is not a member of *List*. This is done by requiring that **`NotElement ~= Element`** for each *Element* of *List*. This is only the same as **`not member(NotElement, List)`** if *NotElement* and *List* are ground.

perm(List$_1$, List$_2$) when List$_1$ or List$_2$

List *List*$_2$ is a permutation of list *List*$_1$. On backtracking, **`perm/2`** returns another permutation. If the elements of the input list were not unique, some of the permutations will be identical. Fails if either argument is not a list.

reverse(ForwList, BackList) when ForwList or BackList

List *BackList* is the reverse of list *ForwList*. The time is proportional to **`O(n)`**, where *n* is the length of either list. Fails if either argument is not a list.

sort(List, SortedList) when List

(Non-logical).

Sort *List* into *SortedList*, removing duplicates, where *List* is a list of arbitrary terms. The system predicate **`compare/3`** is used for term comparison. Delays until *List* is sufficiently instantiated.

sorted(List)

```
?- sorted([]) when ever.
?- sorted(_Head.Tail) when Tail.
```

List is sorted under the non-logical ordering of terms. *List* may contain duplicates.

suffix(List, Suffix) when List

Suffix is successively unified to all of the possible suffixes of *List*, starting with *List* and shrinking to the empty list. Same as **`append(_, Suffix, List)`**. Fails if **`Suffix`** won't unify with some suffix of **`List`**.

5.4.2. Examining Characters

isAlnum(Char) when Char

ASCII code *Char* is alphanumeric. Delays until *Char* is instantiated.

isAlpha(Char) when Char

ASCII code *Char* is alphabetic. Delays until *Char* is instantiated.

isAscii(Char) when Char

Integer *Char* is between 1 and 127. Delays until *Char* is instantiated.

isAsciiL(String) when ground(String)

String is a list of Ascii characters (see **`isAscii`**/**`1`**). Delays until *String* is ground.

isCntrl(Char) when Char

Char is an ASCII code for a control character, that is integer *Char* is between 1 and 32. Delays until *Char* is instantiated.

isDigit(Char) when Char

ASCII code *Char* is a digit. Delays until *Char* is instantiated.

isLower(Char) when Char

ASCII code *Char* is lowercase alphabetic. Delays until *Char* is instantiated.

isPrint(Char) when Char

ASCII code *Char* is a printable character (ASCII codes 9, 10, 12, 13, 32 to 126). Delays until *Char* is instantiated.

isPrintL(String) when ground(String)

String is a list of printable characters (see **`isPrint`**/**`1`**). Delays until *String* is ground.

isUpper(Char) when Char

ASCII code *Char* is uppercase alphabetic. Delays until *Char* is instantiated.

5.4.3. Examining Terms

arg(N, Term, SubTerm) when N and Term

The N^{th} argument of term *Term* is *SubTerm*; it will delay if *N* or *Term* are variables. If *N* is less than 1, or greater than the arity of *Term*, **`arg`**/**`3`** fails.

atom(Term)

(Non-logical).
Term is an atom; if *Term* is currently a variable it fails. See **`isAtom`**/**`1`**.

atomic(Term)

(Non-logical).
Term is an atom or a number; if *Term* is currently a variable it fails. See **`isAtomic/1`**.

atomToString(Atom, String) when Atom or ground(String)

String is the list of ASCII codes in the name of *Atom*, an atom. If *Atom* is a variable, and *String* is not ground, it delays. Fails if the arguments are not of an appropriate type. See also **`name/2`**.

compound(Term)

(Non-logical).
Term is a compound term, that is, not a constant or variable. See **`isCompound/1`**.

cons(Term)

(Non-logical).
The top-level functor in *Term* is '**.**' (cons). Fails if *Term* is currently a variable. See **`isCons/1`**.

const(Term)

(Non-logical).
Term is a constant; if *Term* is currently a variable it fails. See **`isConst/1`**.

duplicate($\mathbf{Term_1}$, $\mathbf{Term_2}$)

(Meta-logical).
$Term_2$ is a copy of $Term_1$ with different variables but the same internal bindings.

float(Term)

(Non-logical).
Term is a float. If *Term* is currently a variable it fails. See **`isFloat/1`**.

functor(Term, Functor, Arity) when Term or Functor and Arity

Term has *Functor* and *Arity*. Delays until *Term* or both *Functor* and *Arity* are not variables. If *Term* is a constant, *Functor* will unify with *Term* and *Arity* will be `0`.

ground(Term)

(Meta-logical).
Term is currently ground, that is, contains no variables.

integer(Term)

(Non-logical).
Term is an integer. If *Term* is currently a variable it fails. See **`isInt/1`**.

intToAtom(Integer, Atom) when Integer or Atom

Integer is converted to *Atom*, or *Atom* to *Integer*. The name of *Atom* is the decimal representation of integer. If both arguments are variables, **`intToAtom`/`2`** delays. Fails if an argument is instantiated to a term of the wrong type.

intToString(Int, String) when Int or ground(String)

String is the list of ASCII codes of the digits in the decimal representation of *Int*. If *Int* is a variable, and *String* is not ground, it delays. Fails if the arguments are not of an appropriate type (cf. **`name`/`2`**).

isAtom(Term) when Term

Term is an atom. Delays until *Term* is instantiated.

isAtomic(Term) when Term

Term is an atom or a number. Delays until *Term* is instantiated.

isCompound(Term) when Term

Term is a compound term, that is, not a constant. Delays until *Term* is instantiated.

isCons(Term) when Term

The top-level functor in *Term* is '**.**' (cons). Delays until *Term* is instantiated.

isConst(Term) when Term

Term is a constant. Delays until *Term* is instantiated.

isExpression(Term) when ground(Term)

Term is a valid arithmetic expression. Delays until *Term* is ground.

isFloat(Term) when Term

Term is a float. Delays until *Term* is instantiated.

isInt(Term) when Term

Term is an integer. Delays until *Term* is instantiated.

isNumber(Term) when Term

Term is a number. Delays until *Term* is instantiated.

isTerm(Term) when Term

Term is a compound term, that is, not a constant. Delays until *Term* is instantiated.

For compatibility reasons, **`IsCompound`/`1`** is should be used in preference to **`IsTerm`/`1`**.

listOfVars(Term, Vars)

(Meta-logical).

Vars is a list of the variables occurring in *Term*.

name(Atom, List) when Atom or ground(List)

List is the list of ASCII codes in the print name of *Atom*, an atom. If *Atom* is a variable, or *List* is not ground, it delays. Fails if the arguments are not of an appropriate type. See also **`atomToString`/2**.

nonvar(Term)

(Meta-logical).

Term is not currently a variable. See **`var`/1**.

number(Term)

(Non-logical).

Term is an number. If *Term* is currently a variable it fails. See **`isNumber`/1**.

occurs(SubTerm, Term)

(Non-logical).

Tests for identity (with **`==`**) between *SubTerm*, a variable or constant, and some subterm of *Term*. Does not bind *SubTerm* if it is a variable.

term(Term)

(Non-logical).

Term is a compound term, that is, not a constant or variable. See **`isTerm`/1**.

For compatibility reasons, **`Compound`/1** is should be used in preference to **`Term`/1**.

termToString(Term, String)
termToString(Flags, Term, String)

(Non-logical).

Converts *Term* to *String* in prefix format (or as specified by *Flags*) with atoms quoted as necessary. Does not convert *String* to *Term*. Possible *Flags* settings are as for **`writev`/2**. See **`sread`/2**.

Term =.. List

List is the functor of *Term* followed by the arguments of *Term*. The call delays if *Term* is a variable or the head of *List* is not instantiated and the length of *List* is not known. Fails if *Term* is an integer.

var(Term)

(Meta-logical).

Term is currently a variable. See **`nonvar`/1**.

waitedOn(Term, Vars)

(Non-logical).
Vars is a list of variables in *Term* on which some goals may be delayed.

5.4.4. Modifying Terms

Normally, Prolog terms are built once and modified only by further instantiation. There are times, usually in the implementation of other reasoning systems, where modifying an existing term directly rather than creating a copy of it encorporating the modification provides enough benefit in clarity of implementation or performance to be worth while. Be warned that programs using the facilities in this section can be very hard to debug and are often not significantly faster than pure versions.

setarg(N, Term, SubTerm)

The N^{th} argument of term *Term* is replaced with *SubTerm*. **`setarg`/3** fails if *N* or *Term* are unbound. If *N* is less than 1, or greater than the arity of *Term*, **`setarg`/3** fails.

Not all NU-Prolog terms can be modified with **`setarg`/3**. The main class of user-visible objects that **`setarg`/3** won't work on is the internal character string type. **`setarg`/3** fails if it cannot perform the replacement.

5.4.5. Comparison of Terms

Logical term comparison observes the following standard ordering:

(1) Variables are equal if they are identical.

(2) Numbers are ordered numerically, with integers less than their floating-point equivalents. Atoms are ordered lexicographically.

(3) Numbers are less than atoms.

(4) If term *X* has arity smaller than *Y*, it is smaller.

(5) Term *X* is less than term *Y* if the functor of *X* is lexicographically less than the functor of *Y*, and *X* and *Y* are equal under conditions (1) to (4).

(6) Term *X* is less than term *Y* if an argument of *X* is less than the corresponding argument of *Y*, and *X*, *Y* and their preceeding arguments are equal under conditions (1) to (5).

Non-logical term comparison is defined by extending the logical ordering with the following ordering for variables:

(1a) Two distinct variables have an ordering based on the local implementation.

(1b) A variable is less than a non-variable.

Programmers should use **`termCompare`/3** rather than the non-logical **`compare`/3** or the **`@<`** family of comparison predicates.

$\mathbf{Term_1}$ = $\mathbf{Term_2}$

$Term_1$ equals $Term_2$; that is, $Term_1$ and $Term_2$ are unified.

$\mathbf{Term_1}$ \= $\mathbf{Term_2}$

(Non-logical).
$Term_1$ and $Term_2$ do not unify; the same as **`\+ Term1 = Term2`**. Similar to ~**=** but, unlike ~**=**, there are no quantifiers and the call never delays. See ~**=**.

$Term_1$ == $Term_2$

(Non-logical).

$Term_1$ and *$Term_2$* are identical; that is, they can be unified without binding any variables.

$Term_1$ \== $Term_2$

(Non-logical).

$Term_1$ and *$Term_2$* are not identical. For example, **`A \== B`** would succeed if *A* and *B* were distinct variables.

$Term_1$ @< $Term_2$

(Non-logical).

$Term_1$ is before *$Term_2$* in the non-logical ordering of terms. See **`termCompare/3`**.

$Term_1$ @> $Term_2$

(Non-logical).

$Term_1$ is after *$Term_2$* in the non-logical ordering of terms. See **`termCompare/3`**.

$Term_1$ @=< $Term_2$

(Non-logical).

$Term_1$ is before or equal to *$Term_2$* in the non-logical ordering of terms.

$Term_1$ @>= $Term_2$

(Non-logical).

$Term_1$ is equal to or after *$Term_2$* in the non-logical ordering of terms.

termCompare(Comp, $Term_1$, $Term_2$)

Logical term comparison. *Comp* is <, **`=`** or > if term *$Term_1$* is before, equivalent to, or after term *$Term_2$* respectively, in the standard ordering for terms. Delays until *$Term_1$* and *$Term_2$* are instantiated enough for comparison to be made under the standard ordering, that is, until any further instantiation would not change the ordering already determined. Fails if *Comp* will not unify with <, **`=`** or >.

compare(Comp, $Term_1$, $Term_2$)

(Non-logical).

Comp is <, **`=`** or > if *$Term_1$* is before, equivalent to or after *$Term_2$* respectively, in the non-logical ordering of terms. Fails if *Comp* will not unify with <, **`=`** or >. See **`termCompare/3`**.

5.5. Arithmetic Operators and Predicates

The predicates in this section deal with numbers and arithmetic expressions. Arithmetic expressions may contain numbers, variables bound to arithmetic expressions, strings of length one (which are considered to have the integer value of the ASCII code of the character) and arithmetic operators. Logical expressions evaluate to **`1`** (true) or **`0`** (false). All predicates which use arithmetic expressions delay until all variables in the expression(s) are bound. Expressions may delay unnecessarily, such as **`0 and X`**, which may delay until *X* is ground before failing. Overflow is not detected in arithmetic.

Arithmetic is performed to implementation-determined precisions which may be determined with the predicate **`arithmeticPrecision/3`**. Note that arithmetic may be more accurate when compiled than interpreted.

5.5.1. Arithmetic Operators

The operators in this section are not predicates; they are expressions which are only evaluated when they occur as an argument to the arithmetic predicates described in the next section. Some of these operators correspond to arithmetic predicates which may evaluate expressions. Integers are converted to floats when necessary, but not vice versa.

maxint

The maximum integer representable by Prolog. Evaluated only when it occurs as an argument to an arithmetic predicate.

minint

The minimum integer representable by Prolog. Evaluated only when it occurs as an argument to an arithmetic predicate.

pi

The number pi. Evaluated only when it occurs as an argument to an arithmetic predicate.

N + M

N plus *M*. Evaluated only when it occurs as an argument to an arithmetic predicate.

N – M

N minus *M*. Evaluated only when it occurs as an argument to an arithmetic predicate.

N * M

N times *M*. Evaluated only when it occurs as an argument to an arithmetic predicate.

N / M

N divided by *M* performed in floating-point. Evaluated only when it occurs as an argument to an arithmetic predicate. The result is always a float.

N // M

N divided by M performed in integer arithmetic. Evaluated only when it occurs as an argument to an arithmetic predicate. Evaluation fails with a warning message if either argument evaluates to a float.

N mod M

N modulo M. Evaluated only when it occurs as an argument to an arithmetic predicate. Evaluation fails with a warning message if either argument evaluates to a float.

N ** M

N to the power of M, where $M \geq 0$ or $N < 0$ and M is equal to a (negative) integer. Evaluated only when it occurs as an argument to an arithmetic predicate. The result is always a float.

sin(X)

The sine of X. Evaluated only when it occurs as an argument to an arithmetic predicate. The result is always a float.

cos(X)

The cosine of X. Evaluated only when it occurs as an argument to an arithmetic predicate. The result is always a float.

tan(X)

The tangent of X. Evaluated only when it occurs as an argument to an arithmetic predicate. The result is always a float.

asin(X)

The arcsine of X. Evaluated only when it occurs as an argument to an arithmetic predicate. The result is always a float.

acos(X)

The arccosine of X. Evaluated only when it occurs as an argument to an arithmetic predicate. The result is always a float.

atan(X)

The arctangent of X. Evaluated only when it occurs as an argument to an arithmetic predicate. The result is always a float.

atan2(X, Y)

atan2(X, Y). Evaluated only when it occurs as an argument to an arithmetic predicate. The result is always a float.

exp(X)

E to the power X. Evaluated only when it occurs as an argument to an arithmetic predicate. The result is always a float.

log(X)

The natural logarithm of X. Evaluated only when it occurs as an argument to an arithmetic predicate. The result is always a float.

log10(X)

The base 10 logarithm of X. Evaluated only when it occurs as an argument to an arithmetic predicate. The result is always a float.

sqrt(X)

The square root of X. Evaluated only when it occurs as an argument to an arithmetic predicate. The result is always a float.

integer(X)

The nearest integer to X between X and zero. Evaluated only when it occurs as an argument to an arithmetic predicate.

float(X)

The floating-point equivalent of X. Evaluated only when it occurs as an argument to an arithmetic predicate. The result is always a float.

round(X)

The floating-point representation of the integer nearest to X. Evaluated only when it occurs as an argument to an arithmetic predicate. The result is always a float.

N /\ M

N bitwise and'ed with M. Evaluated only when it occurs as an argument to an arithmetic predicate. Evaluation fails with a warning message if either argument evaluates to a float.

N \/ M

N bitwise or'ed with M. Evaluated only when it occurs as an argument to an arithmetic predicate. Evaluation fails with a warning message if either argument evaluates to a float.

N ^ M

N bitwise exclusive or'ed with M. Evaluated only when it occurs as an argument to an arithmetic predicate. Evaluation fails with a warning message if either argument evaluates to a float.

N << M

N shifted left M positions. Evaluated only when it occurs as an argument to an arithmetic predicate. Evaluation fails with a warning message if either argument evaluates to a float.

N >> M

N arithmetically shifted right *M* positions. Evaluated only when it occurs as an argument to an arithmetic predicate. Evaluation fails with a warning message if either argument evaluates to a float.

+ N

N. Evaluated only when it occurs as an argument to an arithmetic predicate.

– N

Minus *N*. Evaluated only when it occurs as an argument to an arithmetic predicate.

\ N

Bitwise complement of *N*. Evaluated only when it occurs as an argument to an arithmetic predicate. Evaluation fails with a warning message if the argument evaluates to a float.

not N

Logical complement of *N*. Equivalent to comparing *N* with zero. Evaluated only when it occurs as an argument to an arithmetic predicate.

N < M

Evaluates to 1 if *N* is less than *M*, and to 0 otherwise. Evaluated only when it occurs as an argument to an arithmetic predicate.

N =< M

Evaluates to 1 if *N* is less than or equal to *M*, and to 0 otherwise. Evaluated only when it occurs as an argument to an arithmetic predicate.

N > M

Evaluates to 1 if *N* is greater than *M*, and to 0 otherwise. Evaluated only when it occurs as an argument to an arithmetic predicate.

N >= M

Evaluates to 1 if *N* is greater than or equal to *M*, and to 0 otherwise. Evaluated only when it occurs as an argument to an arithmetic predicate.

N =:= M

Evaluates to 1 if the values of arithmetic expressions *N* and *M* are equal, and to 0 otherwise. Evaluated only when it occurs as an argument to an arithmetic predicate.

N =\= M

Evaluates to 1 if the values of arithmetic expressions *N* and *M* are not equal, and to 0 otherwise. Evaluated only when it occurs as an argument to an arithmetic predicate.

N and M

Evaluates to 1 if *N* and *M* are both non-zero, and to 0 otherwise. Evaluated only when it occurs as an argument to an arithmetic predicate.

N or M

Evaluates to 1 if either *N* or *M* is non-zero, and to 0 otherwise. Evaluated only when it occurs as an argument to an arithmetic predicate.

5.5.2. Arithmetic Predicates

The predicates in this section are the only predicates which will evaluate arithmetic expressions; some of them correspond to operators which may be evaluated. Except where indicated otherwise, these predicates fail on backtracking, and if one of their arguments is not a suitable number or arithmetic expression will print a warning and fail when all the arguments become ground. Errors of this type are detected at compile time if possible.

N < M when ground(N) and ground(M)

Succeeds if the value of arithmetic expression *N* is less than the value of arithmetic expression *M*. Prints a message and fails if an argument is bound to a ground term that is not a number or arithmetic expression.

N =< M when ground(N) and ground(M)

Succeeds if the value of arithmetic expression *N* is less than or equal to the value of arithmetic expression *M*. Prints a message and fails if an argument is bound to a ground term that is not a number or arithmetic expression.

N > M when ground(N) and ground(M)

Succeeds if the value of arithmetic expression *N* is greater than the value of arithmetic expression *M*. Prints a message and fails if an argument is bound to a ground term that is not a number or arithmetic expression.

N >= M when ground(N) and ground(M)

Succeeds if the value of arithmetic expression *N* is greater than or equal to the value of arithmetic expression *M*. Prints a message and fails if an argument is bound to a ground term that is not a number or arithmetic expression.

N =:= M when ground(N) and ground(M)

Succeeds if the values of arithmetic expressions *N* and *M* are equal. Prints a message and fails if an argument is bound to a ground term that is not a number or arithmetic expression.

N =\= M when ground(N) and ground(M)

Succeeds if the value of arithmetic expressions *N* and *M* are not equal. Prints a message and fails if an argument is bound to a ground term that is not a number or arithmetic expression.

N and M when ground(N) and ground(M)

Arithmetic expressions *N* and *M* are both non-zero. Prints a message and fails if an argument is bound to a ground term that is not a number or arithmetic expression.

between(N, M, I) when N and M or I

Similar to **`N =< I, I =< M`**, but if *I* is a variable it is successively unified to all of the integers between *N* and *M* inclusively. *N* and *M* must be numbers, they may not be expressions.

divides(N, M, Div, Mod) when N and (M or Div and Mod) or M and Div and Mod

$N \ / \ M = Div$, $N \ mod \ M = Mod$. If *N*, *M*, *Div*, and *Mod* are not sufficiently instantiated the call will delay. The arguments must be integers or variables, they can not be expressions or floats.

N or M when ground(N) and ground(M)

Either or both of the arithmetic expressions *N* and *M* are non-zero. Prints a message and fails if an argument is bound to a ground term that is not a number or arithmetic expression.

I is N when ground(N)

Arithmetic expression *N* evaluates to *I*, a variable or integer. It delays if *N* contains an unbound variable, so **`is`/`2`** can only be used in one direction, unlike **`plus`/`3`**. Similar to conventional arithmetic assignment in *C*. Prints a message and fails if the second argument is bound to a ground term that is not a number or arithmetic expression, or if the first argument is not a number.

iota(N, M, I) when N and M or I

Similar to **`N =< I, I =< M`**, but if *I* is a variable it is successively unified to all of the integers between *N* and *M* inclusively. *N* and *M* must be numbers, they may not be expressions. For compatibility, **`iota`/`3`** has been renamed **`between`/`3`**.

maxint(N)

N is the largest integer possible in the system (**`maxint`**). The smallest is $-N - 1$. Fails if *N* won't unify with **`maxint`**.

plus(N, M, Sum) when N and M or N and Sum or M and Sum

$N + M = Sum$. If more than one argument is a variable, it will delay. If two are integers, then the third will be calculated. The arguments should not be expressions.

random(Num)

Num is a random integer in the range 0 to **`maxint`** (see **`maxint`/`1`**).

times(N, M, Prod) when N and M or N and Prod or M and Prod

$N * M = Prod$. If more than one argument is a variable, it will delay. If two are integers, then the third will be calculated. The arguments should not be expressions or floats.

5.6. I/O Predicates

These predicates are related to input and output. All input and output is via **streams**. The standard streams are **user_input**, **user_output** and **user_error**. The first two can be referred to as **user** where the context makes clear which stream is intended. In addition, is always a current input stream and a current output stream. I/O predicates which do not have a stream as an argument refer to the current stream. Initially, these are **user_input** and **user_output**, but they may be changed without affecting the standard streams. All I/O predicates are non-logical.

Stream identifiers may be manipulated in any way desired. In particular, they may be asserted and retracted. The three standard identifiers are always valid, but attempts to use other stream identifiers, the streams of which have been closed, will fail with an error message.

Predicates which have single characters as their argument, such as **put/1** and **get/1**, assume that the character is an ASCII code, not an atom or expression. **writev/2** gives users the option of printing strings as sequences of characters or as a list of ASCII codes. The ASCII code for a character *Char* is equivalent to **0'Char**. The only characters which may appear as *Char* are symbol characters, printable characters, white space and escape sequences. Character classes are described in Appendix 7.

5.6.1. Stream Management Predicates

absoluteFileName(RelFile, AbsFile)

Given a valid file name *RelFile*, *AbsFile* is bound to the absolute file name corresponding to it. If *RelFile*.nl exists then it is chosen in preference to *RelFile*.

characterCount(Stream, Chars)

(Non-logical).
Unifies *Chars* with the number of characters that have been read or written on *Stream*. A stream has characterCount 0 before any characters are transfered.

Re-positioning *Stream* with **fseek/3** invalidates the counter.

clearIOError(Stream)

(Non-logical).
Used to reset any error flags on *Stream*, including end-of-file. Prints a warning and fails if *Stream* is not a currently open stream.

close(Stream)

(Non-logical).
Closes *Stream*. *Stream* must be obtained from a previous call to **open/3** or any similar predicate. For compatibility, this predicate also accepts an atom representing a filename as its argument, in which case it closes a (possibly arbitrary) stream associated with that file.

If *Stream* cannot be closed a fatal error will be generated, unless file errors have been turned off with **prologFlag/3**, in which case the call fails. Prints a warning and fails if *Stream* is not a currently open stream.

If **user_input** or **user_output** are closed, they are immediately re-opened on the terminal. If the user attempts to close **user_error**, a warning will be printed and the call will fail.

currentInput(Stream)

(Non-logical).
Stream is unified with the current input stream. Fails if *Stream* will not unify with the current input stream.

currentOutput(Stream)

(Non-logical).
Stream is unified with the current output stream. Fails if *Stream* will not unify with the current output stream.

currentStream(File, Mode, Stream)

(Non-logical).
Stream is a currently open stream on *File*, where *Mode* is either **`read`**, **`write`** or **`append`**. On backtracking, **`currentStream`/3** finds all suitable streams. Fails if *Stream* will not unify with a currently open stream of the appropriate type.

The three standard input streams are ignored by this predicate.

fileErrors

(Non-logical).
Resets **`fileErrors`** flag to **`on`**. See **`prologFlag`/3**.

fseek(Stream, Offset, Whence)

(Non-logical).
Moves the file-pointer associated with *Stream* to a new position specified by *Offset* relative to *Whence*. *Whence* is one of **`beginning`**, **`current`**, or **`end`**, referring to the beginning of the file *Stream* is open on, the current position in the file of *Stream*, or the end of the file. The exact meaning of *Offset* is operating system dependent, but is usually the signed number of bytes to move from *Whence*. The only completely reliable way to determine an offset is to get it from **`ftell`/2**.

Stream must be obtained from a previous call to **`open`/3** or any similar predicate. Only streams open to files can be re-positioned. For compatibility, this predicate also accepts an atom representing a filename as its argument, in which case it re-positions a (possibly arbitrary) stream associated with that file.

If *Stream* cannot be re-positioned a fatal error will be generated, unless file errors have been turned off with **`prologFlag`/3**, in which case the call fails. Prints a warning and fails if *Stream* is not a currently open stream.

ftell(Stream, Offset)

(Non-logical).

Offset is the current position with respect to **`beginning`** of the file-pointer associated with *Stream*. The exact meaning of *Offset* is operating system dependent, but is usually the number of bytes from the beginning of the file.

Stream must be obtained from a previous call to **`open/3`** or any similar predicate. *Offset* is only meaningful for streams open on files. For compatibility, this predicate also accepts an atom representing a filename as its argument, in which case it gives the position of a (possibly arbitrary) stream associated with that file.

Prints a warning and fails if *Stream* is not a currently open stream.

lineCount(Stream, Lines)

(Non-logical).

Unifies *Lines* with the number of lines that have been read or written on *Stream*. The first line on a stream is numbered 1.

Re-positioning *Stream* with **`fseek/3`** invalidates the counter.

linePosition(Stream, LinePosition)

(Non-logical).

Unifies *LinePosition* with the number of characters read or written since the last new-line on *Stream*. The first character on a line is numbered 0.

Re-positioning *Stream* with **`fseek/3`** invalidates the counter.

noFileErrors

(Non-logical).

Sets **`fileErrors`** flag to **`off`**. See **`prologFlag/3`**.

open(File, Mode, Stream)

(Non-logical).

File is an atom specifying a filename. *Mode* is one of the atoms **`read`** (for input), **`write`** (for writing on new files) and **`append`** (for adding to an existing file). If **`open/3`** succeeds, *Stream* is unified with the resulting input stream. If *File* cannot be opened in the specified *Mode*, a fatal error is generated unless the **`fileErrors`** flag is set to **`off`** (see **`prologFlag/3`**), in which case the call fails. Fails if *Stream* cannot be unified with the stream returned by **`open/3`**.

openNullStream(Stream)

(Non-logical).

Stream is unified with a null output stream. Output to this stream is discarded. Fails if *Stream* will not unify with a currently open null stream.

setInput(Stream)

(Non-logical).

Set current input to *Stream*. Prints warning and fails if *Stream* is not a currently open input stream.

setOutput(Stream)

(Non-logical).
Set current output to *Stream*. Prints warning and fails if *Stream* is not a currently open output stream.

sourceFile(File)
sourceFile(Pred, Arity, File)

True if *File* is a user-loaded source file, or if *Pred*/*Arity* is a predicate in the file *File*. On backtracking, finds another appropriate predicate or file.

streamEof(Stream)

(Non-logical).
Succeeds if the end of file has been seen on *Stream*.

streamPosition(Stream, OldPosition, NewPosition)

(Non-logical).
Unifies *OldPosition* with an object describing the current position of *Stream* and re-positions it to *NewPosition*. Positions are internal objects whose representation is implementation dependent. The only valid way to create one is with **`streamPosition`**/**`3`**. **`StreamPosition`**/**`3`** fails if *NewPosition* is not a valid position, though the check is so placed that the idiom **`streamPosition(Stream, Position, Position)`** finds the current position and leaves it unchanged.

To move to a random position in a file, try **`fseek`**/**`3`**.

5.6.2. Input Predicates

The predicates in this section read from current input (or the given stream) either characters, or terms terminated by a period and white space. At end-of-file, either **`-1`** (character input) or the atom **`end_of_file`** (term input) is returned.

If the *Stream* argument to any of the following predicates is not a currently open stream, a warning will be printed and the predicate will fail.

isEof(Term)

Equivalent to **`nonvar(Term), Term = end_of_file`**, but portable. Does not catch end-of-file marker from character input. If *Term* is a variable, **`isEof`**/**`1`** fails.

eof(Term)

Equivalent to **`Term = end_of_file`**, but portable. Does not catch end-of-file marker from character input. If *Term* is a variable, **`eof`**/**`1`** succeeds.

get(Char)
get(Stream, Char)

(Non-logical).

Reads ASCII characters from the current input stream (or *Stream*) and returns *Char*, the ASCII code for the first character read which has ASCII code greater than 32. If there are no such characters, *Char* is unified with **-1**. If end-of-file was reached by a previous call to an input predicate on this stream, a fatal error is generated. Fails if *Char* will not unify with the character read.

get0(Char)
get0(Stream, Char)

(Non-logical).

Char is the ASCII code for the next character to be read from the current input stream (or *Stream*). At end-of-file *Char* is unified with -1. If end-of-file was reached by a previous call to an input predicate on that stream, a fatal error is generated. Fails if *Char* will not unify with the character read.

getl(Line)
getl(Stream, Line)

(Non-logical).

Line is a list of the ASCII codes of the next characters to be read from the current input stream (or *Stream*) up to and including the next new-line character. If end-of-file is reached before a new-line *Line* will not have a new-line as its last character. If end-of-file was reached by a previous call to an input predicate on that stream, a fatal error is generated. Fails if *Line* will not unify with the line read.

getToken(Token, Type)
getToken(Stream, Token, Type)

(Non-logical).
Read the next NU-Prolog *Token* of *Type* from the current input stream (or *Stream*). *Type* is an atom.

Token	Type
[]	end_of_file
VarName	var
Atom	atom
QAtom	quoted
Number	number
String	string
End	end
ASCII	junk

where *VarName* is a string representing the name of a variable, *Atom* is any atom, *QAtom* is an atom that was quoted, *Number* is a number, *String* is a string, *End* is **'. '** — the token returned when the end of a term is encountered, and *ASCII* is a character code not amongst the unescaped permitted characters listed in Appendix 7.

Left parentheses are tokenized as the atom '**(**' with type atom if they are not immediately preceded by a token that could serve as an operator. Periods followed by whitespace (the term terminator) tokenize as **.** , also with type atom. Both can be distinguished from the quoted atom appearing in input by their type.

getTokenList(Tokens)
getTokenList(Stream, Tokens)

(Non-logical).
Reads NU-Prolog token and type pairs from the current input stream (or *Stream*) using **getToken/2** until a term terminator or end of file is encountered. Any term terminator read is discarded, the tokens and types read are consed together as pairs and a list made of them.

read(Term)
read(Stream, Term)

(Non-logical).
Read a NU-Prolog *Term*, terminated by a period and white space, from the current input stream (or *Stream*). If read/1 finds a token sequence terminated by a full stop, but that token sequence cannot be parsed as a term using the normal rules of Prolog syntax and the current operator declarations, it reports a syntax error and skips that token sequence. It keeps on skipping tokens sequences until it finds one that it can parse. It never skips well formed terms. If *Term* contains variables these are distinct from variables not in the term. At end-of-file, **read/1** and **read/2** return the atom **end_of_file**. Fails if *Term* will not unify with the term that is read.

readTerm(Term, NameList, VarList)
readTerm(Stream, Term, NameList, VarList)

(Non-logical).
Read *Term* from the standard input stream (or *Stream*) as for **`read`/`1`** or **`read`/`2`**. *VarList* is bound to a list of the variables in *Term*. *NameList* is bound to a list of the names of the variables represented as strings; variables called '_' (underscore) are not included in *NameList* or *VarList*. Fails if *Term*, *NameList* or *VarList* won't unify with the terms returned by **`readTerm`/`3`** or **`readTerm`/`4`**.

read1(Term)
read1(Stream, Term)

(Non-logical).
Read a NU-Prolog *Term*, terminated by a period and white space, from the current input stream (or *Stream*). If *Term* contains variables these are distinct from variables not in the term. At end-of-file, **`read1`/`1`** and **`read1`/`2`** return the atom **`end_of_file`**. Unlike **`read`/`1`** and **`read`/`2`**, **`read1`/`1`** and **`read1`/`2`** print a warning and fail if a syntax error is encountered. Fails if *Term* will not unify with the term that is read.

read1Term(Term, NameList, VarList)
read1Term(Stream, Term, NameList, VarList)

(Non-logical).
Read a term *Term* from the standard input stream (or *Stream*) as for **`read1`/`1`** or **`read1`/`2`**. *VarList* is bound to a list of the variables in *Term*. *NameList* is bound to a list of the names of the variables represented as strings; variables called '_' (underscore) are not included in *NameList* or *VarList*. Fails if *Term*, *NameList* or *VarList* won't unify with the terms returned by **`read1Term`/`3`** or **`read1Term`/`4`**.

see(File)

(Non-logical).
Switches current input to either the file or stream specified by *File*. If *File* is a stream, but not a currently open input stream, an error is given. If *File* is an atom then it is treated as a file; if there are no open input streams associated with that file then one is opened. If there are more than one, an arbitrary one is chosen as current input. If *File* is neither a stream nor an atom **`see`/`1`** fails.

seeing(Stream)

(Non-logical).
Stream is unified with the current input stream. Same as **`currentInput(Stream)`**.

seen

(Non-logical).
Closes the current input stream, which reverts to **`user_input`**.

skip(Bound)
skip(Stream, Bound)

(Non-logical).
Reads characters from the current input stream (or *Stream*) until character *Bound* appears or end-of-file is reached. If *Bound* is a list, then characters are read until a member of *Bound* is found; *Bound* should therefore be a list of ASCII codes. If *Bound* is neither of these things, it will read characters until end-of-file is reached.

sread(String, Term)

(Non-logical).
Parse *String* to give *Term*. Fails if *String* is not a valid term. Unlike input to **`read/1`**, the term in *String* should not be terminated with a period.

sreadTerm(String, Term, NameList, VarList)

(Non-logical).
Parse *String* yielding *Term* as for **`sread/2`**, where *NameList* and *VarList* are a list of the names of the variables, and a list of the variables, in *Term*, respectively. Variables named '_' are not included in *NameList* or *VarList*. Fails if *String* is not a valid term. Unlike input to **`read/1`**, the term in *String* should not be terminated with a period.

tokenize(StringIn, Token, Type, StringOut)
tokenize(String, Tokens)

(Non-logical).
`Tokenize/4` reads the first *Token* with *Type*, from *StringIn*, leaving any unread characters in *StringOut*. **`Tokenize/2`** converts all of *String* into a list of *Token*`.`*Type* pairs. Tokens and types are as described for **`getToken/3`**. Fails if any argument is of an inappropriate type.

tread(TokenList, Term)

(Non-logical).
Parse *TokenList*, a list of token and type pairs such as that produced by **`getTokenList/1`**, to give *Term*. Fails if *TokenList* is not a valid term. Unlike input to **`read/1`**, the term in *TokenList* should not be terminated with a term terminator.

treadTerm(TokenList, Term, NameList, VarList)

(Non-logical).
Parse *TokenList* yielding *Term* as for **`tread/2`**, where *NameList* and *VarList* are a list of the names of the variables, and a list of the variables, in *Term*, respectively. Variables named '_' are not included in *NameList* or *VarList*. Fails if *TokenList* is not a valid term. Unlike input to **`read/1`**, the term in *TokenList* should not be terminated with a term terminator.

ttyget(Char)

(Non-logical).
Read the next printable character from **`user_input`**. Same as **`get(user_input, Char)`**.

ttyget0(Char)
(Non-logical).
Read a character from `user_input`. Same as `get0(user_input, Char)`.

ttyskip(Bound)
(Non-logical).
Reads characters from `user_input` until character *Bound* appears, or end-of-file is reached. If *Bound* is a list, then characters are read until a member of *Bound* is found. Same as `skip(user_input, Bound)`.

5.6.3. Output Predicates

If the *Stream* argument to any of the following predicates is not a currently open output stream, a warning will be printed and the predicate will fail. All of the predicates in this section fail on backtracking.

display(Term)
display(Stream, Term)
(Non-logical).
Write *Term* on the current output stream (or *Stream*) in prefix format. Similar to `writev/2` (or `writev/3`).

flushOutput(Stream)
(Non-logical).
Flush the specified output *Stream*, causing any buffered output to appear immediately.

format(Format, Arguments)
format(Stream, Format, Arguments)

(Non-logical).

`Format/2,3` is a general formatted output routine. It takes a list of *Arguments* and a *Format*, a string or atom, describing how they are to be printed. Output is either to the current output or to *Stream*.

Format is composed of characters to be printed and of format control directives that cause the printing of one or more of the *Arguments*. The format controls are character sequences of the form *~nc* where *c* is one of the format control characters listed below and *n* is an optional small integer controlling precision, repetition, radix, or fill character. *N* can be specified as either a decimal number, the character **`*`** which causes the number to be supplied by the next unprocessed argument, or **`'`***x* meaning the ASCII code of the single character *x*. The formats that take a numeric argument have defaults if it is absent.

`Format` reports an error if the type of the argument given does not match that expected by the format. Formats expecting numeric arguments evaluate them.

The format control characters are

~**`a`** Print the argument, and atom, without quotes.

~*n***`c`** Print the character specified by the ASCII code given by the argument *n* times. *N* defaults to one.

~*n***`e`** Print the argument, a floating point number, in exponential notation with one digit before the decimal point and *n* digits after it. *N* defaults to six.

~*n***`E`** This is similar to ~**`e`**, but the letter **`E`** rather than **`e`** is used to indicate the exponent.

~*n***`f`** Print the argument, a floating point number, in fixed-point notation with and *n* digits after the decimal point. *N* defaults to six. If *n* is zero, the decimal point is omitted.

~*n***`g`** Print the argument, a floating point number, in either exponential or fixed-point notation, depending on which is smaller. At most *n* significant digits are printed. *N* defaults to six.

~*n***`G`** This is similar to ~**`g`**, but the letter **`E`** rather than **`e`** is used to indicate the exponent.

~*n***`d`** Print the argument, an integer, as a decimal number. If *n* is not zero, a decimal point is inserted *n* digits from the right-hand side. *N* defaults to zero.

~*n***`D`** This is similar to ~**`d`**, but commas are inserted to group the integer part of the number printed.

~*n***`r`** Print the argument, an integer, as a number in radix *n* using the digits **`0`**–**`9`** and the letters **`a`**–**`z`**. This gives valid radixes in the range two to 36. *N* defaults to eight.

~*n***`R`** This is similar to ~**`r`**, but it uses **`A`**–**`Z`** rather than **`a`**–**`z`**.

~*n***`s`** Print the first *n* characters of the argument, a string. Strings shorter than *n* are padded with spaces on the left. *N* defaults to the length of the string.

~**`i`** Ignore the argument.

~**`k`** Pass the argument to **`writeCanonical/1`**.

~**`p`** Pass the argument to **`print/1`**.

~**`q`** Pass the argument to **`writeq/1`**.

~**`w`** Pass the argument to **`write/1`**.

~~ Print one ~.

~*n***`n`** Print *n* newlines. *N* defaults to one.

~**`N`** Print a newline if not at the beginning of a line, and nothing otherwise.

~*n* **|** Set a tab at the *n*th character on the current line. *N* defaults to the current line position.

~*n*+ Set a tab at *n* characters on from the last one. *N* defaults to eight.

~*n*`t` Establish a fill position with fill character *n* in the current line being printed. When the space between two tabs is not filled by explicitly written characters, fill characters are inserted at the fill positions to make up the difference. *N* defaults to 32 (ASCII space).

nl
nl(Stream)

(Non-logical.) A newline is printed on the current output stream (or *Stream*). Same as `put(0'\n)`.

portraycl(Clause)

(Non-logical).
Write *Clause* with atoms quoted where necessary, and parentheses inserted where necessary to represent precedence. A term written with `portraycl/1` can always be read as a syntactically correct term by the NU-Prolog parser if a ''`.`'' and some white space is appended to it.

portraygoals(Goal)

(Non-logical).
Write *Goal* with atoms quoted where necessary, and parentheses inserted where necessary to represent precedence. A term written with `portraygoals/1` can always be read as a syntactically correct term by the NU-Prolog parser if a ''`.`'' and some white space is appended to it.

print(Term)
print(Stream, Term)
print(Stream, Term, Depth)
print(Stream, Term, Depth, Prec)

(Non-logical).
Print *Term* on the current output (or on *Stream*). If *Depth* is given then sub-terms of *Term* nested deeper than *Depth* are printed as the atom `**depth*bound**`. The elements of a list are treated as of the same depth. If *Prec* is given then parentheses are placed around *Term* if needed to make it parse as an argument of an operator of precedence *Prec*. If the user has defined `portray/1`, this is called. If `portray(Term)` fails or does not exist, `write(Term)` is called. `Print` is used by the debugger to display goals and by the interpreter top-level to show answers.

printf(Format, List)
printf(Stream, Format, List)

(Non-logical).
List is printed with *Format* on the current output stream (or *Stream*). *List* is a list of strings, constants and integers. *Format* is a NU-Prolog string specifying the appearance of the output, where the conventions used are similar to **printf** in the *C* programming language. The types of the elements of *List* should match the format string; if the types do not match, the behaviour will be determined by the local implementation of *C*.

For strings and atoms, the format **%s** is used. **%***n***s** may be used to print a string or atom in a right-justified (left-justified if *n* is negative) field of width *n*. If the string or atom is longer than *n* characters, the overflow will be printed to the right of the field. **%***n***.***m***s** may be used to print *m* characters of a string in a right-justified (left-justified if *n* is negative) field of width *n*.

The format **%c** is used for characters (small positive integers in NU-Prolog). **%***n***c** may be used to print a right-justified (left-justified if *n* is negative) character in a field of width *n*.

The format **%d** is for decimal output of integers. The formats **%***n***d** and **%***n***.***m***d** also apply, where *n* and *m* are as for the format for strings and atoms. Similarly, the formats **%o** and **%x** are for octal and hexadecimal output of integers.

Printf is not portable. Use the more flexible **format/2,3** instead.

put(Char)
put(Stream, Char)

(Non-logical).
The character given by the ASCII code *Char* is written on the current output stream (or *Stream*).

putl(String)
putl(Stream, String)

(Non-logical).
The characters given by the list of ASCII codes in *String* is written on the current output stream (or *Stream*).

sformat(Format, Arguments, String)

Sformat/3 is similar to **format/2** but collects the formatted data in *String*.

For the purposes of ~**N** and ~**|**, **sformat/3** assumes that the formatting is being done at the beginning of a line.

The ~**p** option is not yet implemented.

Sformat/3 reports an error if the type of the argument given does not match that expected by the format.

tab(N)
tab(Stream, N)

(Non-logical).
Writes *N* spaces on the current output stream (or *Stream*). Fails if *N* is not a non-negative integer.

tell(File)

(Non-logical).
Switches current output to the file (or stream) specified by *File*. If *File* is a stream, but not a currently open output stream, an error is given. If *File* is an atom then it is treated as a filename; if there are no open output streams associated with that file, one is opened; if there are more than one appropriate streams, an arbitrary one is chosen. If *File* is neither a stream nor an atom, **`tell/1`** fails.

telling(Stream)

(Non-logical).
Stream is unified with the current output stream. Same as **`current_output(Stream)`**.

told

(Non-logical).
Closes the current output stream, which reverts to **`user_output`**.

ttyflush

(Non-logical).
Same as **`flush(user_output)`**.

ttynl

(Non-logical).
A newline is printed on **`user_output`**. Same as **`nl(user_output)`**.

ttyput(Char)

(Non-logical).
Write a character on **`user_output`**. Same as **`put(user_output, Char)`**.

ttytab(N)

(Non-logical).
Writes *N* spaces on **`user_output`**. Same as **`tab(user_output, N)`**.

write(Term)
write(Stream, Term)

(Non-logical).
Writes term *Term* on the current output stream (or *Stream*), taking into consideration current operator declarations. Same as
`writev([list, string, noquote, ops], Term)`
and
`writev(Stream, [list, string, noquote, ops], Term)`
This implies that lists of integers for which **`isPrint`**/**`1`** is true will be printed as strings. If **`numberVars`**/**`3`** is used to ground *Term* before writing, terms of the form **`$VAR(N)`** will be printed as if they were variables, unless the flag **`vars`** is set to **`off`**; see **`prologFlag`**/**`3`**.

Prints message, but succeeds, if *Stream* is not a valid stream.

writeln(Term)
writeln(Stream, Term)

(Non-logical).
Writes term *Term* on the current output stream (or *Stream*) followed by a newline. Same as
`write(Term), nl`
and
`write(Stream, Term), nl(Stream)`
If **`numberVars`**/**`3`** is used to ground *Term* before writing, terms of the form **`$VAR(N)`** will be printed as if they were variables, unless the flag **`vars`** is set to **`off`**; see **`prologFlag`**/**`3`**.

Prints message, but succeeds, if *Stream* is not a valid stream.

writeCanonical(Term)
writeCanonical(Stream, Term)

(Non-logical).
Writes term *Term* on the current output stream (or *Stream*) in a form in which it can be read back by **`read`**/**`1`**. Terms are written in prefix notation, lists are written using the functor **`.`** and atom **`[]`**, and atoms, and thus functors, are quoted if necessary. **`WriteCanonical`** does not print sub-terms of the form **`$VAR(N)`** as variables.

Same as
`writev([cons, nostring, quote, prefix], Term)`
and
`writev(Stream, [cons, nostring, quote, prefix], Term)`,
but with the **`vars`** flag turned **`off`**.

Provided that the value of the **`characterEscapes`** flag is unchanged and that a period and a space are appended, terms written with **`writeCanonical`** can be read back with **`read`**.

Prints message, but succeeds, if *Stream* is not a valid stream.

writeq(Term)
writeq(Stream, Term)

(Non-logical).
Writes term *Term* on the current output stream (or *Stream*), taking into consideration current operator declarations, and quoting atoms (and therefore functors) where necessary. Same as
`writev(Term, [list, string, quote, ops])`
and
`writev(Stream, Term, [list, string, quote, ops])`.
This implies that lists of integers for which **`isPrint`/`1`** is true will be printed as strings. If **`numberVars`/`3`** is used to ground *Term* before writing, terms of the form **`$VAR(N)`** will be printed as if they were variables, unless the flag **`vars`** is set to **`off`**; see **`prologFlag`/`3`**.

Prints message, but succeeds, if *Stream* is not a valid stream.

writev(Flags, Term)
writev(Stream, Flags, Term)

(Non-logical).
Writes *Term* on the current output stream (or *Stream*), taking into consideration the list of *Flags*, which should be ground, or a warning will be printed and the predicate will fail. Possible values of *Flags* (first value is default) are

list (cons)	Write lists using **`[]`** (or **`.`**) notation.
string (nostring)	Write strings using double quotes (or as list of integers). The default writes lists of integers for which **`isPrint`/`1`** is true as strings.
noquote (quote)	Write atoms and functors without quotes (or use quotes when necessary).
quoteall (noquoteall)	Write all atoms and functors with quotes (or use the setting of **`quote`** flag).
ops (prefix)	Take account of operator declarations (or use prefix notation).
base = r	Write integers in base $2 \leq r \leq 36$. Floats are always written in decimal.
prec = p	Place parentheses around *Term* if it is written in operator form with precedence greater than p.

Values in *Flags* other than those defined above are ignored; where both options are absent, the default is taken. If **`numberVars`/`3`** is used to ground *Term* before writing, terms of the form **`$VAR(N)`** will be printed as if they were variables, unless the flag **`vars`** is set to **`off`**; see **`prologFlag`/`3`**.

Prints message, but succeeds, if *Stream* is not a valid stream.

5.7. Database Predicates

Predicates may be either **`static`** or **`dynamic`**. **`Static`** (or **`compiled`**) predicates cannot be manipulated. **`Dynamic`** predicates can be changed by adding or deleting individual clauses (for example, with **`assert`**/**`1`** and **`retract`**/**`1`**); whereas static predicates can be only changed by redefining the whole predicate, (for example, with **`consult`**/**`1`** or **`load`**/**`1`**). By default, predicates are static.

Dynamic predicates may be either in the internal database or the external database. To make an internal database predicate *Functor*/*Arity* dynamic:

(1) If *Functor*/*Arity* is defined in a file which will be compiled or consulted, declare **`dynamic Functor`**/**`Arity`** before the definition appears in the source.

(2) If *Functor*/*Arity* is created by assertions only, then the first call to **`assert`**/**`1`**, **`asserta`**/**`1`** or **`assertz`**/**`1`** will make *Functor*/*Arity* dynamic.

Dynamic predicates do not have implicit quantification.

5.7.1. Loading and Saving Programs

The predicates in this section handle the loading and saving of files. Filenames are given as atoms or strings, so that if they contain '**`.`**', or other non-alphanumeric characters, they should be quoted. Character classes are described in Appendix 7.

consult(File)

(Non-logical).
File is consulted. The extension **`.nl`** or **`.pl`** is appended to *File* if required. If *File* is **`library(Lib)`**, *Lib* is looked for in the NU-Prolog and user-defined libraries. If *File* is a list of files, then each is consulted in turn. All clauses and definite clause grammar rules in the file are added to the internal database and goals are executed. Predicates defined in *File* supersede predicates which are already defined. Goals are written in the form **`?- goal`**.

ensureLoaded(File)

(Non-logical).
Ensure that *File* has been **`load`**ed. If *File* is a list, **`ensureLoaded`**/**`1`** is applied to each member.

lib File

(Non-logical).
Load *File* from the NU-Prolog library or from one of the user-libraries defined by **`libraryDirectory`**/**`1`**. The extension **`.no`** is appended to *File* if required. Predicate definitions in *File* supersede predicates which are already defined. The NU-Prolog library directory may be changed with **`prologFlag`**/**`3`**.

libdirectory(File)

File is either the atom whose name is the UNIX directory where the NU-Prolog library resides, or the result of **`libraryDirectory(File)`**. The system searches for libraries in the order given by **`libraryDirectory/1`** followed by the NU-Prolog library.

The location of the NU-Prolog library may be changed with **`prologFlag/3`**.

libraryDirectory(Dir)

User-defined predicate listing any library directories to be searched when files are loaded by **`lib/1`** or with the **`library(File)`** convention by **`consult/1`**, **`load/1`**, or **`./2`**. *Dir* is an atom whose name is a UNIX directory where a user library resides.

load(File)

(Non-logical).
Load *File*. *File* is searched for with the extensions **`.no`**, **`.nl`**, nothing, or **`.pl`** and the first found is loaded or consulted as appropriate. The extension **`.no`** is appended to *File* if required. If *File* is **`library(Lib)`**, *Lib* is looked for in the NU-Prolog and user-defined libraries. Predicate definitions in *File* supersede predicates which are already defined. Same as **`[File]`**.

[File$_1$,...,File$_n$]

(Non-logical).
Load the list $File_1, \ldots, File_n$ of object files. The extension **`.no`** is appended if required. If $File_i$ is **`library(Lib`**$_i$**`)`**, Lib_i is looked for in the NU-Prolog and user-defined libraries. Same as **`load(File`**$_1$**`),...,load(File`**$_n$**`)`**. New predicate definitions supersede predicates which are already defined.

5.7.2. Property List Predicates

The following predicates are used in accessing and updating property lists of atoms. In all cases, *Key* must be ground, *Prop* an arbitrary term, and *Atom* must be ground. If it is not, a warning is printed and the predicates fail.

addprop(Atom, Key, Prop)

(Non-logical).
Adds the pair <*Key*, *Prop*> to the end of the property list of *Atom*. *Atom* and *Key* must be ground, or a warning is printed and **`addprop/3`** fails. Same as **`addpropz(Atom, Key, Prop)`**.

addprop(Atom, Key, Prop, Reference)

(Non-logical).
Same as **`addprop/3`**, but gives a *Reference* to the property added. Equivalent as **`addpropz(Atom, Key, Prop, Reference)`**.

addpropa(Atom, Key, Prop)

(Non-logical).

Adds the pair <*Key*, *Prop*> to the beginning of the property list of *Atom*. *Atom* and *Key* must be ground, or a warning is printed and **`addpropa`/3** fails.

addpropa(Atom, Key, Prop, Reference)

(Non-logical).

Same as **`addpropa`/3**, but gives a *Reference* to the property added.

addpropz(Atom, Key, Prop)

(Non-logical).

Adds the pair <*Key*, *Prop*> to the end of the property list of *Atom*. *Atom* and *Key* must be ground, or a warning is printed and **`addpropz`/3** fails.

addpropz(Atom, Key, Prop, Reference)

(Non-logical).

Same as **`addpropz`/3**, but gives a *Reference* to the property added.

getprop(Atom, Key, Prop)

(Non-logical).

Unifies <*Key*, *Prop*> with each of the properties of *Atom*. *Atom* must be ground, or a warning is printed and **`getprop`/3** fails.

getprop(Atom, Key, Prop, Reference)

(Non-logical).

Same as **`getprop`/3**, but gives a *Reference* to the property returned.

properties(Atom, Key, PropList)

(Non-logical).

PropList is a list of all properties with *Key* in the property list of *Atom*. *Atom* must be ground, or a warning is printed and **`properties`/3** fails.

putprop(Atom, Key, Prop)

(Non-logical).

If *Key* does not already appear in the property list of *Atom*, add <*Key*, *Prop*>; otherwise, replace the first existing property associated with *Key* by *Prop*. *Atom* and *Key* must be ground, or a warning is printed and **`putprop`/3** fails.

putprop(Atom, Key, Prop, Reference)

(Non-logical).

Same as **`putprop`/3**, but gives a *Reference* to the property changed.

remprop(Atom, Key)

(Non-logical).

Remove all the properties of *Atom* with *Key*. *Atom* must be ground or a warning is printed and **`remprop`/`2`** fails. Same as **`remprop(Atom, Key, _)`**.

remprop(Atom, Key, Prop)

(Non-logical).

Remove all the properties of *Atom* with *Key* which unify with *Prop*. *Atom* must be ground or a warning is printed and **`remprop`/`3`** fails.

5.7.3. Accessing and Updating Dynamic Predicates

The predicates in this section are used in accessing and updating the database of dynamic clauses stored in the internal or external databases.

abolish(Pred, Arity)

(Non-logical).

All information about *Pred*/*Arity* is removed from the database. It is an error for *Pred* not to be an atom, *Arity* not to be an integer, or for *Pred*/*Arity* to be a system predicate.

assert(Clause)

(Non-logical).

If *Clause* – a rule or fact – is about an external database predicate, add *Clause* to the appropriate external database. Otherwise, add *Clause* to the end of the appropriate predicate definition in the internal database. If the predicate definition is not dynamic, it is removed and made dynamic by the call to **`assert`/`1`**. Same as **`assertz`/`1`**, for internal databases.

assert(Clause, Reference)

(Non-logical).

Same as **`assert`/`1`**, but gives a *Reference* to *Clause*. Does not apply to external databases.

asserta(Clause)

(Non-logical).

Add *Clause* at the start of the appropriate predicate definition in the internal database. If the predicate definition is not dynamic, it is removed and made dynamic by the call to **`asserta`/`1`**.

asserta(Clause, Reference)

(Non-logical).

Same as **`asserta`/`1`**, but gives a *Reference* to *Clause*. Does not apply to external databases.

assertz(Clause)

(Non-logical).
Add *Clause* to the end of the appropriate predicate definition in the internal database. If the predicate definition is not dynamic, it is removed and made dynamic by the call to **`assert`/1**.

assertz(Clause, Reference)

(Non-logical).
Same as **`assertz`/1**, but gives a *Reference* to *Clause*. Does not apply to external databases.

clause(Head, Body)

(Non-logical).
There is a clause in the internal database with *Head* and *Body*; the body of a unit clause is the atom **`true`**. On backtracking, finds another matching *Head* and *Body*.

clause(Head, Body, Reference)

(Non-logical).
Same as **`clause`/2**, but gives a *Reference* to clause with *Head* and *Body*. If *Reference* is given, then *Head* and *Body* are unified with the clause specified by *Reference*.

clauses(Functor, Arity, ClauseList)

(Non-logical).
ClauseList is a list of all clauses in the internal database whose head has *Functor* and *Arity*.

dynamic Functor/Arity
dynamic [F_1/A_1,...,F_n/A_n]

(Non-logical).
Remove any previous definition of predicate *Functor*/*Arity* (or $F_1/A_1,...,F_n/A_n$), and declare *Functor*/*Arity* (or $F_1/A_1,...,F_n/A_n$) to be dynamic.

erase(Reference)

(Non-logical).
The dynamic clause with *Reference* is removed from the internal database. If no such clause exists then **`erase`/1** fails.

instance(Reference, Term)

(Non-logical).
Term is most general instance of the property given by *Reference*.

instance(Reference, Key, Term)

(Non-logical).
Term is most general instance of the property given by *Reference*. *Key* specifies which key the property has been stored under (see **`putprop`/3**).

record(Key, Term, Ref)

(Non-logical).

Term is recorded in the internal database as the last item attached to *Key*, which must be instantiated. *Ref* is a database reference to the recorded *Term*. If *Key* is a term, only the principle functor is significant, so that **`p(a)`** represents the same key as **`p(1)`**, but a different key from **`p(c, d)`**. Same as **`recordz`/3**.

recorda(Key, Term, Ref)

(Non-logical).

Term is recorded in the internal database as the first item attached to *Key*, which must be instantiated. *Ref* is a database reference to the recorded *Term*. If *Key* is a term, only the principle functor is significant, so that **`p(a)`** represents the same key as **`p(1)`**, but a different key from **`p(c, d)`**.

recorded(Key, Term)

(Non-logical).

Term has been recorded in the internal database against *Key*, which must be instantiated. On backtracking, **`recorded`/2** will find further matching terms. If *Key* is a term, only the principle functor is significant, so that **`p(a)`** represents the same key as **`p(1)`**, but a different key from **`p(c, d)`**.

recorded(Key, Term, Ref)

(Non-logical).

Term with database reference *Ref* has been recorded in the internal database against *Key*, which must be instantiated. If *Key* is a term, only the principle functor is significant, so that **`p(a)`** represents the same key as **`p(1)`**, but a different key from **`p(c, d)`**.

recordz(Key, Term, Ref)

(Non-logical).

Term is recorded in the internal database as the last item attached to **`Key`**, which must be instantiated. *Ref* is a database reference to the recorded *Term*. If *Key* is a term, only the principle functor is significant, so that **`p(a)`** represents the same key as **`p(1)`**, but a different key from **`p(c, d)`**. Same as **`record`/3**.

retract(Clause)

(Non-logical).

The first clause that matches *Clause* is removed from the internal (or an external) database. On backtracking, the next matching clause is removed.

retractall(Head)

(Non-logical).

Retracts all clauses whose heads match *Head*.

retractall(Functor, Arity)

(Non-logical).

Retracts all clauses where the head has *Functor* and *Arity*.

5.8. Predicates for External Databases

This section describes the predicates available for using external databases to store NU-Prolog predicates. Database predicates behave in the same way as ordinary NU-Prolog predicates with the following restrictions. The order of the clauses cannot be controlled by the programmer. Concurrent reading and writing of external database predicates is not well defined in NU-Prolog. Sometimes it is necessary to add a **`cut`** to avoid these problems. For example, in the subgoal **`p(X), assert(p(Y))`**, the call to **`p(X)`** is still active for reading when **`assert`/`1`** is called. If '**`!`**' is called after the subgoal **`p(X)`**, the system terminates that database access, and **`assert`/`1`** can be called safely.

5.8.1. Creating and Using External Databases

A NU-Prolog deductive database consists of a number of database relations and another rules file which may contain other predicate definitions. A relation can be defined to have any of the available indexing schemes using the parameter **`scheme`**; possible values for **`scheme`** are: **`dsimc`** - this is the default value, (dynamic superimposed coding, described in §5.8.1), **`simc`** (static superimposed coding, 5.8.2), **`sql`** (UNIFY, 5.8.3) and **`rule`** (compiled rules, 5.8.4). Each of these indexing schemes has various parameters, which are described in the subsequent sections. (Note, some of the following predicates do not apply to the **`sql`** indexing schemes because **`sql`** relations can be created from UNIFY.)

dbBackup(Db, File)

(Non-logical).

File is a backup copy of database *Db* (using the UNIX utility *tar*(1)). Use **`dbRestore`/`2`** to restore a database which has been backed up.

dbCons(Db)

(Non-logical).

Deductive database *Db* is consulted. All predicates in the database become accessible and the rules file is consulted. These predicates replace any existing relations with the same functor and arity. Any changes to the database relations, such as with **`assert`/`1`**, update the disc file and are therefore permanent. Changes to predicates defined in the rules file, like changes to normal predicates, are lost at the end of the prolog session.

dbCreate(Db)

(Non-logical).

An empty database named *Db* is created. *Db* must be an atom. A UNIX directory of that name is created and files within it will be used to store all information held in the database. Fails if unable to create database.

dbDefine(Db, Functor, Arity)
dbDefine(Db, Functor, Arity, [Parameter$_1$ = Value$_1$,...])

(Non-logical).

Creates a predicate in the external database *Db* with *Functor* and *Arity*, which can be used to reference the predicate from within Prolog programs. The first form is for use from the top-level of the interpreter, and interactively asks the user about various parameters. In the second form, the values of zero or more parameters can be specified; if a parameter is not specified a default value is assumed.

dbParam(Db, Functor, Arity, Parameter = Value)
(Non-logical).
The current value of *Parameter* for predicate *Functor*/*Arity* in *Db* is *Value*.

dbRestore(Db, File)
(Non-logical).
Db is restored from *File*. *File* must have been created using **`dbBackup/2`**.

dbRules(Db, File)
(Non-logical).
Adds *File* of rules to database *Db*. The previous rules file is overwritten.

dbUndefine(Db, Functor, Arity)
(Non-logical).
Removes the predicate with *Functor*/*Arity* from database *Db*.

5.8.2. Dynamic Superimposed Codeword Databases

A predicate which is defined as a **`dsimc`** predicate using **`dbDefine`** can only store ground unit clauses (facts). The order in which facts are stored is determined by the hashing scheme, not the user. Queries containing more than one **`dsimc`** predicate can be optimized using **`dsimcQuery/1`**. This uses the superjoin optimization [Thom86], which is automatically applied when using the '**`:`**' syntax from the toplevel of the interpreter.

For **`dsimc`**, parameters include:

`avrec`	average record length (default is 8 * arity)
`nrec`	initial number of records
`segsize`	segment size (default is 4096)
`ndata`	number of data files (default is 1)
`nr`	number of records per segment (default computed from **`segsize`** and **`avrec`**)
`lc`	number of records per segment (default computed by **`nr`** * 0.8)
`br`	total number of bits in record codewords (default computed from **`nr`**, **`segsize`** and **`avrec`**)
`bs`	total number of bits in segment codewords (default computed by **`segsize`** / 4)
`kr`	number of bits set in record codewords (default is 11)
`ks`	number of bits set in segment codewords (default computed from **`bs`** and **`nr`**)
`ur`	number of bits to use to determine matching records (default is **`kr`**)
`us`	number of bits to use to determine matching segments (default is **`ks`**)
`template`	template

The larger the descriptors, the better the retrieval performance, as fewer ''false matches'' (which the Prolog unification routine has to filter out) will be retrieved. Of course, larger descriptors require more storage. The number of bits which are set also controls the accuracy of hashing, but it is expensive to set more bits, and the chance of generating false segment matches increases as more bits are set and overlayed in the segment descriptor. The greater the number of bits used (**us**, **ur**), the more accurate the hash, but more time is needed to test for a greater number of bits.

Template takes the same form as the predicate would take when used in the program in prefix format, but each element in the template is a tuple describing properties of the corresponding argument to the predicate. Each element in **template** has the form **Flag:Nbits:Mask**.

Flag must take the value **g**, which means that only ground values will be stored in that field.

Nbits indicates relatively how many bits to use for this field in the superimposed codewords. These values are normalized by the database system. A field which is often used in queries should be assigned a value which is much larger than the value for a field which is used more rarely. The value can be **0**, which means that the value in the field is never considered when retrieving data from the database.

Mask specifies how much weight is to be associated with the field in clustering records within the database.

To determine which segment should hold a new record, the system ANDs the cluster mask with a hash value for the corresponding field. The results for all fields are then OR'ed together to generate the segment number.

To clarify things a bit, consider the following example of a predicate describing an arc in a graph. Each record in this predicate looks like **connect(arc_id, node$_1$, node$_2$)**, where **arc_id** is a unique identifying number for an arc. There are going to be around 30000 facts in this database, and each fact will be around 20 characters long, so using optimisation formulas the following set of parameters can be derived:

br	bs	kr	ks	ur	us	nr	nseg	ndata	avrec
47	13934	10	3	10	3	237	96	1	20

Since the most common type of query to be asked on this predicate involves finding an **arc_id**, given the two nodes, then more bits of the codeword should be allocated to the second and third arguments. The first field is an uninteresting value which is never used in queries, and hence *Nbits* is relatively unimportant, and has not many bits set in the cluster mask. The other two fields are equally likely to be used in a query and have the same relatively high *Nbits* value, and the same number of bits in the cluster mask. The initial **g:0:0** represents the functor, and should always have this value; it is required by the system merely for the sake of completeness.

connect(arc_id,   node$_1$,   node$_2$)

g:0:0(g:2:2040,g:8:4d9b,g:8:9224)

dsimcQuery(Query)
Query is a NU-Prolog query to be optimized using the superjoin algorithm and reordering.

5.8.3. Static Superimposed Codeword Databases

Simc databases [Rama86] can contain unit clauses which need not be ground.

For **simc**, parameters include:

avrec	average record length (default is 8 * arity)
nrec	maximum number of records
segsize	segment size (default is 4096)
ndata	number of data files (default is 1)
nr	number of records per segment (default computed from **segsize** and **avrec**)
nseg	number of segments (default computed by **nrec** / **nr**)
br	total number of bits in record codewords (default computed from **nr**, **segsize** and **avrec**)
bs	total number of bits in segment codewords (default computed from **nr**, **nseg** and **ks**)
kr	number of bits set in record codewords (default is 11)
ks	number of bits set in segment codewords (default is 4)
ur	number of bits to use to determine matching records (default is **kr**)
us	number of bits to use to determine matching segments (default is **ks**)
template	template

Template is the same as for **dsimc** except *Flag* indicates whether or not variables are to be stored in this field; this permits optimisation of database access time. A value of **g** for the **Flag** means that only ground values will be stored in that field whereas **v** means that it may be used to store variables.

5.8.4. UNIFY Databases

NU-Prolog includes an interface for communication via pipes with UNIFY, so that data stored in UNIFY databases can be accessed and modified from NU-Prolog [Zobe86]. To use this interface, no modifications need to be made to UNIFY, except that it must be forced to flush standard output after printing a prompt. This ensures that NU-Prolog can always detect the end of a stream of answers. The depth of a recursive program which makes queries on a UNIFY database is strongly governed by the number of pipes the system can have open at any time; on AT&T's System V UNIX, the limit is seven UNIFY processes.

The database schema of UNIFY databases cannot be modified from within NU-Prolog; as UNIFY provides only an interactive, screen-oriented method for defining database schemas, it is not possible to use **dbDefine/3** to set up the schemas. The UNIX command

initdb *database*

where *database* is the UNIX pathname of the UNIFY database, will create some small files that are used by NU-Prolog when communicating with UNIFY. This command must be given each time the UNIFY schema is modified.

A number of levels of access to UNIFY databases from NU-Prolog are provided, permitting users to choose between SQL queries in which the returned values are bound to Prolog variables, and using a Prolog-like syntax; the former being more efficient, the latter being more straightforward for the NU-Prolog user. In addition, a method is provided for giving a group of Prolog predicates to the interface, to be compiled into a single SQL query. There is also a preprocessor which transforms NU-Prolog predicates into the more efficient low-level predicates.

The interface will map predicates, in which the functor is the name of a UNIFY relation and the arity is the number of attributes of that relation, into SQL queries, and pass them to UNIFY. NU-Prolog always attempt to transform the binding of a ground term into the appropriate type for that attribute before passing it to UNIFY.

sqlQuery(Database, PredList)

Database is the name of the UNIFY database to be accessed. *PredList* is a list of Prolog predicates. These may be either database predicates, of the form given in the previous section, or arithmetic comparisons such as **`A < B`** or **`12 * Qty >= Supply`**. This list is compiled into a single SQL query which is passed to UNIFY.

UNIFY Databases may be modified with the predicates **`assert/1`**, **`retract/1`** and **`retractall/1`**, where the usage for these predicates is as for internal databases. The only difference in behaviour to that of internal databases is that the argument to **`assert/1`** must be ground, and that it is difficult to implement **`retract/1`** efficiently. The SQL command **`delete`** is similar to **`retractall/1`** in Prolog, but there is no direct equivalent to **`retract/1`**. To implement **`retract/1`** correctly, it is necessary to issue two queries; the first finds bindings for any variables in the argument, and the second performs the deletion. It is therefore very expensive to use **`retract/1`** and backtracking to delete a number of predicates.

The predicates in this section provide low-level communication between NU-Prolog and UNIFY. These would be most useful for defining views on databases, which can be done more efficiently in SQL than in Prolog syntax, where a number of UNIFY processes and extensive backtracking might be required to describe the view.

sqlModify(Database, Query)

Database is the database to which the query is being given. *Query* is a string representing a database update expressed in SQL.

sqlAccess(Database, Vars, Query)

Database is the database to which the query is being given. *Vars* is a list of variables for which *Query* will get bindings. *Query* is a string representing an arbitrary UNIFY SQL query (without the '/' terminator). On backtracking, **`sqlAccess(Database, Vars, Query)`** will get new bindings for *Vars*. If the list of variables does not match the number of attributes returned from UNIFY, this predicate will fail expensively, retrieving each set of bindings from UNIFY and failing because they can't be unified with *Vars*. If some of the *Vars* are already bound, any tuple returned from UNIFY with a value which does not match the binding will be thrown away. If any or all of *Vars* is ground, *Query* is optimized. NU-Prolog always attempts to transform the binding of a ground variable into the appropriate type before passing it to UNIFY.

The NU-Prolog system includes a preprocessor, available as the **`-U`** option to *nc*, which examines NU-Prolog programs for UNIFY database predicates and conjunctive groups of database predicates, and translates them into the more efficient **`sqlAccess/3`** format before compilation. This avoids expensive reparsing at each invocation of the predicate.

This preprocessor is most useful if applied as an optimizer to a tested program, rather than applied repeatedly to software under development. This preprocessor could be integrated with program and query optimizers, thus generating more efficient SQL queries.

UNIFY provides no user-level locking primitives. Therefore, transactions cannot be defined for Prolog queries on UNIFY databases while other users have direct access to UNIFY.

5.8.5. Compiled Rule databases

`rule` databases allowed compiled NU-Prolog predicates to be accessed using the external mechanism. **`rule`** databases have no additional parameters.

5.9. Miscellaneous Predicates

currentAtom(Atom)
currentAtom(Module, Atom)

(Non-logical).
Atom is an atom in the module **`user`** (or *Module*).

currentModule(Module)

(Non-logical).
Module is the current module. Currently, only the module **`user`** exists. All atoms and predicates defined in this manual are in this module.

currentOp(Precedence, Type, Op)

(Non-logical).
Op is an operator of specified *Type* and *Precedence*. Fails if any of the arguments is of an inappropriate type. See §3.8.

currentPredicate(Name, Arity)
currentPredicate(Module, Name, Arity)

(Non-logical).
Name/*Arity* is the name of a predicate in the module **`user`** (or *Module*).

expandTerm($Term_1$, $Term_2$)

If the user has defined a predicate **`termExpansion/2`**, this is called and **`expandTerm/2`** commits to its first translation of $Term_1$. Otherwise, if $Term_1$ is a definite clause grammar rule **`expandTerm/2`** translates it into $Term_2$, an ordinary NU-Prolog clause, failing if the rule is not well-formed. If $Term_1$ is not a definite clause grammar rule, **`expandTerm/2`** translates it to itself.

Definite clause grammars are explained in Appendix 2.

termExpansion($Term_1$, $Term_2$)

User defined translation of $Term_1$ to $Term_2$ used by expandTerm/2

libraryPredicate(Library, Predicate)
libraryPredicate(Library, Functor, Arity)

True if *Predicate* (or *Functor*/*Arity*) is a predicate in *Library*. On backtracking, finds another appropriate predicate. See **`predicateProperty/3`**.

nonlogicalPredicate(Predicate)
nonlogicalPredicate(Functor, Arity)

True if *Predicate* (or *Functor*/*Arity*) is a nonlogical system predicate. These do not include library predicates. On backtracking, finds another appropriate predicate. See **`predicateProperty/3`**.

numberVars(Term, N, M) when N

(Non-logical).

Instantiate all variables in *Term* to a term of the form **`$VAR(Num)`**. *N* must be an integer. After the call, all *Num* values will in the range from *N* to $M - 1$. Fails if *N* is not an integer or if $M - N$ is not equal to the number of variables in *Term*. See **`varNumbers`/2** and **`prologFlag`/3**.

op(Precedence, Type, Op)

(Non-logical).

Declares *Op*, an atom or list of atoms, to be an operator with specified *Type* and *Precedence*. It is possible to declare *Op* as only one unary and/or one binary operator. If *Op* is declared as both a unary and binary operator, both must have the same precedence. See §3.8.

phrase(Phrase, TokenList)

`Phrase`/2 is the usual way of calling definite clause grammar rules. *TokenList* is parsed as a phrase of the type *Phrase*, which may be either a non-terminal for which grammar rules are already defined or, in general, the body of a grammar rule. In any case, *Phrase* must be non-variable.

predicateProperty(Name, Arity, Property)

The predicate with *Name* and *Arity* has *Property*. Current properties are **`built_in`**, **`compiled`**, **`dynamic`**, **`database`**, **`library`**, **`pure`** and **`system`**.

Of these **`built_in`** and **`system`** are equivalent – both are included for compatibility with other systems.

prologFlag(Flag, Value)

(Non-logical).

Enquires about the *Value* of *Flag*

prologFlag(Flag, $Value_1$, $Value_2$)

(Non-logical).
Change *$Value_1$* of *Flag* to *$Value_2$*. If *$Value_1$* and *$Value_2$* are identical, **prologFlag/3** fails without changing the value of *Flag*.

Flag	Default Value	Other Values
characterEscapes	on	off
debugging	debug	off
fileErrors	off	on
libdirectory	system dependent	filename
optimizeQuery	off	on
redefinitionWarning	on	off
vars	on	off

CharacterEscapes controls expansion of character escapes (defined in Appendix 7) on input. **Debugging** controls whether or not spy-points (described in §5.3) are in effect. When **fileErrors** is **on**, failure to open or close a file results in an error message and a call to **abort/0**. **Libdirectory** specifies the name of the NU-Prolog library directory. **OptimizeQuery** controls whether queries to the top level database query facility will be affected by various query transformations (re-ordering, superjoin, etc.). **RedefinitionWarning** controls the printing of warnings when system predicates are redefined. When **vars** is **on**, terms of the form **$VAR(N)** are printed as variable names.

statistics

(Non-logical).

statistics(Statistics)

(Non-logical).
Provides statistics about NU-Prolog's use of resources. *Statistics* is a list of elements of the form **Name=Value** where *Name* is an atom and *Value* is a list of numbers. Values of *Name* include

memory	[total memory, 0]
program	[total program space allocated, 0]
global	[global stack in use, global stack free]
local	[local stack in use, local stack free]
trail	[trail stack in use, trail stack free]
utime	[cpu time used, cpu time used since last call to **statistics**]
stime	[system cpu time used, system cpu time used since last call to **statistics**]
time	sum of utime and stime

Times are given in milliseconds, and sizes in bytes. The exact definitions of some of these quantities may vary from machine to machine.

systemPredicate(Predicate)
systemPredicate(Functor, Arity)

True if *Predicate* (or *Functor/Arity*) is a system predicate. These do not include library predicates. On backtracking, finds another appropriate predicate.

varNumbers(TermIn, TermOut)

(Non-logical).

TermOut is *TermIn* with all terms of the form **`$VAR(Num)`**, where *Num* is an integer, replaced by variables. Two **`$VAR(Num)`** terms with the same *Num* will be bound to the same variable. See **`numberVars/3`**.

5.10. Predicates for Accessing the UNIX Operating System

access(Path, Mode)

(Non-logical). UNIX only.
Checks the given *Path* (an atom or string) for accessibility according to *Mode*, an inclusive binary or of the numbers 2'100 (test for read permission), 2'010 (test for write permission), 2'001 (test for execute permission). Fails if access is denied.

cd(Dir)
chdir(Dir)

(Non-logical). UNIX only.
Make *Dir* (an atom or string) the current working directory. Fails if the user does not have execute permission on *Dir*.

chmod(Path, Mode)

(Non-logical). UNIX only.
Change access permissions on *Path* (an atom or string) to be *Mode*. *Mode* is constructed by or'ing together some combination of the following:

8'4000	set user ID on execution
8'2000	set group ID on execution
8'1000	save text image after execution
8'0400	read by owner
8'0200	write by owner
8'0100	execute (search on directory) by owner
8'0070	read, write, execute (search) by group
8'0007	read, write, execute (search) by others

Fails if the user is not the owner of *Path* (and is not root).

csh

(Non-logical). UNIX only.
Invokes the UNIX shell *csh*. The NU-Prolog process is suspended until the shell process terminates. See **`sh`/`0`**.

directory(Dir, Files)

(Non-logical). UNIX only.
Files is a list of the filenames in directory *Dir* (an atom or string). Fails if the user can not read *Dir*.

environ(Environment)

UNIX only.
Get the user's UNIX *Environment*, a list of pairs of the form **`Name=Value`** where *Name* is an atom and *Value* is a string.

exec(Program, Arguments)

(Non-logical). UNIX only. Transfer control from NU-Prolog to another program. *Program* is the name of a file to be executed and *Arguments* is a list of arguments, including argument number zero, to be handed to *Program*. **`Exec`/`2`** fails if it is unable to run *Program*. If *Program* is not an absolute pathname, the environment variable **`PATH`** is searched for a directory containing *Program*.

exit(Code)

(Non-logical).

The NU-Prolog process terminates with *Code*, an integer. *Code* is made available to the parent process.

fork

fork(Pid)

(Non-logical). UNIX only.

Creates another NU-Prolog process, with the same core image. The only difference is that the call to **`fork`/`0`** in the parent process succeeds but the call in the child process fails. *Pid* is unified to the process id of the child; if *Pid* will not unify with this id, **`fork`/`1`** will fail in the parent, but the child will still be created. Care must be taken to ensure the two processes do not compete for **`user_input`** or other open files.

fork(Pid, Read, Write)

(Non-logical). UNIX only.

Like **`fork`/`1`**, but in the parent *Read* and *Write* are streams connected to the standard output and standard input of the child respectively.

getenv(Name, Value)

UNIX only.

Get the *Value* (a string) of environment variable *Name* (an atom). Fails if *Name* is not an environment variable. If *Name* is a variable, alternative <*Name*, *Value*> pairs are retrieved on backtracking.

getegid(Id)

UNIX only.

Id is the effective group ID of the current process.

getgid(Id)

UNIX only.

Id is the real group ID of the current process.

getgroups(Groups)

UNIX only.

Groups is a list of integers representing the current group access list of the user process.

getlogin(Login)

UNIX only.
Login is the users login ID.

getpid(Pid)

UNIX only.
Pid is the process ID of the current process.

getppid(Pid)

UNIX only.
Pid is the process ID of the parent of the current process.

getpw(Name, PwEnt)

UNIX only.
Name is a user name (an atom or string) or a user ID (an integer) for which *PwEnt* is the password entry. *PwEnt* is a list of elements of the form **`Name=Value`**, where *Name* is an atom and *Value* is a string. Values of *Name* include **`name`**, **`passwd`**, **`uid`**, **`gid`**, **`gecos`**, **`dir`** and **`shell`**. Fails if *Name* is not a valid user name or user ID.

getuid(Uid)

UNIX only.
Uid is the users real user ID.

geteuid(Uid)

UNIX only.
Uid is the users effective user ID.

getwd(Dir)

(Non-logical). UNIX only.
Dir is the current working directory pathname.

hostname(Host)

Host is the name of the machine on which NU-Prolog is running.

kill(Pid, Signal)

(Non-logical). UNIX only.
Send *Signal* to the process with *Pid*. *Signal* may be either a signal name (an atom) or a signal number (an integer). Fails if *Signal* is not valid, or was not sent successfully. Valid signals are described in Appendix 9.

link($Name_1$, $Name_2$)

(Non-logical). UNIX only.
A hard link to $Name_1$ is created with $Name_2$. Fails if either argument is not an atom or string, or if the user does not have appropriate permissions.

mkdir(Dir)

(Non-logical). UNIX only.
Make a new directory *Dir* (an atom or string). Fails if the user does not have the appropriate permissions.

rename(Old, New)

(Non-logical). UNIX only.
The link named *Old* takes the name *New*. Both must be atoms or strings. Under UNIX system V, both must be files; under Berkeley UNIX, they may also be directories. Fails if the user does not have appropriate permissions.

rmdir(Dir)

(Non-logical). UNIX only.
Remove the directory *Dir* (an atom or string). Fails if the user does not have appropriate permissions, or if the directory is not empty.

sh

(Non-logical). UNIX only.
Invokes the UNIX shell *sh*. The NU-Prolog process is suspended until the shell process terminates. See **`csh/0`**.

signal(Signal, Action)

(Non-logical) UNIX only.
Set *Action* for *Signal*. *Signal* may be either a signal name (an atom) or a signal number (an integer); these are listed in Appendix 9. *Action* is either the functor of a predicate with arity of 3 which will be called on *Signal*, or one of the atoms **`ignore`** or **`default`**. Three possible default *Action*s are provided: **`$break`**, which calls **`break/0`**. **`$exit`**, to exit on *Signal*; and **`$ignore`**, to ignore *Signal*. The default for a keybourd interrupt is **`$break`**. **`ignore`** and **`default`** provide the UNIX behaviors of SIG_IGN and SIG_DFL respectively.

If the user defines a predicate of arity 3 with functor *Action*, the arguments given to that predicate on *Signal* will be the break level, the signal number, and the number of signals pending, respectively. If *Signal* is not supported under the user's version of UNIX, the effect of using **`signal/2`** is undefined.

sleep(Seconds)

UNIX only.
Sleep for *Seconds* (an integer). Fails if *Seconds* is not ground.

stat(Path, Status)

(Non-logical). UNIX only.

Status is information about a file or directory *Path* (an atom or string). Read, write and execute permission on *Path* are not required, but all directories in the pathname must be reachable or the call will fail. *Status* is a list of elements of the form **`Name=Value`** where *Name* is an atom and *Value* is an integer. Values of *Name* include

ino	this inode’s number
mode	protection
nlink	number of hard links
uid	uid of owner
dev	device type
gid	group id of owner
size	total size of file
atime	last access time of file
mtime	last modify time of file
ctime	last status change of file
blocks	number of blocks allocated (BSD UNIX only)

system(Command)

(Non-logical). UNIX only.

Calls the UNIX shell *sh* with *Command*, a string or atom, as a command line. Fails if *Command* is of an inappropriate type.

system(Command, Status)

(Non-logical) UNIX only.

Command, a string or atom, is passed to UNIX for execution, and *Status* is bound to the exit status returned. Fails if *Command* is of an inappropriate type or if *Status* is already bound to something that does not unify with the exit status returned.

system(Program, Arguments, Status)

(Non-logical). UNIX only. *Program* is the name of an file to be executed and *Arguments* is a list of arguments, including argument number zero, to be handed to *Program*. If *Program* is not an absolute pathname, the enviroment variable **`PATH`** is searched for a directory containing *Program*. The *Program* and *Arguments* are passed to UNIX for execution (cf. **`exec/2`**) and *Status* is bound to the exit status returned. Fails if either *Program* or *Arguments* are of an inappropriate type or if *Status* is already bound to something that does not unify with the exit status returned.

time(Time)

(Non-logical). UNIX only.

Time is the decoded current time. It is a list of elements of the form **`Name=Value`**, where *Name* is an atom and *Value* is an integer. Values of *Name* are **`year`**, **`month`** (*Value* is 1..12), **`date`** (Value is 1..31), **`day`** (values are the atoms **`Mon`**, ... , **`Sun`**), **`hour`**, **`minute`** and **`second`**.

time(When, Time)

(Non-logical). UNIX only.

Time is the decoded version of the timestamp When. It is a list of elements of the form **`Name=Value`**, where *Name* is an atom and *Value* is an integer. Values of *Name* are **`year`**, **`month`** (*Value* is 1..12), **`date`** (Value is 1..31), **`day`** (values are the atoms **`Mon`**, ... , **`Sun`**), **`hour`**, **`minute`** and **`second`**.

truncate(File, Length)

(Non-logical). UNIX only.

File is truncated to be *Length* characters, or left alone if already shorter. Fails if the user does not have appropriate permissions.

umask(Umask)

(Non-logical). UNIX only.

Set default file creation mask to *Umask*. The low-order nine bits of the umask are used whenever a file is created to clear corresponding bits in the file's access mode. The default umask is 8'022. See **`chmod`**/**`2`**.

unlink(Path)

(Non-logical). UNIX only.

Remove the entry for *Path* (an atom or string) from its directory. Fails if the user does not have the appropriate permissions.

wait(Pid, Status)

(Non-logical). UNIX only.

User process waits until one of its child processes terminates or a signal is received. *Pid* is bound to the process id of the terminating process, *Status* to its exit status.

5.11. Foreign Function Interface

On machines in which it is implemented, the foreign function interface allows NU-Prolog to load and execute functions written in other languages. Currently only C and languages with similar calling conventions are supported, but adding calling interfaces for languages such as Pascal and FORTRAN should not be difficult. The interface is available only on machines running a Berkeley flavour of UNIX, and not always then. The primary requirement is support for the –A incremental loading option to /bin/ld. At the time of writing it is known to run on Sun-3s, Sun-4s, Encores and VAXes.

The files containing foreign functions and the individual functions are described by giving clauses for the predicates **`foreignFile`**/**`2`** and **`foreign`**/**`[2,3]`** respectively. These clauses are loaded into a running NU-Prolog and **`loadForeignFiles`**/**`2`** called with a list of files and a list of libraries to load. After loading, the NU-Prolog predicates specified by the **`foreign`**/**`[2,3]`** clauses are connected to the foreign functions named in the **`foreignFile`**/**`2`** clauses. The descriptive predicates may then be abolished if desired.

Parameters are passed to and from NU-Prolog according to the specifications given in **`foreign`**/**`[2,3]`**. Each parameter or return value of the foreign function is specified as one of +**`(X)`**, **`–(X)`**, or **`[–(X)]`**, where **`X`** is one of **`integer`**, **`float`**, **`single`**, **`double`**, **`atom`**, **`term`**, **`pointer`**, or **`string`**. **`Integer`**, **`float`**, **`atom`**, and **`term`** indicate that a NU-Prolog object of that type will be passed one way or the other across the interface. On the Prolog side of the interface, **`single`** and **`double`** are synonyms for **`float`**, but they may be different types on the foreign function side. See the table below. Both **`atom`** and **`string`** represent atoms in NU-Prolog, but are represented differently in the foreign language. A **`pointer`** is a NU-Prolog representation of a machine memory address. The only pointer objects that can be seen by a user of NU-Prolog are those returned over the foreign function interface, and their only use is to be given to other foreign functions. NU-Prolog checks that arguments are of their specified type and causes the foreign call to fail silently if any are not.

The +**`(X)`** arguments are values to be supplied by NU-Prolog to the foreign function and must be bound when the function is called. The **`–(X)`** arguments are places for values to be returned by the function and are unified with whatever the function stores to that parameter. Optionally, there may be a single argument of the form **`[–(X)]`** which is unified with the return value of the function.

The NU-Prolog types that can be passed and their corresponding C types are given in the table below. The C types **`Word`** and **`Object`** are defined in public.h which is part of the NU-Prolog library and should be included in any C programs that use the interface. **`Integer`**, **`float`**, **`single`**, and **`double`** are passed in the obvious fashion, with Prolog integers being converted to floating point if passed to a +**`float`**, +**`single`**, or +**`double`** argument; **`atom`** is passed as an integer index into an internal NU-Prolog table of atoms; **`string`** as the print-name of the atom passed; **`pointer`** as the address that was used to make it; and **`term`** as itself. Functions to manipulate **`atom`**, **`pointer`** and **`term`** arguments are provided by public.h.

A common source of confusion is the **`string`** parameter type. It is used to pass the print-name of a Prolog atom as a C (char *), and return a C (char *) as a Prolog atom. It cannot be used to pass a Prolog list of character codes.

Arguments of the form **`–(X)`** are passed as the address of a suitably sized area of memory for the foreign function to store to. The results of a call to a foreign function are undefined if no value is stored in the addresses passed for **`–(X)`** arguments, or if the function has a **`[–(X)]`** argument but does not return a value.

If a function has no **`[–(X)]`** argument its return value will be ignored.

Prolog	C
+integer	Word
+float	double
+single	float
+double	double
+atom	Word
+term	Object
+string	char *
+pointer	char *
-integer	Word *
-float	double *
-single	float *
-double	double *
-atom	Word *
-term	Object *
-string	char **
-pointer	char **
[-integer]	return Word
[-float]	return double
[-single]	return float
[-double]	return double
[-atom]	return Word
[-term]	return Object
[-string]	return char *
[-pointer]	return char *

foreign(Function, Language, Specification)
foreign(Function, Specification)

`Foreign`/**`3`**, and the shorthand **`foreign`**/**`2`** which is equivalent to **`foreign`**/**`3`** with *Language* **`c`**, are used to declare the way in which NU-Prolog calls functions loaded with the foreign function interface. Note that **`foreign`**/**`[2,3]`** merely define a table. The predicate is never called.

Function is the name of the function as it is given in *Language*, *Language* is the language in which the function is written, and *Specification* is a term **`F(`**$\mathtt{Arg_1}$**`,`** $\mathtt{Arg_2}$**`, ...,`** $\mathtt{Arg_N}$**`)`** where the NU-Prolog predicate **`F`**/**`N`** is to be connected to *Function* according to the $\mathtt{Arg_i}$. The $\mathtt{Arg_i}$ of the form +**`(X)`** or **`-(X)`** are passed to the foreign function in the order in which they appear. The optional argument of the form **`[-(X)]`** is unified with the function's return value.

Currently, the only language supported is **`c`**.

foreignFile(FileName, Functions)

`ForeignFile`/2 is used to declare the functions that will be loaded from a file of foreign functions by **`loadForeignFiles`/2**. *FileName* is the name of a file and *Functions* is a list of names of functions defined in that file that are to be connected to NU-Prolog predicates. Functions that are not to be called directly from NU-Prolog should not be listed. Like **`foreign`/`[2,3]`**, **`foreignFile`/2** merely defines a table. It is never called.

loadForeignFiles(Files, Libraries)

Load a list of *Files* containing functions written in another programming language, resolving undefined symbols in the list of *Libraries*. The functions to be loaded are determined by looking in **`foreignFile`/2** and their language and interface in **`foreign`/`[2,3]`**.

Any libraries appropriate to the language of the foreign functions are loaded automatically. Others should be given their full names such as ‘‘/usr/lib/libm.a’’.

Loading foreign functions is much more complicated than loading a .no file and may take some considerable time.

CHAPTER 6

LIBRARY PREDICATES

This chapter describes predicates available in the NU-Prolog library. A library provides each of the predicates listed under *lib_name* and is loaded with the predicate **`lib lib_name`**. Normally, this predicate would be used as a goal such as

```
?- lib lib_name.
```

Some of these descriptions include a **`when`** declaration for that predicate. However, because most predicates are defined recursively, the **`when`** declaration is only a partial indication of the instantiation required before the predicate can be completely executed.

6.1. Compatibility

To load the predicates in the compatibility library use the subgoal **`lib compat`**. The predicates in this library provide compatibility with some other Prolog systems. There are, however, usually more predicates in the library than are documented here.

current_atom(Atom)
current_atom(Module, Atom)

(Non-logical).
Atom is an atom in the module **`user`** (or *Module*). Same as **`currentAtom/2`**.

current_input(Stream)

(Non-logical).
Stream is unified with current input stream. Fails if *Stream* has previously been bound to something that is not a currently open input stream. Same as **`currentInput/1`**.

current_op(Precedence, Type, Op)

(Non-logical).
Op is an operator of specified *Type* and *Precedence*. Fails if any of the arguments is of an inappropriate type. Same as **`currentOp/3`**.

current_output(Stream)

(Non-logical).
Stream is unified with the current output stream. Fails if *Stream* has previously been bound to something that is not a currently open output stream. Same as **`currentOutput/1`**.

current_predicate(Functor, Term)

(Non-logical).
Functor is the name of a user-defined predicate and *Term* is the most general term corresponding to that predicate.

current_stream(File, Mode, Stream)

(Non-logical).
Stream is a currently open stream on *File*, where *Mode* is either **`read`**, **`write`** or **`append`**. On backtracking, **`current_stream/3`** finds all suitable streams. Fails if *Stream* has previously been bound to something that is not a currently open stream of the appropriate type. Same as **`currentStream/3`**.

false

Same as **`fail/0`**.

fileerrors

(Non-logical).
Resets **`fileErrors`** flag to **`on`**. See **`prologFlag/3`**. Same as **`fileErrors/0`**.

flush_output(Stream)

(Non-logical).
Flush the specified output *Stream*. Same as **`flushOutput/1`**.

gc

Enable garbage collection (actually do nothing).

gcguide(Parameter, Old, New)

Change garbage collection or memory management parameter for *Parameter* from *Old* value to *New* value (actually do nothing).

gnot(Vars, Goal) when ground(Vars)

(Non-logical).
Generalized negation of *Goal*. Delays until all of the variables in term *Vars* are ground, then negates *Goal*. Superseded by **`not some V Goal`**, where *V* is a term containing the variables in *Goal* that are not in *Vars*.

halt

(Non-logical).
Exit from NU-Prolog.

incore(Goal).

Same as **`call(Goal)`**.

nofileerrors

(Non-logical).
Sets **`fileErrors`** flag to **`off`**. See **`prologFlag/3`**. Same as noFileErrors.

nogc

Disable garbage collection (actually does nothing).

otherwise

Same as **`true/0`**. Useful for layout of programs which use **`->`**.

predicate_property(Skeleton, Property)

The predicate the functor and arity of which are those of the term *Skeleton* has *Property*. Same as **`functor(Skeleton, Functor, Arity), predicateProperty(Functor, Arity, Property)`**.

prolog_flag(Flag, Value$_1$, Value$_2$)

(Non-logical).
Same as **`PrologFlag/3`**.

putatom(Term)
putatom(Stream, Term)

(Non-logical).
Term is printed on the current output stream (or *Stream*).

set_input(Stream)

(Non-logical).
Set current input to *Stream*. Prints warning and fails if *Stream* is not a currently open input stream. Same as **`SetInput`**/**`1`**.

set_output(Stream)

(Non-logical).
Set current output to *Stream*. Prints warning and fails if *Stream* is not a currently open output stream. Same as **`setOutput`**/**`1`**.

subgoal_of(Goal)

(Non-logical)
Current goal is a subgoal of Goal. Same as **`ancestors(L), member(Goal, L)`**.

NOT YET IMPLEMENTED.

trimcore

Release free space from data areas. Automatically called from command level of interpreter.

6.2. Ordered Sets

To load the predicates in the ordered sets library use the subgoal **`lib osets`**. This module provides predicates which manipulate sets as represented by ordered lists with no duplicates. The ordering is defined by **`termCompare/3`**. The benefit of the ordered representation is that the elementary set operations can be done in time proportional to the sum of the argument sizes, rather than the product. The behaviour of these predicates is not well defined if an unsorted list, or a list with duplicates, is given as argument where a set is expected.

This library was adapted from code written by Richard O'Keefe. All of the predicates in this library fail on backtracking.

addElement(Elem, Set_1, Set_2) when Elem and Set_1

True when Set_1 and Set_2 are sets represented as ordered lists and $\mathtt{Set_2 = Set_1} \cup \mathtt{\{Elem\}}$. Delays until *Elem* and Set_1 are instantiated.

delElement(Elem, Set_1, Set_2) when Elem and Set_1

True when Set_1 and Set_2 are sets represented as ordered lists and $\mathtt{Set_2 = Set_1 \setminus \{Elem\}}$. Delays until *Elem* and Set_1 are instantiated.

disjoint(Set_1, Set_2) when Set_1 and Set_2

True when the two ordered sets have no element in common. Delays until they are instantiated.

intersect(Set_1, Set_2) when Set_1 and Set_2

True when the two ordered sets have at least one element in common. Delays until they are instantiated.

intersect(Set_1, Set_2, Intersection) when Set_1 and Set_2

True when *Intersection* is the ordered representation of $\mathtt{Set_1} \cap \mathtt{Set_2}$, provided that both sets are ordered. Delays until Set_1 and Set_2 are instantiated.

listToSet(List, Set) when List

True when *Set* is a list consisting of the ordered, sorted elements of *List* with duplicates removed. Delays until *List* is instantiated.

setMember(Element, Set) when Element and Set

True when *Element* is a member of *Set*. Takes advantage of *Set* being sorted, examining only the part of *Set* that is smaller than *Element*. Delays until both are instantiated.

setEq(Set_1, Set_2)

True when the two arguments represent the same set.

subset(Set_1, Set_2) when Set_1 and Set_2

True when every element of ordered Set_1 appears in ordered Set_2.

subtract(Set_1, Set_2, Difference) when Set_1 and Set_2

True when *Difference* contains all and only the elements of ordered Set_1 which are not in ordered Set_2. Delays until Set_1 and Set_2 are instantiated.

symdiff(Set_1, Set_2, Diff) when Set_1 and Set_2

True when *Diff* is the symmetric difference of Set_1 and Set_2, delays until they are instantiated.

union(Set_1, Set_2, Union) when Set_1 and Set_2

True when *Union* is the union of ordered sets Set_1 and Set_2. When something occurs in both sets, only one copy is kept. Delays until Set_1 and Set_2 are instantiated.

6.3. Debugging

To load the predicates in the debugging library use the subgoal **`lib debug`**. The predicates in this library provide a common set of predicates useful for debuggers of various types.

6.3.1. Initializing the debugging environment

dConsult(File)
dConsult([File1, File2, ...])

(Non-logical).
Like **`consult`**/**`1`** but clauses are saved in the debugging area and do not effect other code.

dLoad(File)
dLoad([File1, File2, ...])

(Non-logical).
Same as **`dConsult`**/**`1`**, but if an up to date *File.no* exists, it is also loaded.

processGoal(File, Goal, NameList, VarList)

(Non-logical).
This user defined predicate will be used for goal processing if defined when a file is being dConsulted. The '?-' is stripped off before **`processGoal`** is called. If this call fails, the goal will be stored as usual. n.b. when, dynamic, pure and operator declarations are separately handled.

6.3.2. Accessing the debugging environment

dPred(F, A)

(Non-logical).
A procedure with functor F and arity A exists in the debugging area.

dPreds(FNList)

(Non-logical).
FNList is the list of procedures in the debugging area, in the form $[F1/A1,F2/A2,...]$.

dClause(F, A, Clause)

(Non-logical).
Clause is a clause of procedure F/A in th debugging area. All clauses are represented by terms of the form **`cl((Head:-Body),NameList,VarList)`** where *NameList* and *VarList* are as in **`readTerm`**/**`3`**. F and/or A may be variables.

dClauses(F, A, ClauseList)

(Non-logical).
ClauseList is the list of clauses (in cl/3 format) of the predicate F/A. F and/or A may be variables.

dFile(F, A, File)
(Non-logical).
Predicate *F*/*A* has been loaded from *File* (where *File* is not '@').

dGoal(File, Goal)
(Non-logical).
Goal is a goal in the debugging area, read from *File*.

dGoals(GoalList)
dGoals(File, GoalList)
(Non-logical).
GoalList is the list of goals in the debugging area from *File*, or all goals.

dDec(F, A, DType, Dec)
(Non-logical).
Dec is a declaration of type *DType* for procedure *F*/*A*.

dDecs(F, A, DType, DecList)
(Non-logical).
DecList is the list of declarations of type *DType* for procedure *F*/*A*.

dProp(F, A, PType, Prop)
(Non-logical).
Prop is a property of type *PType* for procedure *F*/*A*.

dProps(F, A, PType, PropList)
(Non-logical).
PropList is the list of properties of type *PType* for procedure *F*/*A*.

dDCall(F, A, CF, CA)
(Non-logical).
Predicate *F*/*A* has a call to *CF*/*CA* in the body of one of its clauses.

dDCalls(F, A, FNList)
(Non-logical).
FNList is the list of procedures directly called by procedure *F*/*A*.

dDAncs(F, A, Preds)
(Non-logical).
Preds are the predicates which call *F*/*A* directly.

dICall(F, A, CF, CA)

(Non-logical).

Predicate *F/A* calls *CF/CA*, possibly indirectly via other procedures. *F* and *A* must be instantiated.

dICalls(F, A, FNList)

(Non-logical).

FNList is the list of procedures called (possibly indirectly) by procedure *F/A*.

dGoalDCall(G, F, A)

(Non-logical)

F/A is called directly by goal.

dGoalICall(G, F, A)

(Non-logical)

F/A is called (possibly indirectly) by goal.

6.3.3. Manipulting the debugging environment

dAddPred(F, A)
dAddPred(F, A, File)

(Non-logical).

Assert the existence of *F/A* in the debugging environment. **`dAddPred`/3** asserts the existence of *F/A* and associates it with *File*. Fails if *F/A* is associated with another file.

dAddPreds(Predicates)
dAddPreds(Predicates, File)

(Non-logical).

Assert the existence of all the predicates [*F*1/*A*1,*F*2/*A*2,...] in *Predicates* in the debugging environment. **`dAddPreds`/3** asserts the existence of the predicates and associates them with *File*, or fails if any is already associated with a different file.

dAbolishPred(F, A)

(Non-logical).

Remove *F/A* and all associated information from the debugging environment.

dAbolishPreds(Preds)

(Non-logical).

Remove the predicates [*F*1/*A*1,*F*2/*A*2,...] in *Preds* and all associated information from the debugging environment.

dAddClause(Clause)

(Non-logical).

Adds Clause (in cl/3 format) to the end of a procedure.

dAddClauses(List)

(Non-logical).

Add a list of clauses (in cl/3 format) to the end of a procedure.

dRmClause(F, A, Clause)

(Non-logical).

Removes a clause from the procedure F/A.

dRmClauses(F, A)

(Non-logical).

Removes all clauses from the procedure F/A.

dAddGoal(File, Goal)

(Non-logical).

Adds *Goal* to the debugging area. Goal is considered to have come from *File*. For example **`dAddGoal(user,goal(assert(count(0)),[],[]))`**.

dAddGoals(File, GoalList)

(Non-logical).

Adds a list of goals to the debugging area, considered to have come from *File*.

dRmGoal(File, Goal)

(Non-logical).

Removes *Goal* read from *File* from the debugging area.

dRmGoals
dRmGoals(File)

(Non-logical).

Removes all goals from *File* from the debugging area, or all goals.

dAddDec(F, A, DType, Dec)

(Non-logical).

Adds a declaration *Dec* of type *DType* to procedure F/A. For example **`dAddDec(len,2,type,len(list,int))`**.

dAddDecs(F, A, DType, DecList)

(Non-logical).

Adds a list of declarations *DecList* of type *DType* to procedure F/A.

dRmDec(F, A, DTyp, Dec)

(Non-logical).

Removes declaration *Dec* of type *DType* from procedure F/A.

dRmDecs(F, A)
dRmDecs(F, A, DType)

(Non-logical).
Removes all declarations of type *DType* from procedure F/A, or all declarations.

dAddProp(F, A, PType, Prop)

(Non-logical).
Adds a property *Prop* of type *PType* to procedure F/A. For example **`dAddProp(perm,2,direct_calls,[append/3,delete/3])`**.

dAddProps(F, A, PType, PropList)

(Non-logical).
Adds a list of properties *PropList* of type *PType* to procedure F/A.

dPutProp(F, A, PType, Prop)

(Non-logical).
Puts a property *Prop* of type *PType* with procedure F/A. For example **`dPutProp(perm,2,direct_calls,[append/3,delete/3])`**.

dRmProp(F, A, PType, Prop)

(Non-logical).
Removes *Prop* of type *PType* from procedure F/A.

dRmProps(F, A)
dRmProps(F, A, PType)

(Non-logical).
Removes all properties of type *PType* from procedure F/A, or all properties.

6.3.4. Miscellaneous

dEdit(Pred, Arity)

(Non-logical).
Revise the file containing the definition of **`Pred/Arity`**, positioning the cursor appropriately.

dListing

(Non-logical).
List the predicates currently in the debugger. This is analogous to **`listing/0`**.

dPortraycl(Clause)

(Non-logical).
Clause is a clause in cl/3 format and is output with its original variable names via **`portraycl`**.

6.4. Curses

This library provides access to the Unix curses library. (Screen updating with ‘‘optimal’’ cursor movement).

This library depends on the availability of the foreign function loading facility of NU-Prolog. It also depends on the local curses library. Users are advised to look at relevant curses doccumentation.

Use is as similar to curses use from C as is practicable. Functions with return values have an extra argument, and functions which return error conditions in C will fail. "mv" variants of output predictes are provided where they exist in the C curses package. Predicates which do not have a window argument operate on the standard screen.

`printw` and **`scanw`** and are not provided, instead window forms of **`write`** **`writev`** and **`format`** are available. **`Addch`**, **`addstr`**, **`getch`**, and **`getstr`** remain. **`_putchar`** and **`getcap`** are not provided.

6.4.1. Access to Global Variables

Variables provided in curses to describe the terminal environment are accessed through the following predicates.

curscr(Win)
: (Non-logical).
Win is a window descriptor for **`curscr`**.

stdscr(Win)
: (Non-logical).
Win is a window descriptor for **`stdscr`**.

def_term(Type)
: (Non-logical).
Type is an atom bound to the default terminal type.

my_term(Flag)
: (Non-logical).
Returns *Flag*, an integer true(1) or false(0), if *Flag* is a variable or sets *Flag* if *Flag* is ground. If true, use the terminal type in **`def_type`** regardless of terminal type.

lines(LINES)
: (Non-logical)
Binds *LINES* to the number of lines on the terminal if *LINES* is a variable. Sets the number of lines to be *LINES* if *LINES* is bound. Fails if *LINES* is not an integer.

cols(COLS)
: (Non-logical)
Binds *COLS* to the number of lines on the terminal if *COLS* is a variable. Sets the number of lines to be *COLS* if *COLS* is bound. Fails if *COLS* is not an integer.

err(ERR)

ERR is an integer returned by curses when things fail. In NU-Prolog, such calls fail.

ok(OK)

OK is the curses constant **`OK`** which is returned by some functions which may fail.

6.4.2. Output Predictes

addch(Ch)

(Non-logical).
Add *Ch* to *stdscr* at current (y,x) co-ordinates

waddch(Win, Ch)

(Non-logical).
Ch is the ASCII code of a character to be printed at the current cursor position. *Win* is a window descriptor, returned by **`newwin`**, **`subwin`**, **`stdscr`** et al. The current (y,x) co-ordinates are updated. Fails if it would cause the screen to scroll illegally.

addstr(Str)

(Non-logical)
Same as **`stdscr(Win),waddstr(Win,Str)`**.

waddstr(Win, Str)

(Non-logical).
Writes a string on the window at the current cursor positions. Fails if this would cause illegal scrolling. (Will print as much as it can.)

box(Win, Vert, Hor)

(Non-logical).
Draws a box around the window using the characters corresponding to the ASCII codes of *Vert* and *Hor* for drawing the vertical and horizontal lines respectively.

clear

(Non-logical)
Clears *stdscr* to blanks. Will cause the screen to be cleared on the next refresh. Moves the current (y,x) co-ordinates to (0, 0).

wclear(Win)

(Non-logical).
Clears the entire window to blanks. Will cause screen to be cleared on the next refresh. Moves the current (y,x) co-ordinates to (0, 0).

clearok(Scr, Boolf)

(Non-logical).

Boolf is true (1) or false (0). Sets the clear flag for the screen *Scr*. Causes a clear-screen on the next refresh if true or prevents one if false. Unlike clear does not alter the contents of the screen.

clrtobot

(Non-logical).

Clears *stdscr* to the bottom. Has no ‘‘**mv**’’ variant.

wclrtobot(Win)

(Non-logical).

Clears to the bottom of the window. Has no ‘‘**mv**’’ variant.

clrtoeol

(Non-logical).

Clears to the end of the current line on *stdscr*. Has no ‘‘**mv**’’ variant.

wclrtoeol(Win)

(Non-logical).

Clears to the end of the line on *Win*. Has no ‘‘**mv**’’ variant.

delch

(Non-logical).

Delete the character at the current (y,x) co-ordinates on *stdscr*, moving all remaining characters to the left. The rightmost character becomes a blank.

wdelch(Win)

(Non-logical).

Delete the character at the current (y,x) co-ordinates, moving all remaining characters to the left. The rightmost character becomes a blank.

deleteln

(Non-logical).

Delete the current line on *stdscr*. Remaining lines move up and the bottom line becomes blank.

wdeleteln(Win)

(Non-logical).

Delete the current line. Remaining lines move up and the bottom line becomes blank.

erase

(Non-logical).

Erase *stdscr* without setting clear flag. Has no ‘‘**mv**’’ variant.

werase(Win)

(Non-logical).
Erase the window without setting clear flag. Has no ‘‘**mv**’’ variant.

flushok(Win, Boolf)

(Non-logical).
Boolf is true(1) or false(0). Normally, refresh flushes stdout when it is finished. **flushok** controls this.

formatw(Format, Args)
formatw(Win, Format, Args)

(Non-logical).
Replaces **printw**. Same as **format/2** and **format/3**, with the window *Win* (or *stdscr*) replacing a stream.

insch(C)

(Non-logical).
C is the ASCII code of a character to be inserted at the current (y,x) co-ordinates on *stdscr*. Characters after it shift to the right, the last character disappears. Fails if it would cause the screen to scroll illegally.

winsch(Win, C)

(Non-logical).
C is the ASCII code of a character to be inserted at the current (y,x) co-ordinates. Characters after it shift to the right, the last character disappears. Fails if it would cause the screen to scroll illegally.

insertln

(Non-logical).
Insert a line above the current one on *stdscr*. Every line below is shifted down and the bottom line disappears. The current line becomes blank and the current (y,x) co-ordinates are not altered. Has no ‘‘**mv**’’ variant.

winsertln(Win)

(Non-logical).
Insert a line above the current one. Every line below is shifted down and the bottom line disappears. The current line becomes blank and the current (y,x) co-ordinates are not altered. Has no ‘‘**mv**’’ variant.

move(Y, X)

(Non-logical).
Change the current (y,x) co-ordinates of *stdscr* to (Y,X). Fails if it would cause the screen to scroll illegally.

wmove(Win, Y, X)

(Non-logical).

Change the current (y, x) co-ordinates to (Y, X). Fails if it would cause the screen to scroll illegally.

nlw
nlw(Win)

(Non-logical).

Outputs a newline on the window.

overlay(Win1, Win2)

(Non-logical).

Overlay window 1 on window 2. This is non-destructive: blanks on window 1 are untouched on window two.

overwrite(Win1, Win2)

(Non-logical).

Overwrite window 1 on wondow 2. Blanks on window 1 become blanks on window two.

refresh

(Non-logical).

Synchronize the terminal screen with the desired window. Fails if this would cause the screen to scroll illegally. In this case, a partial update will be done.

wrefresh(Win)

(Non-logical).

Synchronize the terminal screen with the desired window. Fails if this would cause the screen to scroll illegally. In this case, a partial update will be done.

As a special case, if **`wrefresh()`** is called with *curscr* the screen is cleared and redrawn as it is currently. Useful for redrawing corrupted screens.

standout

(Non-logical).

Put *stdscr* in standout mode.

standend

(Non-logical).

End standout mode on *stdscr*.

wstandout(Win)

(Non-logical).

Put *Win* in standout mode.

wstandend(Win)

(Non-logical).
End standout mode for *Win*.

writew(Term)
writew(Win, Term)

(Non-logical).
Same as **`write`/2**, **`write`/3**, but with *Win* a window not a stream.

writelnw(Term)
writelnw(Win, Term)

(Non-logical).
Same as **`writeln`/2**, **`writeln`/3**, but with *Win* a window not a stream.

writevw(Flags, Term)
writevw(Win, Flags, Term)

(Non-logical).
Same as **`writev`/2**, **`writev`/3**, but with *Win* a window not a stream.

6.4.3. Input Predicates

cbreak

(Non-logical).
Put the terminal in cbreak mode.

nocbreak

(Non-logical).
End cbreak mode.

crmode

(Non-logical).
(Obsolete) Equivalent to **`cbreak`**.

nocrmode

(Non-logical).
(Obsolete) Equivalent to **`nocbreak`**.

echo

(Non-logical).
Turn terminal echoing on.

noecho

(Non-logical).
Sets the termianal not to echo characters.

getch(Ch)

(Non-logical).
Gets a character from the terminal and (if necessary) echos it on *stdscr*. Fails if it would cause the screen to scroll illegally.

wgetch(Win, Ch)

(Non-logical).
Gets a character from the terminal and (if necessary) echos it on the window. Fails if it would cause the screen to scroll illegally.

getstr(Win, Str)

(Non-logical).
Gets a string from the terminal and returns it as the atom *Str*. Fails if it would cause the screen to scroll illegally.

wgetstr(Win, Str)

(Non-logical).
Gets a string from the terminal and returns it as the atom *Str*. Repeated calls are made to **`getstr`** hence echoing etc. are the same. Fails if it would cause the screen to scroll illegally.

raw

(Non-logical).
Set the terminal to raw mode.

noraw

(Non-logical).
Unset the terminal from raw mode.

6.4.4. Miscellaneous Predicates

baudrate(Baudrate)

Returns the output baud rate of the terminal.

delwin(Win)

(Non-logical).
Relieves the window of the burden of existence. Does not attend to subwindows, although these become invalid. Do your own housekeeping!

endwin

(Non-logical).
Finish up window routines before exit. Restores the terminal to the state previous to that when **`initscr`** (or **`gettmode`** and **`setterm`**) were called. Should always be called before exiting.

erasechar(Ch)

Ch is the ASCII code of the terminal erase character.

getyx(Win, Y, X)

(Non-logical).
Returns the current (Y,X) co-ordinates for *Win*.

inch(Ch)

(Non-logical).
Ch is the ASCII code of the character at the current (y,x) co-ordinates of *stdscr*.

winch(Win, Ch)

(Non-logical).
Ch is the ASCII code of the character at the current (y,x) co-ordinates.

initscr

(Non-logical).
Initialize the screen routines. This must be called before any screen routines. Fails if enough memory cannot be allocated.

killchar(Ch)

Ch is the ASCII code of the line kill character for the terminal.

leaveok(Win, Boolf)

(Non-logical).
Sets the boolean flag for leaving the cursor after the last change. If true (*Boolf* = 1) the cursor will be left after the last update on the terminal, and the current (y, x) co-ordinates changed accordingly. If false (*Boolf* = 0), the cursor will be moved to the current (y, x) co-ordinates. Initially false.

longname(Name)

Name is an atom bound to the long name of the terminal described by the **`termcap`** entry.

fullname(Name)

Name is an atom bound to the longest name of the terminal described by the **`termcap`** entry.

mvwin(Win, Y, X)

(Non-logical).
Move the home position of *Win* to (Y,X). Fails if this would move some or all of the window off the screen. For subwindows, fails if an attempt is made to move off its main window. If a main window is moved, all its subwindows are also moved.

newwin(Lines, Cols, BeginY, BeginX)

(Non-logical)
Create a new window with *Lines* and *Cols*, starting at position (*BeginY*,*BeginX*). If either *Lines* of *Cols* is zero, that dimension wil be set to (*LINES* − *BeginY*) or (*COLS* − *BeginX*) respectively. Thus to get a new window of demensions *LINES x COLS*, use **`newwin(0,0,0,0)`**.

nlon

(Non-logical).
Turn newline/carriage return mapping on. Equivalent to curses **`nl`**.

nonl

(Non-logical).
Unset the terminal from newline/carriage return mapping.

scrollok(Win, Boolf)

(Non-logical).
Set the scroll flag for the given window. True = 1, False = 0. Initial value is scrolling not allowed (False).

touchline(Win, Y, StartX, EndX)

(Non-logical).
Similar to **`touchwin`** on a single line. Marks the first change for the given line to be *StartX*, if it is before the first change mark, and the last change mark to be *EndX* if it is after the last change mark.

touchoverlap(Win1, Win2)

(Non-logical).
Touch the window *Win*2 in the area which overlaps *Win*1. If there is no overlap, no changes are made.

touchwin(Win)

(Non-logical).
Make it appear that evey location on the window has changed. Usually only needed with overlapping windows.

subwin(Win, Lines, Cols, BeginX, BeginY, NewWin)

(Non-logical).
Similar to **`newwin`**. Any change made to either window in the area covered by the subwindow will be made on both windows. *BeginY* and *BeginX* are relative to the overall screen, not *Win*.

unctrl(Ch, Str)

Str is an atom bound to a string representation of the ASCII code *Ch*. Control characters become their upper case equivalents preceeded by ‘‘^’’. Other letters stay as they are.

6.4.5. Details

gettmode

(Non-logical).
Get the ttystats. Normally done by **`initscr`**.

mvcur(LastY, LastX, NewY, NewX)

(Non-logical).
Moves the terminal's cursor from (*LastY*,*LastX*) to (*NewY*,*NewX*) in approximately optimal fashion. When using the screen routines, this should not be called by the user, **`move`** and **`refresh`** should be used instead, so the routines know what's going on.

scroll(Win)

(Non-logical).
Scroll the window up one line. Not normally used by the user.

savetty

(Non-logical).
Save the current tty characteristic flags. Performed automatically by **`initscr`**.

resetty

(Non-logical).
`Resetty` restores the tty characteristic flags to what **`savetty`** stored. Performed automatically by **`endwin`**.

setterm(Name)

(Non-logical).
Set the terminal characteristics to those of the terminal named *Name*. Normally called by **`initscr`**.

tstp

(Non-logical).
Save the current tty state and put the terminal to sleep. When the process gets restarted, it restores the tty state and calls **`wrefresh(curscr)`**. **`Initscr`** sets the signal SIGSTP to trap to this routine.

APPENDIX 1
Example NU-Prolog Programs

Declarative Constructs

Example 1

The following predicate is true when *Person* takes no computer science courses; it will delay until *Person* is ground.

```
no_cs(Person) :-
      all Unit not takes(Person, cs, Unit).
```

Example 2

Flatten lists of lists (of lists ...) into a single list.

```
?- flatten([], _) when ever.
?- flatten(A._, _) when A.
flatten([], []).
flatten([].A, B) :-
      flatten(A, B).
flatten((A.B).C, D) :-
      flatten(A.B.C, D).
flatten(A.B, A.C) :-
      A ~= [],
      A ~= _._,                 % Implicit quantifiers.
      flatten(B, C).
```

The expression **`A ~= _._`** is equivalent to **`all X.Y A ~= X.Y`**.

Example 3

A and *B* are connected in a graph. If *A* and *B* are ground, only one path is considered.

```
connected(A, B) :-
      some Path path(A, B, Path).
```

Example 4

A is a subset of *B*, where sets are represented as lists; this only works as a test under the current implementation of **`all/2`**.

```
?- pure subset/2.
subset(A, B) :-
      all E member(E, A) => member(E, B).
```

Example 5

Return value(s) *Val* of *Key* in an association list *AssocL*; if *Key* is not present, add it to the list, giving *NewAssocL*.

```
lookup(Key, Val, AssocL, NewAssocL) :-
     (if some V member(Key-V, AssocL) then
          Val = V,                % V is quantified
          NewAssocL = AssocL
     else
          NewAssocL = (Key-Val).AssocL
     ).
```

Example 6

termCompare/3 is a logical term comparison predicate using **compare/3**. **if** delays until *Term*1 and *Term*2 are identical or don't unify. If they don't unify, **compare/3** will act logically.

```
termCompare(C, Term1, Term2) :-
     if Term1 = Term2 then
          C = (=)
     else
          compare(C, Term1, Term2).
```

When Declarations

Example 7

```
?- append(A, B, C) when A or C.
append([], A, A).
append(A.B, C, A.D) :-
     append(B, C, D).
```

This specifies that **append/3** may be called when either the first or third arguments are nonvariables. Thus **?- append([X, 2], [3], Y).** and **?- append(X, Y, [1, 2, 3]).** would proceed, whereas **?- append(X, [3], Y).** would delay immediately and **?- append(1.X, [3], Y).** would delay at the first recursive call until *X* or *Y* are not variables.

Example 8

```
?- sorted([]) when ever.
?- sorted(A.B) when A and B.
sorted([]).
sorted([A]).
sorted(A.B.C) :-
     A =< B,
     sorted(B.C).
```

This specifies that **sorted/1** may be called when its argument is the empty list or a list constructor with both the head and the tail being nonvariables. Thus **?- sorted([]).** and **?- sorted(1.2.X).** would proceed, whereas **?- sorted(X).** and **?- sorted(1.X).** would delay.

Example 9

```
?– safeNot(C) when ground(C).
safeNot(C) :–
      call(C),
      !,
      fail.
safeNot(C).
```

This specifies that **safeNot/1** may be called when its argument is ground (that is, contains no variables, even in subterms). Thus **?– safeNot(sorted([1, 3, 2])).** would proceed, whereas **?– safeNot(sorted([1, 3, 2 | X])).** would delay.

Aggregate Functions

The following example was given by Naish [Nais84] and illustrates the use of **solutions/3**.

Example 10

Consider the following program:

```
drinks(tim, tea).
drinks(tim, milk).
drinks(tim, beer).

drinks(joe, tea).
drinks(joe, wine).
```

In the following queries, all variables in the goal are local.

```
?- solutions(P, drinks(P, tea), S).
      S = [joe, tim].

?- solutions(P, drinks(P, wine), S).
      S = [joe].

?- solutions(D, some P drinks(P, D), S).
      S = [beer, milk, tea, wine].
```

When there are global variables in the goal – *P* in the following example – **solutions/3** can have several answers. The global variables will be bound by **solutions/3** if they are bound by the goal inside **solutions/3**. There may also be other answers with delayed inequalities. These answers can often be avoided by forcing the set to have at least one element.

```
            % ... assuming that P would not be implicitly locally
            % quantified in the following query ...
?- solutions(D, drinks(P, D), S).
      S = [beer, milk, tea], P = tim ;
      S = [tea, wine], P = joe ;
      S = [], P ~= tim, P ~= joe.
```

where **P ~= tim** represents a delayed inequality.

Solutions/3 can also be called recursively (note the use of the term **PF.PR**, to ensure the set has at least one element)

```
?- solutions(D – (PF.PR), solutions(P, drinks(P, D), PF.PR), S).
      S = [beer–[tim], milk–[tim], tea–[joe, tim], wine–[joe]].
```

The delayed inequalities may also contain multiple variables and universal quantifiers:

```
?- solutions(_, some Y X = f(Y), []).
      all Y1 X ~= f(Y1).

?- solutions(_, X = f(Y), []).
      X ~= f(Y).
```

In all of the above examples, **setof/3** would return the same answers as **solutions/3**, except that in the cases with delayed inequalities **setof/3** would fail, and the notation **Var^Goal** is used instead of **some Var Goal**.

Example 11

The following are examples of queries, some of which use aggregate functions, which may be given to the top level of the interpreter.

```
?- R : min(T, member(T, [6, 2, 4]), R).

2


?- R : max(T, member(T, [pear, X, orange]), R), X = apple.

pear


?- : count(T, member(T, [pear, apple, orange, pear]), 3).

Yes


?- R : sum(S*3, T, member(T/S,
      [orange/10, pear/100, apple/1, apple/1, apple/1000]), R).

3333


?- A/B sorted A, -B : member(A/B, [2/3, 4/5, 2/6]).

2/6
2/3
4/5
```

Example 12

The following example queries are derived from Sections 7.3 and 7.4 (Built-in Functions and Update Operations) of the [Date81]. The examples refer to a simple supplier/part database consisting of three relations: **s(Supplier, Name, Status, City)**, **part(Part, Name, Colour, Weight, City)**, **sp(Supplier, Part, Quantity)**, and two rules to access the key fields: **s(Supplier)**, **p(Part)**.

7.3.1 Get the total number of suppliers.

```
C : count(S, s(S), C).
```

7.3.2 Get the total number of suppliers currently supplying parts.

```
C : count(S, sp(S, _, _), C).
```

7.3.3 Get the number of shipments for part **p2**.

```
C : count(S, sp(S, p2, _), C).
```

7.3.4 Get the total quantity of part **p2** supplied.

```
Sum : sum(Qty, S, sp(S, p2, Qty), Sum).
```

7.3.5 Get supplier numbers for suppliers with status value less than the current maximum status value in **s**/**4**.

```
S : s(S, _, St, _), St < Max, max(St1, s(_, _, St1, _), Max).
```

7.3.6 For each part supplied, get the part number and the total quantity supplied of that part.

```
P, Sum : p(P), sum(Qty, S, sp(S, P, Qty), Sum).
```

7.3.7 Get part numbers for all parts supplied by more than one supplier.

```
P : p(P), count(S, sp(S, P, _), C), C > 1.
```

7.3.8 For all parts such that the total quantity supplied is greater than **300** excluding from the total all shipments for which the quantity is less than or equal to **200**), get the part number and the maximum quantity of the part supplied; and order the result by descending part number within those maximum quantity values.

```
P, MQ sorted MQ, -P :
     p(P),
     sum(Q, S, (sp(S, P, Q), Q > 200), Sum),
     Sum > 300,
     max(Q1, sp(_, P, Q1), MQ).
```

7.4.1 Change the colour of part **p2** to **yellow**, increase its weight by **5**, and set its city to "unknown" (**null**).

```
update C to yellow, W to W1, City to null in p(p2, _, C, W, City)
     where W1 is W + 5.
```

7.4.2 Double the status of all suppliers in **london**.

```
update S to S1 in s(_, _, S, london) where S1 is 2 * S.
```

7.4.3 Set the quantity to zero for all suppliers in **london**.

```
update Q to 0 in sp(S, _, Q) where s(S, _, _, london).
```

7.4.4 Change the supplier number for **s2** to **s9**.

```
update S to s9 in s(S, _, _, _) where S = s2.
update S to s9 in sp(S, _, _) where S = s2.
```

7.4.5 Add part **p7** (name **washer**, colour **grey**, weight **2**, city **athens**) to table p.

```
insert p(p7, washer, grey, 2, athens).
```

7.4.6 Table **temp** has one column. Enter into **temp**, part numbers for all parts supplied by supplier **s2**.

```
insert temp(P) where sp(s2, P, _).
```

7.4.7 Delete supplier **s1**.

```
delete s(s1, _, _, _).
```

7.4.8 Delete all shipments.

```
delete sp(_, _, _).
```

7.4.9 Delete all shipments from suppliers in **london** and also the suppliers concerned.

```
delete sp(S, _, _) where s(S, _, _, london).
delete s(_, _, _, london).
```

APPENDIX 2
Definite Clause Grammars

Definite clause grammars (**DCGs**) are a powerful class of grammar that extend context-free grammars (**CFGs**) by allowing nonterminals to be compound terms, and by permitting sequences of calls to predicates to be embedded in productions [Pere80]. The notation for DCGs allows a more concise parser specification than would otherwise be possible in NU-Prolog, while still making the constructs of NU-Prolog available. DCGs productions are converted to NU-Prolog clauses by a preprocessor, available as the **-D** option to *nc*.

Using DCG notation, a CFG production has the form:

nonterminal --> $\texttt{item}_1$**, . . . ,**$\texttt{item}_n$**.**

where a **nonterminal** is a NU-Prolog atom, and each $\texttt{item}_i$ is a nonterminal or a sequence of terminals. A sequence of terminals is written as a list, where a terminal is an arbitrary term. A null sequence is written as the empty list. Some examples of CFG productions are:

```
full_statement --> [].

full_statement --> label, debugging, statement.

label --> [label(L), ':'].
```

The **-->** operator indicates to the preprocessor the presence of a DCG production. The second production above can be read as ‘‘**full_statement** can take the form of a **label**, followed by **debugging**, followed by a **statement**’’.

A DCG production takes an input list of terminals and ‘consumes’, from the front of the list, a sequence of terminals that fits its specification. Its output is those terminals remaining. To translate a DCG production into a NU-Prolog clause, each nonterminal is associated with a predicate of the same name with arguments for the input and output lists mentioned above. Thus the translated form of the previous examples would be:

```
full_statement(SIn, SIn).
full_statement(SIn, SOut) :-
      label(SIn, SOut1),
      debugging(SOut1, SOut2),
      statement(SOut2, SOut).
label(label(L).‘:’.SOut, SOut).
```

(The **.** form of list notation is used in the example translations for clarity.) Hence, the DCG notation hides the two arguments that all predicates corresponding to nonterminals must have.

In extending the notation described above to that of full DCGs, nonterminals may be compound terms. Where this is the case, the corresponding predicates in the translation will have arguments in addition to the input and output arguments; the input and output arguments will always appear last. For example:

```
const_declaration(const(I, N)) -->
      [ident(I), '=', number(N)].
```

becomes:

```
const_declaration(const(I, N), ident(I).'='.number(N).SOut, SOut).
```

To permit sequences of calls to predicates to be embedded in a production, an item may be a sequence of calls enclosed in braces, for example:

```
declaration_check(type(I)) -->
      [],
      {
            writeln('Declaration check'),
            defined(I)
      }.
```

The DCG preprocessor allows for these items by not expanding subgoals enclosed in braces. This additional notation gives DCGs general programming power.

The following DCG production and its translation, defining a Pascal-style **for** statement, demonstrate the power of the DCG notation.

```
statement -->
      [for], variable(Type), {enumerable(Type)},
      [':='], expression(Type),
      ([to] ; [downto]), expression(Type), [do],
            statement.
```

This becomes:

```
statement(for.SIn, SOut) :-
      variable(Type, SIn, SIn1),
      enumerable(Type),
      SIn1 = ':='.SIn2,
      expression(Type, SIn2, SOut1),
      (
            SOut1 = to.SIn3
      ;
            SOut1 = downto.SIn3
      ),
      expression(Type, SIn3, do.SIn4),
      statement(SIn4, SOut).
```

A well known parsing ambiguity is that associated with the **else** part of a Pascal-style **if–then–else** statement. The requirement that an **else** be associated with the syntactically closest unmatched **if–then** is achieved below through the simple application of the NU-Prolog **if–then–else** construct. The NU-Prolog **if–then–else** precludes the association of an **else** with the wrong **if–then**, even on backtracking. Parentheses distinguish the use of the terminals **if**, **then** and **else** from their NU-Prolog usage.

```
statement -->
      [(if)], expression(boolean), [(then)],
            statement,
      if [(else)] then
            statement.
```

This becomes:

```
statement((if).SIn, SOut) :-
      expression(boolean, SIn, (then).SIn1),
      statement(SIn1, SOut1),
      (if SOut1 = (else).SIn2 then
            statement(SIn2, SOut)
      else
            SOut1 = SOut
      ).
```

Two predefined unary non-terminals that can be used in DCG productions are **append** and **=**. After preprocessing, they will function as calls to the standard nuprolog **append/3** and as ternary unification '**=(S,S,S).**' respectively. The first of these is useful for prepending to the input list (instead of the usual consuming), the second for examining the input list (ie. look ahead). The output of the preprocessor may be optimized for these non-terminals, as demonstrated by the example of **append** given later.

Finally, a further form of DCG production is allowed that permits an output terminal list that is longer than that provided as input. The left-hand side of a production may be augmented with a sequence of terminals to be prepended to the output list. This extended output terminal list will be parsed normally by productions invoked later with the extended list as input.

The productions that follow, in conjunction with the definition of **full_statement** given above, would allow the use of the keyword **debug** to indicate statements that can be turned off or on for debugging purposes. The production defining the **if** statement can then handle this construct, with only the definition of a boolean **$debug** being required.

```
debugging, [(if), identifier($debug), (then)] --> [debug].
debugging --> [].
```

can be expressed equivalently as

```
debugging --> [debug], append([(if), identifier($debug), (then)]).
debugging --> [].
```

Either of these will be translated to

```
debugging(debug.SIn, (if).identifier($debug).(then).SIn).
debugging(SIn, SIn).
```

APPENDIX 3
Using the NU-Prolog Compiler

Using *nc* is like using the *C* compiler, *cc*. For example, consider a program which consists of three files, **ex.nl**, **infer.nl**, **util.nl**. An executable image of this program, **ex**, could be produced by using the command

```
nc -o ex ex.nl infer.nl util.nl
```

Alternatively, this could be done in stages:

```
nc -c ex.nl
nc -c infer.nl
nc -c util.nl
nc -o ex ex.no infer.no util.no
```

It is simplest to use *cake* (1)† to compile this program. If *make* (1) is to be used, it is necessary to explicitly state a rule for the dependency between **.nl** and **.no** files at the head of the **Makefile**. A suitable **Makefile** for the program would be:

```
NC=/usr/bin/nc
NIT=/usr/bin/nit

.SUFFIXES: .nl .ns .no
.nl.no: ; $(NC) -c $*.nl
.nl.ns: ; $(NC) -S $*.nl

SRCS= ex.nl infer.nl util.nl
OBJS= ex.no infer.no util.no

ex:   $(OBJS)
      $(NC) -o ex $(OBJS)

nit:  $(OBJS)
      $(NIT) $(OBJS) > nits
```

A suitable **Cakefile** for the program would be

```
#define MAIN    ex
#define FILES   ex infer util

#include <Nuprolog>
#include <Main>
```

Once the object file is generated, it is executed by typing its name, possibly followed by some command line arguments for the program. These arguments will be made available to the entry predicate as a list of atoms. For example:

```
./ex a /mip/db c
```

will invoke the **ex** program with the goal **main([npm, a, '/mip/db', c])**.

† *cake* (1) was written by Zoltan Somogyi (zs@cs.mu.oz.au) and is distributed with NU-Prolog.

APPENDIX 4
Porting MU-Prolog Programs to NU-Prolog

This Appendix describes changes which should be made to MU-Prolog programs when porting them to NU-Prolog.

(1) If a predicate definition is to be modified, declare it as **dynamic/1**. For example, if the predicate **p/1** is to be modified, the MU-Prolog definition

```
p(a).
```

should be replaced by

```
?- dynamic p/1.
p(a).
```

(2) The names of the safe and unsafe negation predicates have been changed.

	MU-Prolog	NU-Prolog
safe	~	not
unsafe	not, \+	\+

~ is not supported by NU-Prolog.

(3) Some MU-Prolog predicates, such as **wflags/1**, will never be implemented, but the functionality is given by other predicates.

(4) Most MU-Prolog library predicates, such as **append/3** and **ground/1**, have been built into NU-Prolog.

(5) Some MU-Prolog predicates exist but under different names. Using the library **compat** may solve this problem.

(6) **wait** declarations have been superseded by **when** declarations. Although they are not functionally equivalent, it is usually easy to replace a given **wait** declaration by a **when** declaration.

APPENDIX 5
Using DSIMC Databases

The predicates and parameters for manipulating **dsimc** databases are described fully in section 5.8. The purpose of this section is to work through the process of setting up a simple database using the predicates described in that section. This should provide a good basis for understanding precisely what needs to be done to use external **dsimc** databases with NU-Prolog.

Consider a database describing a network of computers such as the UUCP network. A model of this network will describe entities such as nodes and connections. Let us say that we use two relations to model the network: **node** and **connection**.

The **node** relation contains the (unique) network name of the computer, and a brief description of the location of the machine†. This can be implemented as a Prolog predicate **node/2**, containing such facts as:

```
node(mulga, "Computer Science, Melbourne University, Australia").
node(munnari, "Computer Science, Melbourne University, Australia").
node(seismo, "Centre for Seismic Studies, Virginia, USA").
```

The **connection** relation contains the names of the **node**s at either end, and a cost; we assume that this will denote a one way connection, and that if communication is possible in the other direction there will be a separate entry, possibly with a different cost. This can be implemented as a Prolog predicate **conn/3**, containing facts such as:

```
conn(munnari, seismo, 150).
conn(munnari, mulga, 10).
```

Before we assign parameters to a relation, we must determine the various properties of the relation:

* It is useful if we know approximately how large an average record will be for a given relation, that is, how many characters will be required to store the values of the fields and field separators. If records are very large, then we might wish to increase the sizes of segments from the default value of 4096, to allow at least 50 records per segment (remembering that around half of the segment space is occupied by index descriptors and record pointers). It is also useful to know approximately how many facts there are likely to be in each relation.

* We should have some idea of the types of queries which are going to be asked on the relation. For example, it is highly likely that we will be asking for a list of all sites which are connected to a given site and so we will generally have the first field specified in a query on the **conn/3** relation. It is far less likely that we will ask for all sites which have a given cost, and so we would expect the cost field of the **conn/3** relation to be infrequently specified in queries.

Let us suppose that the average site name is 7 characters long, and the average site description is 40 characters long. Then the average **node** record will require 50 characters to store (7 for the

---

† Note that the descriptive information is precisely that – descriptive. As currently organised, it can play no ‘‘active’’ role in queries on the database. If we wanted to use the location information, for example to partition nodes geographically, we would impose a more rigid structure on the location and probably allocate new fields for the specific components such as **country**, **state**, **organisation**, **department**, giving, for example, **node(seismo,Centre for Seismic Studies,virginia,usa)**.

site name, 40 for the description, plus a comma to separate the fields and two quotes around the description). If the average cost is a four-digit number, then **conn** records will require 20 characters (14 for two site names, 4 for the cost value, and 2 commas to separate the fields). Assuming that there are around 10000 sites (therefore 10000 **node** records), and that each site is connected to 10 others, then there will be around 100,000 **conn** records.

The following program (annotated with line numbers) will set up the database with parameters derived from the above considerations:

```
01: %
02: % Create a network database and initialise the relations,
03: %
04:
05: main(_.[]) :-
06:  writeln('usage: creatdb database'),
07:  exit(1).
08: main(_.DB.[]) :-
09:  (dbCons(DB) ->
10:           ($ondisc(DB, node(_, _), _) ->
11:                   dbUndefine(DB, word, 2)
12:           ),
13:           ($ondisc(DB, conn(_, _, _), _) ->
14:                   dbUndefine(DB, letter, 3)
15:           ),
16:  ;
17:           dbCreate(DB)
18:  ),
19:  dbDefine(DB, node, 2,
20:           [scheme=dsimc,
23:            avrec=50,
24:            nrec=10000,
25:            template="g:0:0(g:1:7fffffff,g:0:0)"]),
26:  dbDefine(DB, conn, 3,
27:           [scheme=dsimc,
28:            avrec=20,
29:            nrec=100000,
30:            template="g:0:0(g:8:29228922,g:4:12892289,g:1:44545454)"]).
31: main(_) :-
32:  writeln('Problems creating database/relation').
```

Note that the program requires one argument on the command line which is the name of the database (line 10). On line 11, we attempt to open the database, in case it already exists; if the **dbCons** fails, then we assume that it doesn't and try to create it (line 19). On lines 12 and 15 we are testing whether the relations **node** and **conn** already exist; if they do, then we delete their existing definitions with **dbUndefine** (lines 13 and 16). The parameters for the **node** relation (lines 22-24) have been explained above, but we note here that the template (line 25), is set up to indicate that we will never use the second field for indexing. We have designed the template for the **conn** relation on the basis that we are very likely to ask queries using the first field (weight:8), occasionally using the second field (weight:4), and very rarely using the cost field (weight:1). The **g** values in the template fields indicate that the corresponding fields are always ground, which is the only sensible value for a DSIMC database.

APPENDIX 6
Reserved Predicate Names

The following predicate names have been reserved for future use.

apply			
count	countIf	countIfNot	
delete	deleteIf	deleteIfNot	deleteDuplicates
find	findIf	findIfNot	
funcall			
lambda			
mapcar	mapcars	maplist	maplists
mapcon	mapcons	mapcan	mapcans
notrace			
position	positionIf	positionIfNot	pred
reduce			
type			

APPENDIX 7
Permitted Characters

SYMBOL CHARACTERS

These include: **+ - * / ~ < = > ‘ ~ : . ? @ # &**

PRINTABLE CHARACTERS

These include graphic characters and white space characters.

Graphic characters

ASCII codes 33 to 126

White space

ASCII code 9 (horizontal tab)
ASCII code 10 (new line)
ASCII code 12 (form feed)
ASCII code 13 (carriage return)
ASCII code 32 (space)

PERMITTED CHARACTERS

Permitted characters are characters which may appear in the text of strings or atoms (subject to rules concerning quoting). These include the printable characters or an escape sequence:

Escape Sequence	Character
b	Backspace
c	Ignore up to but not including next graphic character
d	Delete
e	Escape
f	Form feed
n	New line
r	Carriage return
s	Space
t	Tab
v	Vertical Tab
octal_string	Character given by one to three octal digits
\\~*char*	Control character with ASCII code of *char* mod 32
non-printable	Ignore *non-printable* character
other	Character *other*

PrologFlag/3 may be used to turn off character escapes, so that the only escape sequence recognised is repeated single quotes (in quoted atoms) and repeated double quotes (in strings). All other sequences are taken literally.

A complete list of ASCII characters follows.

ASCII Code	Graphic and (Non-Graphic) Characters	Octal Escape Sequence	Other Escape Sequences
0	()	\0	\~@
1	()	\1	\~A
2	()	\2	\~B
3	()	\3	\~C
4	()	\4	\~D
5	()	\5	\~E
6	()	\6	\~F
7	()	\7	\~G
8	(Backspace)	\10	\~H or \b
9	(Horizontal tab)	\11	\~I or \t
10	(Newline)	\12	\~J or \n
11	(Vertical tab)	\13	\~K or \v
12	(Form feed)	\14	\~L or \f
13	(Carriage return)	\15	\~M or \r
14	()	\16	\~N
15	()	\17	\~O
16	()	\20	\~P
17	()	\21	\~Q
18	()	\22	\~R
19	()	\23	\~S
20	()	\24	\~T
21	()	\25	\~U
22	()	\26	\~V
23	()	\27	\~W
24	()	\30	\~X
25	()	\31	\~Y
26	()	\32	\~Z
27	(Escape)	\33	\~[or \e
28	()	\34	\~
29	()	\35	\~]
30	()	\36	\~~
31	()	\37	\~_

ASCII Code	Graphic and (Non-Graphic) Characters	Octal Escape Sequence	Other Escape Sequences
32	(Space)	\40	\s
33	!	\41	
34	"	\42	
35	#	\43	
36	$	\44	
37	%	\45	
38	&	\46	
39	’	\47	
40	(	\50	
41	)	\51	
42	*	\52	
43	+	\53	
44	,	\54	
45	-	\55	
46	.	\56	
47	/	\57	
48	0	\60	
49	1	\61	
50	2	\62	
51	3	\63	
52	4	\64	
53	5	\65	
54	6	\66	
55	7	\67	
56	8	\70	
57	9	\71	
58	:	\72	
59	;	\73	
60	<	\74	
61	\=	\75	
62	>	\76	
63	?	\77	

ASCII Code	Graphic and (Non-Graphic) Characters	Octal Escape Sequence	Other Escape Sequences
64	@	\100	
65	A	\101	
66	B	\102	
67	C	\103	
68	D	\104	
69	E	\105	
70	F	\106	
71	G	\107	
72	H	\110	
73	I	\111	
74	J	\112	
75	K	\113	
76	L	\114	
77	M	\115	
78	N	\116	
79	O	\117	
80	P	\120	
81	Q	\121	
82	R	\122	
83	S	\123	
84	T	\124	
85	U	\125	
86	V	\126	
87	W	\127	
88	X	\130	
89	Y	\131	
90	Z	\132	
91	[	\133	
92	\	\134	
93	]	\135	
94	~	\136	
95	_	\137	

ASCII Code	Graphic and (Non-Graphic) Characters	Octal Escape Sequence	Other Escape Sequences
96	‘	\140	
97	a	\141	
98	b	\142	
99	c	\143	
100	d	\144	
101	e	\145	
102	f	\146	
103	g	\147	
104	h	\150	
105	i	\151	
106	j	\152	
107	k	\153	
108	l	\154	
109	m	\155	
110	n	\156	
111	o	\157	
112	p	\160	
113	q	\161	
114	r	\162	
115	s	\163	
116	t	\164	
117	u	\165	
118	v	\166	
119	w	\167	
120	x	\170	
121	y	\171	
122	z	\172	
123	{	\173	
124	|	\174	
125	}	\175	
126	~	\176	
127	(Delete)	\177	\d

APPENDIX 8
Standard Operator Declarations

```
?- op(1200, xfx, (:-)).
?- op(1200, xfx, (-->)).
?- op(1200, fx, (?-)).
?- op(1200, fx, (:-)).
?- op(1180, fx, (useIf)).
?- op(1175, fx, (:)).
?- op(1175, xfx, (:)).
?- op(1175, xfx, where).
?- op(1175, fy, insert).
?- op(1175, fy, delete).
?- op(1175, fy, update).
?- op(1172, xfx, (in)).
?- op(1171, xf, (sorted)).
?- op(1171, xfx, (sorted)).
?- op(1170, xfy, (else)).
?- op(1160, fx, (if)).
?- op(1150, fy, dynamic).
?- op(1150, xfx, (then)).
?- op(1150, fy, pure).
?- op(1100, xfy, (;)).
?- op(1100, fx, (type)).
?- op(1050, xfy, (->)).
?- op(1000, xfy, (',')).
?- op(980, xfx, (to)).
?- op(950, fxy, some).
?- op(950, fxy, gSome).
?- op(950, fxy, gAll).
?- op(950, fxy, all).
?- op(920, xfy, =>).
?- op(920, xfy, <=>).
?- op(920, xfy, <=).
?- op(900, xfx, when).
?- op(900, fy, man).
?- op(900, fy, wait).
?- op(900, fy, spy).
?- op(900, fy, once).
?- op(900, fy, not).
?- op(900, fy, nospy).
?- op(900, fy, ls).
?- op(900, fy, listing).
?- op(900, fy, lib).
?- op(900, fy, \+).
?- op(900, fy, (~)).
?- op(740, xfy, or).
?- op(720, xfy, and).
?- op(700, xfx, ~=).
?- op(700, xfx, is).
?- op(700, xfx, \==).
?- op(700, xfx, \=).
?- op(700, xfx, @>=).
?- op(700, xfx, @=<).
?- op(700, xfx, @>).
?- op(700, xfx, @<).
?- op(700, xfx, >=).
?- op(700, xfx, >).
?- op(700, xfx, =\=).
?- op(700, xfx, ==).
?- op(700, xfx, =<).
?- op(700, xfx, =:=).
?- op(700, xfx, =..).
?- op(700, xfx, =).
?- op(700, xfx, <).
?- op(600, xfy, '.').
?- op(500, yfx, \/).
?- op(500, yfx, /\).
?- op(500, yfx, -).
?- op(500, yfx, +).
?- op(500, fx, \).
?- op(500, fx, (-)).
?- op(500, fx, (+)).
?- op(400, yfx, >>).
?- op(400, yfx, <<).
?- op(400, yfx, //).
?- op(400, yfx, /).
?- op(400, yfx, *).
?- op(300, xfx, mod).
?- op(300, xfy, **).
?- op(200, xfy, '^').
```

APPENDIX 9
Signals

The following are the signal names and numbers that can be used in conjunction with **signal/2**. If the signal is not supported under the user's version of UNIX, the effect of using **signal/2** is undefined.

Signal name	Signal number	Effect
sighup	1	Hangup
sigint	2	Interrupt (rubout)
sigquit	3	Quit (ASCII FS)
sigill	4	Illegal instruction (not reset when caught)
sigtrap	5	Trace trap (not reset when caught)
sigiot	6	IOT instruction
sigemt	7	EMT instruction
sigfpe	8	Floating point exception
sigkill	9	Kill (cannot be caught or ignored)
sigbus	10	Bus error
sigsegv	11	Segmentation violation
sigsys	12	Bad argument to system call
sigpipe	13	Write on a pipe with no one to read it
sigalrm	14	Alarm clock
sigterm	15	Software termination signal from kill

APPENDIX 10
Bugs and Limitations of NU-Prolog

There are a number of known bugs and limitations to the NU-Prolog system (these are commonly called ‘‘features’’). Some of these may be eliminated in the future; others are technical or theoretical problems which cannot be eliminated.

Limitations of Call/1

Quantification of variables

If a variable name occurs in separate scopes, it will refer to the same variable, and is a programming error.

Uniquely occurring variables are not quantified, so that, for example, **`call(not p(_))`** will delay.

Undefined Predicates

System predicates which are defined at compile time, but not otherwise, may cause *nc* to print ‘‘undefined predicate’’. This also applies to user-defined predicates which occur in goals in NU-Prolog source.

Occur Check

The unification algorithm used by NU-Prolog does not make the ‘‘occur check’’.

nit

nit does not know about old-fashioned quantification, that is, of the form **`V^Goal`**.

Floundering

Floundering in stand alone programs is not reported.

Bugs in Manual

NU-Prolog is under constant development. Parts of the manual, including this Appendix, may contain errors. Theoretical considerations preclude a guarantee that the preceding sentences, or this sentence, is correct.

REFERENCES