Sceptic Version 4
User Manual

May 1993
Revision 1: June 1993

R. Cooper
J. Farringdon

Department of Psychology
University College London

Sceptic was originally developed by S. Hajnal, J. Fox & P. Krause at the Imperial Cancer Research Fund (London), and the current version retains much from this work. This manual is based on the manual for version 3 (Hajnal *et al.*, 1989), and includes material from that manual.

Preface

Sceptic is a programming language originally developed at the Imperial Cancer Research Fund (London) (c.f. Hajnal *et al.*, 1989). It has been successfully used to implement complex, time-evolving systems (e.g., biological process simulations (Zicha & Fox, 1991) and logical reasoning mechanisms (Hajnal *et al.*, 1989)), autonomous systems (Fox, 1992), planning systems (O'Neill, 1992) and control processes in image interpretation and medical problem solving (Walker, 1991). It is currently being developed at UCL as a modelling language within the context of a modelling methodology for cognitive psychology. The project, which involves R. Cooper, J. Farringdon, J. Fox and T. Shallice, is aimed at developing a systematic methodology, with appropriate computational support, for computational modelling. Within the bounds of this project Sceptic has been used to develop rational reconstructions of previously implemented theories as well as implementations of theories which previously were only described verbally (see the associated case studies).

The work on cognitive modelling has taken Sceptic as a starting point because:

a. the language interpreter supports a number of mechanisms that are commonly assumed in cognitive theories (e.g., pattern-directed processing, content addressable memory retrieval/update, sequential processing and parallel data propagation modes); and

b. it has a simple but expressive syntax which assists clear and succinct representation of data-structures and processes, and its execution model is simple and easily described.

Sceptic runs under several Prologs. The environment currently comprises database facilities (i.e., Prolog's database), a rule interpreter, a set of libraries containing code that we have found useful in a number of modelling applications, and a debugger. It is envisaged that this may be extended to include object oriented facilities and a graphical interface.

This version of Sceptic retains the distinctive control structure of previous versions of Sceptic (triggered productions, or conditional rewrite rules), as developed at ICRF, but differs from previous versions on a number of counts. A complete list is given in an appendix, but it is worth noting here the altered relation between Sceptic and Prolog. In previous versions, the Sceptic rule interpreter was an extension of Prolog which ran on top of Prolog. It was possible to mix Sceptic and Prolog in an arbitrary manner by escaping to the Prolog interpreter which was running underneath the Sceptic interface. Version 4 attempts to redefine the relation between Sceptic and Prolog in two ways. First, Sceptic rules and standard Prolog predicates are viewed as having very different functions within the system, and to reflect this the mixing of rules and predicates is highly constrained. Second, the interface has been substantially altered such that it differentiates Sceptic rules and Prolog predicates and acts accordingly. In doing this we have attempted to import many aspects of Prolog, as transparently as possible, into Sceptic. Users familiar with Prolog will note many features of Sceptic which have been imported (often without explicit mention) from Prolog.

R. Cooper, J. Farringdon
May 19, 1993

Contents

1 The Sceptic Language

Sceptic is a programming language which extends Prolog (see, for example, Clocksin & Mellish, 1981) by the addition of a forward chaining control structure (the 'conditional rewrite rule'). This control structure allows procedural control aspects of a program to be distinguished from purely declarative aspects, which may be expressed in a subset of standard Prolog.

The Sceptic execution model revolves around a changing database. Rewrite rules test this database and conditionally modify the database or trigger further rewrite rules. A Sceptic program generally consists of a specification of an initial state of the database and a set of rules which specify how the database evolves over time, or how the database changes in response to particular events.

1.1 Basic concepts

Sceptic interfaces to its database via two basic kinds of entities:

Conditions: These are similar to Prolog predicates. They are purely logical objects which may query the database but not alter it. Paralleling Prolog predicates, conditions may succeed, instantiating variables contained therein, or fail.

Actions: These explicitly perform what are normally side effects within Prolog. Most importantly, they allow output and database modification. There is no notion of success or failure for actions, and they do not instantiate variables.

Conditions are either primitive or complex. The primitive conditions are just the Prolog primitives which have no side-effects. Complex conditions may be defined via predicate definitions as in standard Prolog, with the restriction that only conditions (primitive or complex) may be employed in the definition. In particular, Prolog primitives with side-effects (such as those which modify the database) may not be employed in the definition of complex conditions. This ensures that all conditions, primitive or complex, have no side-effects.

Actions, like conditions, may also be primitive or complex. The primitive actions are those Prolog primitives which have side-effects: most notably, database modification primitives and output primitives. Complex actions (which are sometimes referred to as 'non-terminals') are defined via 'triggered productions', or 'conditional rewrite rules'. These provide declarative definitions of processes in terms of their subprocesses and the conditions which govern their execution.

The two varieties of statement available in Sceptic allow a division between process control, which is specified in terms of forward chaining rewrite rules, and the declarative specification of conditions implicated in that control. That is, a programming style may be adopted within Sceptic whereby a program consists of a set of conditional rewrite rules which specify the possible control processes, together with a set of predicate definitions which specify the conditions which govern that control.

1.2 Syntax

Much of Sceptic's syntax is inherited from Prolog. In particular, all terms in Sceptic are Prolog terms and the same conventions are used for distinguishing variables and ground terms. Thus, ground terms are either atomic (consisting of either a string of alphanumeric characters beginning with a lowercase letter or an arbitrary sequence of characters enclosed in single quotes), or complex (consisting of an atomic term followed by an open bracket, a comma separated argument list, and a close bracket) and variables are strings of alphanumeric characters beginning with an uppercase letter or an underscore. `term`, `'Time 14'` and `attribute(packet, square)` are terms. `Time14` and `_Target` are variables.

As mentioned above, complex conditions are defined via standard Prolog, *viz*:

```
Condition :- SubConditions.
```

In such definitions `Condition` is a term representing the complex condition being defined. `SubConditions` is a comma separated, or semi-colon separated, sequence of terms. `Condition` is true (or succeeds) if the subconditions succeed. Conjunctive subconditions are separated by commas; disjunctive subconditions are separated by semi-colons. (Hence ":-" is read as "if"; "," is read as "and"; and ";" is read as "or".) Multiple definitions may be given for a single condition. Such definitions are read disjunctively. Further details may be obtained from any Prolog reference (e.g., Clocksin & Mellish, 1981).

Conditional rewrite rules have the syntax:

```
NonTerminal: Conditions => Expansion.
```

In such rules, `NonTerminal` is a term (naming the action being defined) and `Conditions` and `Expansion` are comma separated sequences of terms. The terms which constitute `Conditions` are each conditions (being either primitive or complex as above) and the terms which constitute `Expansion` are each actions (again, being either primitive or complex). Multiple definitions may be given for a single non-terminal. Such definitions are read conjunctively—they define simultaneously possible expansions.

For example, the following rewrite rule will write to the screen all clauses of the form `f(X,Y)`.

```
showf: f(X,Y) => write(f(X,Y)), nl.
```

Conditional rewrite rules may be read as "if `Conditions` hold (at rewrite time), then the process named `NonTerminal` consists of the sequence of subprocesses `Expansion`", or in more declarative terms, "if `Conditions` hold (at rewrite time), then `NonTerminal` rewrites as `Expansion`".

1.3 Declarations

A Sceptic program must also include declarations for any database elements required. This can be done within a Sceptic file with a `dynamic/1` directive (see the following section). The following (from the motive processing case study) declares that Sceptic's database will contain elements of the form `motive(X, Y, Z)`:[1]

```
:- dynamic motive/3.
```

Sceptic accepts two other declarations: `mode/1` and `public/1`. These are currently ignored, but are recommended for documentation purposes. A `mode` declaration specifies the intended instantiation patterns of the arguments of a condition. If, for example, the condition `permute/2` is intended to be called with its first argument instantiated and yield a value for its second argument, then this would be declared as:

```
:- mode permute(+Arg1,-Arg2).
```

The intended instantiation pattern for calls to `permute/2` is indicated by the prefixes to the arguments: '`+`' indicates that the corresponding argument should be instantiated; '`-`' indicates that it should be uninstantiated; '`?`' indicates that it may be either; and '`:`' indicates it should be a callable condition. This last option only relates to metapredicates: predicates which call their arguments. (Note the `mode` declarations are not relevant for triggers. As rewrite rules cannot return values, all of their arguments should be instantiated when they are called.)

The `public/1` declaration is used for applications which are composed of several files. In any file, each predicate/rewrite rule which is called by predicates/rewrite rules from some other file should be declared as public. Thus if `permute/2` is defined in one file but called from another, then this should be declared in the file in which `permute/2` is defined as follows:

```
:- public permute/2.
```

[1] `dynamic` is defined as a prefix operator, and so brackets are not required around its argument.

1.4 Directives

Other directives, apart from declarations, may also occur in a Sceptic file. Directives are distinguished by being preceded by ':-', and are executed via the standard Sceptic mechanism when the Sceptic file is being consulted/compiled. Thus if a directive is a condition then all solutions of that condition will be printed, and if it is an action it will be triggered. Useful directives (apart from `dynamic/1`) are `consult/1` and `compile/1`.

1.5 Comments

Comments use the normal Prolog conventions: any material between an unquoted `%` sign and the end of that line, any material between the symbols `/*` and `*/` is treated as a comment.

2 Using Sceptic

2.1 Starting the system

To start the system under Unix, type `sceptic` at the prompt (here `%`):

```
% sceptic
```

The system starts with a message stating the current version, and a prompt ('`sc> `').

```
Sceptic 4.0 Beta (SICStus prolog 2.1 #7; Sparc).
sc>
```

This is an "empty" system. The first step is normally to load a Sceptic application, consisting of one or more files of rewrite rules, predicates, and directives.

2.2 Loading a file

By default Sceptic filenames end with the extension '`.sc`'. To consult a file called `test.sc`, type:

```
sc> consult(test).
```

The file extension `.sc` will be used automatically, unless the file `test` exists. A filename with extension may also be given in full (with quotes as required by the Prolog parser) and must be given if it differs from the default extension:

```
sc> consult('test.1').
```

As with Prolog, `consult/1` may take a list of filenames as arguments (in which case the files are loaded in order). Again, as with Prolog, if a list is typed at the prompt Sceptic attempts to interpret the elements of that list as filenames and consult the corresponding files. The following is thus the shorthand way of consulting several files:

```
sc> [file1, file2, file3, file4].
```

If Sceptic is in verbose mode (see section 2.6), feedback will be given whilst consulting, listing the heads and triggers of conditions and rewrite rules loaded.

Once a file has been loaded, the rewrite rules and conditions present can be inspected by issuing the following commands to the '`sc> `' prompt:

`listing.`	list all rewrite rules
`listing(Term).`	list rewrite rules matching Term

Note that, as in Prolog, all commands must be terminated with a period ('`.`'). The command `listing(Term)` takes as its argument either a functor (e.g., `member`), in which case all rewrite rules whose trigger is that functor and all conditions whose head is that functor (and with any arity) are displayed, a functor/arity specification (e.g., `member/2`), in which case only those rewrite rules/conditions whose trigger/head is that functor with that arity are displayed, or a structure (e.g., `member(X,Y)`), in which case only those rewrite rules/conditions whose trigger/head unifies with that structure are displayed.

2.3 Example: mini.sc

Here is a very small application. It is contained in the file `mini.sc` in the `Examples` directory of the distribution.

```
% File: mini.sc

% Declare f/2 as dynamic:

:- dynamic(f/2).
```

```
% On the trigger input(Patt), assert the pattern as a clause
% for a predicate f(Patt,Source) with Source as u for user,
% but only if that Patt is not already present

input(X): not(f(X,u)) => assertz(f(X,u)).

% On the trigger showf, output an indented list of items entered so far.
% (Note that a rewrite rule needing no conditions other than the trigger
% has the single trivial condition 'true'.)

showf: true => write('Items entered so far:'), nl.
showf: f(X,u) => write('   '), write(X), nl.
```

and an example run (remarks in {...} do not appear on the screen):

```
% sceptic
Sceptic 4.0 Beta (SICStus prolog 2.1 #7; Sparc).
sc> consult(mini).                          { load application     }
{loading /usr/local/sceptic/Examples/mini.sc...}
{/usr/local/sceptic/Examples/mini.sc loaded, 30 msec 0 bytes}
sc> showf.                                  { showf is a trigger   }
Items entered so far:                       { there aren't any yet }
sc> input(fred).                            { input(X) is a trigger }
sc> showf.
Items entered so far:
   fred                                     { now both rewrite rules for showf fire }
sc> input(jo).
sc> showf.
Items entered so far:
   fred
   jo
sc>
sc> listing(f).                             { an example of calling a primitive action }
f(fred,u).                                  { listing shows the 'raw' clauses          }
f(jo,u).
sc> halt.                                   { exit Sceptic and return to unix }
%
```

2.4 Typing input to the prompt

The example above demonstrates reading triggers from the prompt and processing them according to Sceptic's execution model. The top level loop can distinguish between conditions and actions, and when a condition is entered it returns bindings for any variables in that condition. For example:

```
sc> X is 2 + 3.                         { This is a Prolog condition }
5 is 2 + 3
sc> member(X, [a,b,c]).                 { Assuming the standard definition of member/2 }
member(a, [a,b,c])
member(b, [a,b,c])
member(c, [a,b,c])
sc>
```

A comma separated sequence of conditions and actions may also be entered at the prompt. Sceptic will process each condition/action in the sequence separately. Thus, given the input:

```
sc> condition1, condition2, action1, condition3.
```

Sceptic will first evaluate `condition1`, listing all solutions. It will then evaluate `condition2` before firing `action1` and finally evaluating `condition3`. Note that as each term is processed separately, variables instantiated in any condition will *not* be bound across terms.

2.5 Initialisation: the .scepticrc file

If you have a `.scepticrc` file in your home directory, or anywhere on the Sceptic search path (see section 6), then this file will be consulted immediately upon Sceptic starting.

This file typically contains settings and directives like `sc_library_path(My_Search_Path)`, `debug`, and `verbose`, together with personal predicate and rule definitions, for example `prolog_mode/1` and `list/1`.

For the sake of portability, we recommend that applications should not be dependent upon entries in a `.scepticrc` file. Any such entries should be moved from the `.scepticrc` file into either the application or a suitable library (see section 6).

2.6 Verbose mode

In verbose mode, Sceptic gives feedback whilst consulting files, listing all predicates and triggers loaded. Verbose mode is entered by issuing the command `verbose` at the Sceptic prompt. The command `noverbose` returns Sceptic to its normal mode.

2.7 Recovery from error

In the event of something going disastrously wrong, one may end up back in Prolog (e.g., as the result of using ⟨crtl⟩-C). Sceptic may be restarted from Prolog by typing '`sc.`'

3 The Execution Cycle

3.1 Expansion of actions

When a trigger is called, each rewrite rule matching that trigger is taken in turn. All instantiations of its conditions are found and the corresponding actions queued for execution. This constitutes a single 'burst of expansion'. A Sceptic cycle consists of a series of such bursts, continuing until there are no more triggers to process.

Queued actions are interpreted one at a time. If they are triggers, they are called as above, causing another burst of expansion. If they are primitive actions they are directly executed (without instantiating variables). An error message is printed if a queued action is neither a trigger nor a primitive.

Essentially, then, execution revolves around a stack of unprocessed actions, and the processing cycle involves popping the top element off this stack and either executing it (if it is a primitive), or expanding it (otherwise). Figure 1 illustrates the stack for several cycles of processing an example trigger.

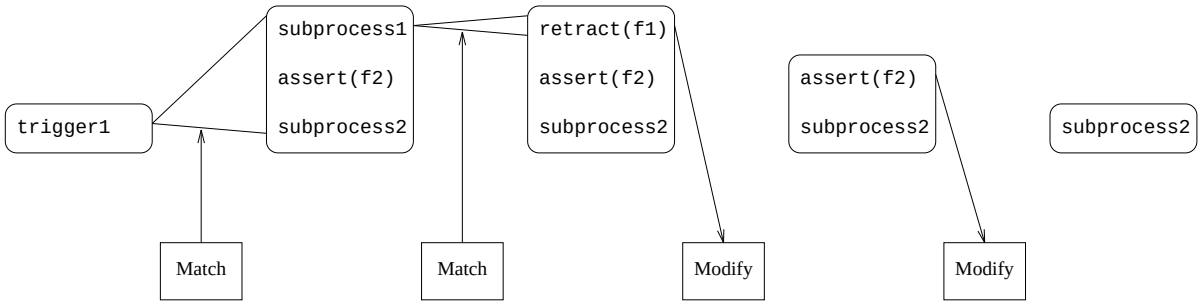

Figure 1: The Sceptic stack during execution

The following points should be noted:

a. All rewrite rules matching a trigger are used. There is no notion of a 'cut' (as in standard Prolog) to indicate that the 'right' rewrite rule has been found.

b. All instantiations of conditions are found before any actions take place (separation of 'recognise' and 'act' phases). Therefore, for example, an assertion action cannot cause a new instantiation to be found on that same burst of expansion.

c. Actions are stacked rather than queued. That is, the expansion of a trigger is pushed onto the top of the stack (resulting in depth first processing) rather than being added to the end of a queue (which would result in breadth first processing).

3.2 Example: Database modify and print

Consider the following rewrite rules (assuming the usual Prolog definition of `member/2`).

```
print_elements_and_modify_db(List):
     member(Z, List)
  => write(Z), nl,
     assert_or_retract(Z).

assert_or_retract(+X):
     not(clause(X, true))
  => assert(X).
```

```
assert_or_retract(+X):
     clause(X, true)
  => write(X), write(' is already in the database.'), nl.
assert_or_retract(-X):
     clause(X, true)
  => retract(X).
assert_or_retract(-X):
     not(clause(X, true))
  => write(X), write(' is not in the database.'), nl.
```

The trigger `print_elements_and_modify_db/1` takes as its argument a list where each element is a term prefixed by `+` or `-`. When the trigger is called, each element in the list is written to the screen and added to or deleted from the database, depending on its prefix. A message is printed if an element to be added is already present, or if an element to be deleted is absent.

If the database contains the elements `f(a)` and `f(c)`, and the trigger is called with the argument `[+f(a),+f(b),-f(c),-f(d)]`, then on the initial burst of expansion

```
print_elements_and_modify_db([+f(a),+f(b),-f(c),-f(d)])
```

will be rewritten as the sequence

```
write(+f(a)) nl assert_or_retract(+f(a)) write(+f(b)) nl assert_or_retract(+f(b))
write(-f(c)) nl assert_or_retract(-f(c)) write(-f(d)) nl assert_or_retract(-f(d))
```

Each element of this sequence will then be processed in turn. The output will be as follows:

```
sc> print_elements_and_modify_db([+f(a),+f(b),-f(c),-f(d)]).
+f(a)
f(a) is already in the database.
+f(b)
-f(c)
-f(d)
f(d) is not in the database.
sc>
```

3.3 End-of-cycle processing

A distinguished trigger, `eoc`, is generated at the end of the processing cycle (i.e., when the trigger stack becomes empty). If there are rewrite rules with the trigger `eoc`, they are processed in the normal way. If new triggers are generated and processed, then the `eoc` trigger will be generated again at the end of the new cycle. If the `eoc` trigger generates no new tokens, the cycle terminates. (This may occur either if there are no `eoc`-driven rewrite rules, or if the rewrite rules have conditions which fail.) Figure 2 shows how end-of-cycle processing fits into the Sceptic execution model.

The `eoc` facility can be used, for example, to implement an agenda facility, where anything on the agenda is automatically handled at the end of the cycle. If items placed on the agenda match the condition `agenda(Item)`, and the processing of `execute(agenda(Item))` removes that match, the following rewrite rule will execute the agenda until there is nothing left on it, and then terminate.

```
eoc: agenda(Item) => execute(agenda(Item)).
```

Another use of `eoc` is to construct applications which automatically cycle continuously, rather than returning to the prompt after each burst of processing. The pursuer/evader case study is an example of such an application.

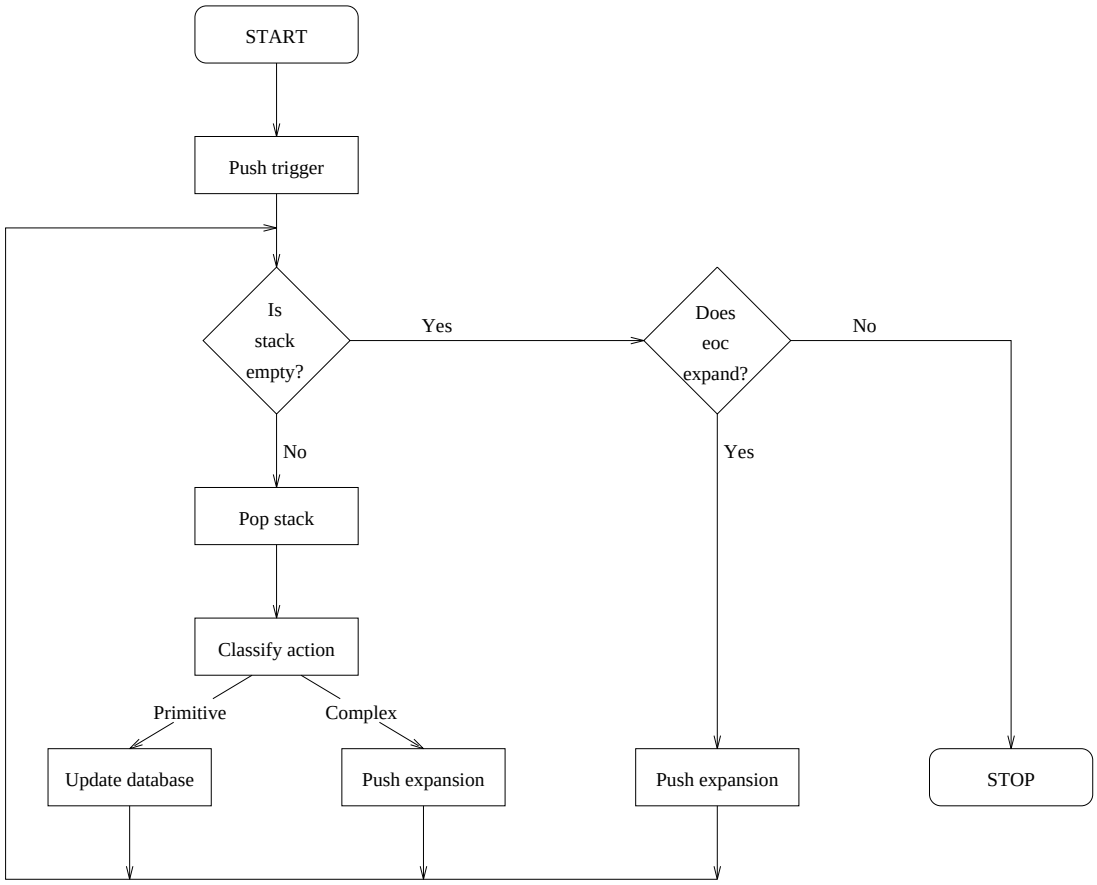

Figure 2: Execution of Sceptic actions (with `eoc`)

4 Summary of Primitives

This section briefly describes all standard primitive conditions and actions. The descriptions are generally very brief: further details may be obtained from the documentation of the underlying Prolog system. Note also that individual Prologs may provide further primitives over and above those listed here. For further details, see the relevant Prolog-specific documentation.

4.1 Primitive conditions

Conditions may be sub-divided into three sorts: testers, generators, and metapredicates. Testers simply test the database (or their arguments) and either succeed or fail. Generators generate data, either from the database or by transforming their arguments. Generating conditions still succeed or fail, but if they succeed they also instantiate variables. Furthermore, if a generator's arguments are fully instantiated when it is called, then that generator effectively acts as a tester. The distinction between generators and testers is thus functional, rather than absolute, but it can clarify data manipulation throughout processing. Metapredicates are those conditions which take other conditions as their arguments and manipulate those conditions.

4.1.1 State testers

`atom(+X)`: This succeeds if `X` is instantiated to an atom.

`atomic(+X)`: This succeeds if `X` is instantiated to an atom or a number.

`sc_eof(+X)`: This succeeds if `X` is the result returned on reading to the end of file.

`false`: This always fails.

`float(+X)`: This succeeds if `X` is instantiated to a floating point number.

`ground(+X)`: This succeeds if `X` is a ground term (i.e., if `X` is instantiated to a term containing no uninstantiated subterms).

`integer(+X)`: This succeeds if `X` is instantiated to an integer.

`nonvar(+X)`: This succeeds of `X` is not totally uninstantiated (i.e., if and only if `var(X)` fails).

`number(+X)`: This succeeds if `X` is instantiated to a number (i.e., an integer of a float).

`true`: This always succeeds.

`var(+X)`: This succeeds if `X` is uninstantiated (i.e., if and only if `nonvar(X)` fails).

4.1.2 Generators

`bagof(?X,:C,?B)`: This generates the list `B` of values of `X` that yield solutions to the predicate `C`. All variables in `C` apart from `X` are universally quantified. If there are no solutions of `C` then the predicate fails. `B` will contain duplicates if multiple solutions exist for a single value of `X`.

`compare(?Op,?X,?Y)`: `Op` is the result of comparing `X` and `Y`.

`current_op(?Precedence,?Type,?Name)`: `Name` is currently defined as an operator of type `Type` and precedence `Precedence`. (Note: the order of the arguments of this predicate depends on the Prolog in which Sceptic is implemented.)

`sc_exists(+F,-AF)`: This succeeds it `F` is the (relative) name of a file (from the current directory), and if so instantiates `AF` to the absolute name of that file.

`findall(?X,:C,?L)`: This generates the list `L` of values of `X` that yield solutions to the predicate `C`. All variables in `C` apart from `X` are existentially quantified. If there are no solutions of `C` then `L` is instantiated to the empty list.

`functor(?T,?F,?A)`: Given a term `T`, this decomposes `T` into its principal functor `F` and its arity `A`. Alternately, given a functor `F` and an arity `A` this instantiates `T` to a schematic term with principle functor `F` and arity `A`.

`get0(-C)`: This returns the ascii value of the next character in the current input, and updates the input pointer accordingly.

`get0(+S,-C)`: This returns the ascii value of the next character in the input stream `S`, and updates the input pointer accordingly.

`get(-C)`: This returns the ascii value of the next non-blank, non-layout character in the current input, and updates the input pointer accordingly.

`get(+S,-C)`: This returns the ascii value of the next non-blank, non-layout character in the input stream `S`, and updates the input pointer accordingly.

`is(?X,+Y)`: This evaluates the mathematical expression `Y` and assigns it to `X`. The syntax `X is Y` is also permissible for this predicate.

`length(+L,?N)`: This instantiates `N` to the length of the list `L`.

`sc_library_directory(?D)`: This instantiates `D` to the name of a library directory (see section 6).

`name(?A,?C)`: This interconverts atoms and character strings. If `A` is instantiated to an atom, this instantiates `C` to the list of ascii values of the characters in that atom. If `C` is a list of ascii values, this instantiates `A` to the atom corresponding to that ascii string.

`open(+F,+M,-S)`: Open the file with name `F` in mode `M` (`read/write/append`) and return the stream `S` associated with that file.

`read(?T)`: This instantiates `T` to the next term in the current input stream, and updates the input pointer such that it points to the next input term.

`read(+S,?T)`: This instantiates `T` to the next term in the input stream `S`, and updates the input pointer such that it points to the next input term.

`setof(?X,:C,?S)`: This generates the set `S` of values of `X` that yield solutions to the predicate `C`. All variables in `C` apart from `X` are universally quantified. If there are no solutions of `C` then the predicate fails. `S` will contain no duplicates.

`sort(+X,?Y)`: If `X` is a list then this instantiates `Y` to the corresponding sorted list.

4.1.3 Metapredicates

`cache(:X)`: This succeeds if and only if `X` succeeds. If possible, this metapredicate will either call `X` and store its results (until the next database modification) so that further calls to `X` will not require duplicating the processing required to prove `X`, or directly return the results of calling `X` as calculated on the previous call by directly matching against Sceptic's cache.

`call(:X)`: This succeeds if and only if `X` succeeds. This metapredicate simply directly calls its argument.

`not(:X)`: This succeeds if and only if `X` fails. No arguments in `X` will be instantiated by the call. Note that `not` provides a mechanism for testing for the existence of a solution to a predicate without instantiating any variables. The call:

```
not( not( X ) )
```

Will succeed once if and only if there is at least one solution to `X`.

4.2 Primitive actions

4.2.1 File loading

`consult(X)`: If `X` is an atom and the current directory contains a file whose name is `X` (or `X.sc`, or `X.pl`), then that file is loaded into Sceptic. If `X` is a list of atoms corresponding to a list of files, then each corresponding file in the list is consulted in turn.

If `X` or any element of `X` is of the form `library(Y)` then the library directories (see section 6) will be searched for the file `Y` and that file shall be consulted.

Following many Prologs, `[H|T]` when entered at the prompt is synonymous with `consult([H|T])`.

`compile(X)`: `compile/1` is analogous to `consult/1`, except that all predicates in the relevant files are compiled. Whilst compilation may lead to a substantial decrease in execution time, it does have its drawbacks. File compilation works by attempting to compile the file using the underlying Prolog's compilation facilities, and then translating any rewrite rules into the appropriate internal format. Thus compilation does not compile rewrite rules; compiled predicates cannot be debugged; and compilation may misinterpret directives. To avoid Prolog executing the Sceptic directives, particularly in files which contain directives to compile other files, the actions `sc_compile/1` and `sc_consult/1` may be used in place of `compile` and `consult`. Compilation is only recommended for experienced programmers dealing with well-tested implementations.

4.2.2 Input/Output

`close(S)`: Close the stream `S`.

`flush_output`: Flush the current output stream (i.e., send all buffered output to the terminal or file associated with the stream).

`flush_output(S)`: Flush the output stream `S` (i.e., send all buffered output to the terminal or file associated with the stream).

`nl`: Output a newline character on the current output stream.

`nl(S)`: Output a newline character on the output stream `S`.

`put(C)`: Output the character with ascii code `C` to the current stream.

`put(S,C)`: Output the character with ascii code `C` to the stream `S`.

`see(F)`: Set the current input stream to the file `F`.

`seen`: Close the file associated with the current input stream and read future input from the terminal.

`skip(N)`: Skip all characters in the input up to and including that whose ascii code is `N`.

`skip(S,N)`: Skip all characters in the input stream `S` up to and including that whose ascii code is `N`.

`tab(N)`: Output `N` spaces to the current stream.

`tab(S,N)`: Output `N` spaces to the stream `S`.

`tell(F)`: Set the current output stream to the file `F`.

`told`: Close the file associated with the current output stream and send future output to the terminal.

`write(T)`: Write the term `T` to the current output.

`write(S,T)`: Write the term `T` to the stream `S`.

4.2.3 Database modification

`assert(X)`: Add an instance of the term `X` to the Sceptic database.

`retract(X)`: Remove an instance of the term `X` from the Sceptic database.

`set_assert(X)`: Ensure that there is at least one term of the form `X` in the Sceptic database.

`retractall(X)`: Remove all terms with head `X` from the Sceptic database.

4.2.4 Debugging/Tracing execution

`debug`: Enter debug mode (see Section 8).

`nodebug`: Exit debug mode (see Section 8).

`set_tracking_level(Phase, Level)`: Set the tracking level used when in debug mode. (For further details see Section 8.)

`spy`: Display all spy points (see Section 8).

`spy(X)`: Place a spypoint on a predicate/trigger (see Section 8).

`nospy(X)`: Remove a spypoint from a predicate/trigger (see Section 8).

`verbose`: Enter verbose mode (see Section 2.6).

`noverbose`: Exit verbose mode (see Section 2.6).

4.2.5 Miscellaneous

`garbage_collect`: Force a Prolog garbage collection.

`halt`: Exit Sceptic (returning to the operating system).

`listing(X)`: List all predicates and rewrite rules in the Sceptic database which match the term or functor/arity specification `X`.

`listing`: List all predicates and rewrite rules in the Sceptic database.

`op(Precedence, Type, Name)`: Declare the operator `Name` of type `Type` with precedence `Precedence`. Operators may be used as in standard Prolog. They are essentially a form of syntactic sugar, allowing some variation in the usual Prolog syntax. Any Prolog textbook should provide a full discussion of operators (e.g., Clocksin & Mellish, 1981).

`pl(X)`: Call the underlying Prolog predicate `X`. This primitive provides full access to the underlying Prolog system. It is provided mainly because we have found it useful in developing Sceptic, but its use in actual applications is strongly discouraged. Predicates like `asserta/1` and `assertz/1` are only accessible as actions by using `pl/1`.

`repeat(T, N)`: Call the trigger `T` `N` times in succession.

`statistics`: Print statistics concerning space and time usage.

5 Modelling in Sceptic

5.1 Control-flow Sceptic

As mentioned above, the two varieties of statement available in Sceptic allow a process to be expressed in terms of the conditional rewrite rules which specify the control together with the predicate definitions which specify the conditions used in that control. We refer to this as the "control-flow" style of Sceptic specification.

Programming in the control-flow style is most easily performed in a top-down manner, specifying the overall process in terms of its immediate subprocesses. Thus, in the case of syllogistic reasoning, the process of generating a conclusion given a pair or premises can be broken down into the subprocesses of constructing a model of the premises and of deriving an informative conclusion from that model. Suppose, for the sake of example, that we do not wish to commit to the details of the processing involved in constructing a model, but we do wish to commit to further details of deriving a conclusion. The construct-model process would then be treated as a condition which generates data (a model) which the derive-conclusion process uses. We would thus have the following rewrite rule:

```
syllogistic_reasoning(Premise1,Premise2):
    construct_model(Premise1, Premise2, Model)
 => derive_conclusion(Premise1, Premise2, Model).
```

The construct-model process may be specified in declarative terms in standard Prolog (and perhaps even given via table look-up), thus making no commitment to the algorithms employed in constructing a model. Further rewrite rules may be given for deriving a conclusion, specifying the details of the algorithms posited at whichever level of granularity is desired. In this way Sceptic allows a clear division between level 1, or mathematical, statements of a subprocess (simply specifying input/output relations in declarative/logical terms) and level 2, or algorithmic, statements of a subprocess.

5.2 Data-flow Sceptic

A potential problem with the control-flow style is that when processing a trigger with multiple expansions, the execution of one expansion may modify the database, influencing the execution of subsequent expansions. Were such dependencies to be intended they would obscure the clarity of a specification. They should therefore be avoided.

A solution to this problem is to focus on data-flow, rather than control-flow, and synchronise database modifications. One option is to restrict actions to database modifications, and allow only a single trigger which is called cyclicly. On each cycle, all condition evaluation will be completed before any database modification is effected. The limitation is not as severe as it first seems, and the style is in fact very conducive to models which can be phrased in terms of a set of data elements (or "micro-states"), each changing according to specifiable rules on every cycle.

For example, in the motive processing case study a rule to deactivate those motives whose goal has been achieved might be as follows:

```
eoc:
    motive(Id, Goal, active),
    goal_is_achieved(Goal)
 => retract(motive(Id, Goal, active)),
    assert(motive(Id, Goal, achieved)).
```

Here, Sceptic's `eoc` trigger is used to automatically generate the cyclic behaviour.

In the above case, one rule would be required for each type of transition. An alternative is to specify the model in terms of schemata specifying the micro-state transitions, together with a Sceptic engine to process the schemata. Given schemata of the form `transform(Old, New, Condition)`, which are intended to be read as "the element `Old` should be replaced by the element `New` if `Condition` is satisfied", the heart of such an engine might consist of a single rewrite rule:

```
eoc:
    match(Old),
    transform(Old, New, Condition),
    call(Condition)
 => retract(Old),
    assert(New).
```

In general, a data-flow specification of a process will consist of:

1. declarations and initial values of the micro-state elements;
2. rules for transforming micro-state elements (possibly abstracted as above); and
3. definitions of the predicates which govern micro-state transitions.

5.3 Pitfalls for the unwary

The following points have caused problems for several people when learning to use Sceptic. They should be borne in mind when starting to write Sceptic programs.

Actions cannot instantiate variables: Variables which are uninstantiated when an action is called will not be bound by that call. For example in the rewrite rule:

```
process(X): true => process1(X,Y), process2(Y).
```

the variable `Y` will be uninstantiated when `process2` is called. This rule should be rewritten as:

```
process(X): generate(X,Y) => process2(Y).
```

and can be read as to `process X`, if `Y` can be generated from `X` then `process2 Y`. This behaviour is a consequence of the universal quantification of variables within rewrite rules. An undesirable further consequence is that `process1` must be written as a condition (`generate/2`).

Declaration of dynamic predicates: Dynamic predicates, that is, predicates which are asserted or retracted during Sceptic's processing, must be declared as dynamic. In general dynamic predicates will also need to be initialised. Note that the dynamic declaration must precede the initialisation.

Multiple use of Predicates/Triggers: All predicates with the same head must be loaded from a single file. Similarly, all rewrite rules with the same trigger name must be loaded from a single file. Loading a second file containing predicates/rewrite rules with already used heads/trigger names will result in the original predicates/rewrite rules being overwritten. This is analogous to the behaviour of 'reconsult' in Prolog.

6 Library Routines

Version 4 of Sceptic includes an interface to a number of general purpose library files, containing some standard predicate and trigger definitions. These predicates and triggers have been found to be useful in a number of modelling applications, and we strongly recommend, in the interests of standardisation, that they be used where possible. The following libraries currently exist:

`control.sc`	some basic Sceptic control structures
`counters.sc`	routines for maintaining counters and generating unique identifiers
`list_processing.sc`	standard list processing predicates (such as `member/2`, `delete/3` and `append/3`)
`maths.sc`	miscellaneous mathematical functions including a pseudo-random number generator
`parameters.sc`	routines for setting and checking parameters and flags
`read.sc`	routines for reading fancy input
`tms.sc`	truth maintenance system, a constrained memory structure

We view these libraries as an integrated part of the Sceptic environment in that they are intended to facilitate the development of computational models. Of particular interest to psychologists will be the simple routines for maintaining parameters (corresponding, perhaps, to individual differences), the facilities for generating both uniformly distributed and normally distributed pseudo-random numbers (corresponding, perhaps, to noise in some module which processes continuous inputs), and the constrained memory.

The primitive actions `compile/1` or `consult/1` may be used to incorporate a library by giving the argument as `library(+File)`. Thus, if an application requires the standard list processing predicates, then these may be incorporated from the Sceptic prompt as follows:

```
sc> consult(library(list_processing)).
{compiling /usr/local/lib/sceptic/list_processing.sc...}
{/usr/local/lib/sceptic/list_processing.sc compiled, 950 msec 3730 bytes}
sc>
```

Alternately, the library could be included by using `consult(library(+File))` as a directive in the main application file. That is, if the application file contains the line

```
:- consult(library([list_processing,maths])).
```

then when the application file is consulted the list processing and maths libraries will automatically be included. The argument of `library/1` may be a single file or a list of files.

Sceptic locates library files by searching its search path. The search path consists of the current directory, followed by the user's home directory, followed by any directories declared as library directories. A directory `X` may be declared as a library directory by including the following predicate in your application file:

```
sc_library_directory(X).
```

Asking `sc_library_directory(X)` at Sceptic's prompt will return all library directories in the order in which they are searched.

6.1 Library: control.sc

Triggers

`case(L)`:
: Fires the first action from the list of condition action pairs where the condition is true. `L` takes the form `[ condition_1 - action_1, condition_2 - action_2, ... , otherwise - default_action ]`, where `condition_n` is a Sceptic condition, and `action_n` is a Sceptic action.

`fire_list(L)`:
`L` is interpreted as a list of triggers. This fires each trigger in `L` in sequence.

6.2 Library: counters.sc

A counter is an atom with an associated numeric value. The routines in this library allow counters to be declared and their values to be incremented and decremented.

Triggers

`create_counter(Counter,Value)`:
Creates a counter with the specified initial value.

`delete_counter(Counter)`:
Removes the specified counter from the database.

`increment_counter(Counter)`:
Increments the specified counter by 1.

`decrement_counter(Counter)`:
Decrements the specified counter by 1.

`list_counters`:
Displays the current counters and their values.

`set_counter(Counter, Value)`:
Resets the counter to the specified value.

Conditions

`counter(?Counter, ?Value)`:
`Value` is the current value of `Counter`

`current_reference(+Counter, +Prefix, ?Reference)`:
The current value of `Counter` is appended to `Prefix` to form the reference symbol `Reference`. `Counter` is not updated.

`generate_reference(+Counter, +Prefix, ?Reference)`:
The current value of `Counter` is appended to `Prefix` to form the reference symbol `Reference`. `Counter` is updated.

6.3 Library: list_processing.sc

Triggers

`write_bracketed_list(L)`:
`L` may be an atom, list, or list of lists. The contents, minus any empty lists, will be printed in round brackets on the current output stream.

`write_bracketed_list(Stream, L)`:
`L` may be an atom, list, or list of lists. The contents, minus any empty lists, will be printed in round brackets to `Stream`.

Conditions

`append(?A, ?B, ?C)`:
Appends lists `A` and `B` to form list `C`.

`append_atoms(+A, +B, +C)`:
Appends atoms `A` and `B` to form atom `C`.

`append_atom_list(+LA, -A)`:
Appends all the atoms in the list `LA` producing the atom `A`.

`append_lists(+LL, -L)`:
Appends all the lists contained in the list `LL`, producing the list `L`.

`convert_comma_term_to_list(+C, ?L)`:
The contents of the comma separated list of terms `C` is the same as the list `L`.

`convert_list_to_comma_term(+L, ?C)`:
: The contents of list `L` is the same as the list of comma separated terms `C`.

`delete(?X, ?B, ?C)`:
: All elements of list `B` are in list `C` except element `X`.

`last_element(?L, ?X)`:
: `X` is the last element of list `L`.

`member(?X, ?L)`:
: `X` is a member of the list `L`.

`position_in_list(?X, ?L, ?N)`:
: The element `X` is at position `N` from the beginning of list `L`.

`recursively_remove_empty_lists(+InList, -OutList)`:
: This succeeds if `OutList` is the result of removing any empty lists, or lists of empty lists, or lists of lists of empty lists, or ..., from in `Inlist`.

`remove_duplicates(+L1, -L2)`:
: Duplicate elements of list `L1` are removed to produce list `L2`.

`replace_all_occurrences(?L1, ?X1, ?X2, ?L2)`:
: List `L2` is the same as list `L1` except that all occurrences of element `X1` are replaced by element `X2`.

`replace_first_occurrence(?L1, ?X1, ?X2, ?L2)`:
: List `L2` is the same as list `L1` except that the first occurrence of element `X1` is replaced by element `X2`. Fails if `X1` does not appear in `L1`.

`reverse(+LA, -AL)`:
: The elements of list `LA` are in the reverse order to the elements of list `AL`.

`set_equal(+A, +B)`:
: This succeeds if and only `B` is some permutation of `A`. That is, if `A` and `B` are interpreted as multisets, then they are equal.

`sublist(+A, +B)`:
: This succeeds if `Sub` is a sublist of `List`. That is, if `List` can be formed from `Sub` by adding extra elements to `Sub` without altering the order of the original elements in `Sub`.

`subset(+Sub, +List)`:
: This succeeds if each and every member of `Sub` is a member of `List`. If an element occurs several times in `Sub` then it must occur at least as many times in `List`. That is, if `Sub` and `List` are interpreted as multisets, then `Sub` is a subset of `List`.

6.4 Library: maths.sc

Triggers

`randomise`:
: Generate and set a seed for pseudo-random number generation. This seed is based on the current date and time.

`set_random_seed(A, B, C)`:
: Set the seed for pseudo-random number generation to the triple ⟨A,B,C⟩.

Conditions

`absolute_value(+A, -B)`:
: Absolute value: $B = |A|$

`acos(+A, -B)`:
: Inverse cosine: $B = cos^{-1}(A)$, where `B` is in the range 0 to π radians

`arithmetic_mean(+L, -M)`:
: The arithmetic mean of the list of numbers `L` is `M`.

`asin(+A, -B)`:
: Inverse sine: $B = sin^{-1}(A)$, where `B` is in the range $\pm\frac{\pi}{2}$ radians

`atan(+A, -B)`:
: Inverse tangent: $B = tan^{-1}(A)$, where `B` is in the range $\pm\frac{\pi}{2}$ radians

`cos(+A, -B)`:
Cosine: `B` = cos(`A`), where `A` is given in radians.

`exp(+X, -Y)`:
Exponential: `Y` = $e^{\texttt{X}}$

`factorial(+A, -B)`:
Factorial: `Y` = `X`! = (`X` × (`X` − 1) × (`X` − 2) × ... × 1)

`geometric_mean(+L, -G)`:
The geometric mean of the list of numbers `L` is `G`.

`log(+X, -Y)`:
Natural logarithm: `Y` = ln(`X`)

`match_random_seed(-A, -B, -C)`:
Returns the current three integer random seed.

`pi(-P)`:
Returns the value of pi to 20 significant figures.

`product_list(+L, -P)`:
The product of the list of numbers `L` is `P`.

`power(+A, +B, -C)`:
Power: `C` = $\texttt{A}^{\texttt{B}}$

`random(-N)`:
Return a random number `N` less than 1 but not less than 0.

`random(?D,-N)`:
Return a random number `N` from a specified distribution `D`. `D` may be either `normal(M,V)`—a normal distribution with mean `M` and variance `V`—or `uniform`—the flat distribution generated by `random/1`.

`sigmoid(+X, -Y)`:
`Y` = $\frac{1}{1+e^{-\texttt{X}}}$

`sin(+A, -B)`:
Sine: `B` = sin(`A`), where `A` is given in radians.

`sqrt(+A, -B)`:
Square root: `B` = $\sqrt{\texttt{A}}$

`sum_list(+L, -S)`:
The sum of the list of numbers `L` is `S`.

`tan(+A, -B)`:
Tangent: `B` = tan(`A`), where `A` is given in radians.

There are in fact two maths library files, either of which may be explicitly loaded. The functions listed below are defined by approximations in the library `maths_approx.sc` (for SICStus version 0.7 Prolog and Poplog), and by built in functions in `maths_built_in.sc` (for SICStus version 2.1 and Quintus Prolog). All other mathematical functions are defined by common definitions. If Sceptic has been correctly built on your system these differences should be transparent and all functions should be available within the `maths.sc` library.

Approximated functions:

`exp(+X, -Y)`	`log(+X, -Y)`	`power(+A, +B, -C)`
`cos(+A, -B)`	`sin(+A, -B)`	`tan(+A, -B)`
`acos(+A, -B)`	`asin(+A, -B)`	`atan(+A, -B)`
`absolute_value(+A, -B)`	`sqrt(+A, -B)`	

6.5 Library: parameters.sc

Triggers

`list_flags`:
: Show the status of all the flags.

`list_parameters`:
: List all parameters and their values.

`set_flag(F)`:
: Set the flag `F` (to true).

`set_parameter(P, V)`:
: Set the parameter `P` to `V`.

`toggle_flag(F)`:
: Toggle the status of the flag `F`.

`unset_flag(F)`:
: Unset the flag `F` (i.e., set it to false).

Conditions

`flag_is_set(?F)`:
: Succeeds for a flag `F` which is set.

`parameter(?P, ?V)`:
: Succeeds for each current parameter `P` and value `V`.

6.6 Library: read.sc

Conditions

`read_word_sequence(-Words)`:
: `Words` is a list of words read from the current input stream. The input should contain no punctuation characters (except apostrophes, which are treated as letters) and be terminated by a newline.

`read_word_sequence(+Stream, -Words)`:
: `Words` is a list of words read from the input stream `Stream`. The input should contain no punctuation characters (except apostrophes, which are treated as letters) and be terminated by a newline.

`read_integer(-Int)`:
: Reads an integer value `Int` from the current input stream.

`read_integer(+Stream, -Int)`:
: Reads an integer value `Int` from the input stream `Stream`.

`read_to_eoln(-List)`:
: Read from the current input stream to the end of the next line and return the list of characters read as `List`.

`read_to_eoln(+Stream, -List)`:
: Read from the input stream `Stream` to the end of the next line and return the list of characters read as `List`.

`skip_to_eoln`:
: Skip over (i.e., ignore) all input from the current input stream until to the end of the next line.

`skip_to_eoln(+Stream)`:
: Skip over (i.e., ignore) all input from stream `Stream` until to the end of the next line.

6.7 Library: tms.sc

The truth maintenance system provides a memory structure in which items are kept in a set, and over which integrity (truth) constraints are automatically applied.

Truth is maintained over the data set with respect to rules of the form:

```
Antecedents ->> Consequents.
```

`Antecedents` and `Consequents` are both lists which may use the following syntax, specifically `not/1` may not be nested, and calls to Prolog predicates must be wrapped in `call/1`.

Antecedents: [a, b, c(X,Y), not(Y), not([d,e,X]), call(predicate)]
Consequents: [d, e, not(f(X))]

The tms does not analyse the rules for consistency, and even with consistent rule sets non-deterministic situations can arise.

Three example rules:

```
[ loc(X,Y), food(A,B), call(reach(X,Y,A,B)) ] ->> [ target(A,B) ]. % target food within reach

[ number(A), number(B), call(A > B) ] ->> [ not(number(B)) ] % only remember the highest number

[ number(A), call(colour(A, C)) ] ->> [ colour(C) ] % storing a number will store it's colour
```

Conditions

`match(?Atom)`:
: Match `Atom` with an atom in the tms memory.

`tms_support(?Atom, -Support)`:
: Return one of the logical supports for `Atom`, an atom in the tms memory.

Actions

`la(+Atom)`:
: Logical Assert, add `atom` and propagate logical consequences to the tms memory.

`lr(+Atom)`:
: Logical Retract, retract `Atom` from, and propagate logical consequences to, the tms memory.

`assert_tms_rule(+Ants ->> Cons)`:
: Assert the rule `Ants ->> Cons` to the tms rule set, and propagate logical consequences to the tms memory.

`retract_tms_rule(+Ants ->> Cons)`:
: Retract the rule `Ants ->> Cons` from the tms rule set, and propagate logical consequences to the tms memory.

`tms_clear`:
: Clear the tms memory.

`tms_check`:
: Check the rules are consistent with the tms memory, making modifications if necessary.

`tms_facts`:
: List the facts in the tms memory.

`tms_rules`:
: List the rules used by the tms.

`tms_support(+Atom)`:
: List the supporting rules for the tms memory element `Atom`.

7 Advanced Modelling Considerations

7.1 Modelling in the Data-flow style

7.1.1 Local and global computation

Data-flow Sceptic emphasises Sceptic's database, and "local" computation is central to the processing. By this we mean that on each cycle each data element is transformed independently of transformations on other data elements. One consequence of this is that certain types of process are difficult to model in the data-flow style.

Consider, for example, the allocation of resources. This problem occurs in the nursemaid scenario of the motive processing case study. The problem is to allocate one resource each to several processes, where the number of resources is restricted to less than the number of processes. Thus there might be four processes competing for two resources. In the control-flow style one could simply randomly select a process and allocate a resource to it, and continue until no unallocated resources remain. In data-flow Sceptic one must essentially generate a mapping for resources to processes and allocate according to that mapping. The problem is that the generation of the mapping is a global process—it requires knowledge of all processes and all resources—yet data-flow rules work on an instance-by-instance basis. This solution may be coded as follows:

```
eoc:
    match(resource(R, unallocated)),
    generate_resource_process_mapping(Map),
    member(R-P, Map)
 => retract(resource(R, unallocated)),
    assert(resource(R, allocated_to(P))).
```

The idea here is that database elements of the form `resource/2` are being transformed, and this is done by generating a mapping of resources to processes (and this may be the difficult bit). This mapping is a list whose members are of the form `R-P` where `R` is the identifier of a resource and `P` is the identifier of the process which is to be allocated to it. The rule transforms elements of the form `resource(R, unallocated)` to elements of the form `resource(R, allocated_to(P))`. Note that the mapping generated must be deterministic (i.e., with a fixed database, each call to `generate_resource_process_mapping/1` must generate the same mapping) as the predicate will be called for each unallocated resource.

7.1.2 Caching the results of global computation

The above example is highly inefficient as the same mapping must be generated for each resource. Sceptic includes a caching mechanism which is designed purely for efficiency purposes and which is highly appropriate for global computation within the data-flow style. A call to any predicate (presumably a generating predicate) `X` can be cached by replacing it with `cache(X)`. The effect of this is to either

a. call `X` directly if `X` has not been called since the last database modification, and record all solutions to `X`; or

b. retrieve the previous solutions of `X` if the results of `X` are recorded in the cache.

Thus the above rule might be modified to:

```
eoc:
    match(resource(R, unallocated)),
    cache(generate_resource_process_mapping(Map)),
    member(R-P, Map)
 => retract(resource(R, unallocated)),
    assert(resource(R, allocated_to(P))).
```

This will improve the efficiency as the resource-process mapping will be generated only once (on each cycle) and then saved so that further calls simply look up the results of the first call.

At present the caching mechanism is relatively primitive, and Sceptic only caches calls to predicates where all variables in the call are uninstantiated. Attempting caching of predicates which do not satisfy this condition will simply call the predicate directly, and will not alter the behaviour or improve the efficiency of the program.

7.2 Caveats

7.2.1 Efficiency

The current implementation was designed with some concern for efficiency, but other considerations were given priority. There are clearly areas where efficiency could be improved. For example, coding decision trees requires rules of the form:

```
trigger1(X):
    test(X)
 => trigger2a.
trigger1(X):
    not(test(X))
 => trigger2b.
```

In executing these rules Sceptic currently evaluates the left-hand condition separately for each rule. Clearly this condition really need only be evaluated once. The caching facility introduced in this version provides a mechanism for the programmer to specify this, but such efficiency issues should not be of concern to the programmer, and there remain other clear avenues for further efficiency gains. These may be addressed in future releases.

7.2.2 Why not just use Prolog?

Sceptic is written in Prolog and thus does nothing which cannot be coded directly in Prolog. The potential interest in the Sceptic notation and control mechanism arises from the two styles of specification (control-flow and data-flow) which it supports, and the extent to which these naturally expresses a range of applications. It is our belief that the control-flow style encourages the clear separation of declarative and procedural aspects of processing, and that the data-flow style provides a simple declarative framework for expressing processes. The choice of control-flow or data-flow for any particular application will depend to some extent on the application (data-flow is best suited to cyclic processes that can be treated in term of micro-state transitions), but also on the intended use of the application (the procedural process/declarative sub-process possibilities of control-flow make it particularly suitable for distinguishing those processes for which a theoretical commitment to the algorithmic nature of sub-processes is desired from those processes for which only a commitment to the input/output relations is desired).

7.2.3 Is Sceptic a specification language?

One postulated use of Sceptic was as an executable specification language. However, writing Sceptic applications still has a large element of programming (albeit in a very high-level language) and, as with Prolog, Sceptic lacks the type declaration and checking facilities which are the minimal theorem proving capabilities required of a specification language. In addition, although Sceptic has a very succinct syntax, the declarative semantics remain to be formally specified. However, Sceptic appears to provide a harness or technique for coding a class of data-driven applications in a way which naturally expresses some aspects of those applications. The clarity gained may be a helpful step towards specification. We have already found cases where a set of Sceptic rewrite rules has provided a means of discussing alternative behaviours in a way that (larger) raw Prolog programs or (vaguer) English descriptions have not.

8 Debugging[2]

This section describes the Sceptic-oriented debugger. A debugger for Sceptic rewrite rules was first introduced to Sceptic version 3 by Saki Hajnal. Version 4 includes extensions to this debugger allowing conditions to also be debugged.

8.1 Starting the Debugger

The Sceptic interpreter has a separate execution mode for debugging code. The debugging related commands (i.e., primitive actions) are:

```
debug.
nodebug.
spy.
spy(Term).
nospy(Term).
```

The `debug` and `nodebug` commands switch Sceptic to and from it's debug mode. Programs run in debug mode require significantly more memory and execution time than when run in nodebug mode. Sceptic starts by default in nodebug mode, but this could be changed by adding the `debug` directive to your `.scepticrc` file.

The `spy` and `nospy` commands take as their argument either a functor, in which case all predicates/rewrite rules whose trigger is that functor (and with any arity) are spied, a functor/arity specification, in which case only those predicates/rewrite rules whose trigger is that functor with that arity are spied, or a structure, in which case only those predicates/rewrite rules whose trigger matches that structure are spied. Using `spy` with no arguments lists all current spy points.

A spied predicate or rewrite rule will only be caught if Sceptic is in debug mode. Using `spy/1` places Sceptic in debug mode automatically. Removing all spy points will cause Sceptic to automatically revert to nodebug mode.

8.2 The Sceptic Debugger

The debugger has three phases: condition (con), recognise (rec) and action (act). The condition phase is used when debugging conditions. By placing spypoints and using moving/displaying commands you can trace predicate evaluation at various granularities as in Prolog. The act phase is used when debugging the act cycle of action execution. By placing spypoints and using moving/displaying and tracking levels you can trace trigger expansion at various granularities, remaining in the act phase. Also at any trigger at which you have paused, you can choose to debug the recognise phase, and trace the evaluation of rewrite rule conditions, again at various granularities.

Each phase has various commands for moving to a new pause point or displaying information. At any pause point, typing '?' or 'h' to the debugging prompt will show the available commands. Each phase also has an (independently set) tracking level, which controls how much information will be displayed *between* pause points.

The system will stop and prompt at the first occurrence of a spied predicate/trigger. When in the rec or act phase, the prompt is of the following form:

```
(Phase|Level|Cmd:Name) ?
```

where:

[2] Much of this section is taken with little modification from Hajnal *et al.* (1989).

`Phase` is `con`, `rec` or `act`
`Level` is the current tracking level for this phase
`Cmd` is the last command
`Name` is the full name of that command

When in the con phase, the prompt is of the following form:

```
Point: Predicate? [Cmd]
```

where:

`Point` is either `Call`, `Exit`, `Retry`, or `Fail`
`Predicate` is the current instantiation of the spied predicate
`Cmd` is the last command, and default next command

You may input commands (see below) to the prompt. Note that debugging commands are terminated by RETURN and do not require a full stop. Typing RETURN on its own will repeat the last command, which is shown in the prompt. With the initial commands as described below, this may not be the last command typed by the user. This aspect is subject to review.

8.3 Initial command settings

At the start of each condition phase debugging cycle (i.e., when you have input a condition to the prompt), the initial command is 'leap' (`l`). At the start of each act phase debugging cycle (i.e., when you have input an action to the prompt), the initial command is 'leap' (`l`). At the start of each recognise phase debugging cycle (i.e., when you have chosen to debug the recognise phase by issuing the command `dr` while debugging a rewrite rule), the initial command is 'new rewrite rule' (`p`).

8.4 Condition phase commands

When Sceptic is in debug mode and a condition is entered at the prompt that condition will be evaluated with debugging enabled. This allows debugging of conditions in a similar way to that in traditional Prolog. When debugging conditions the move commands can trace the predicates being called, becoming instantiated, retrying (backtracking) and failing, indicated by `Call`, `Exit`, `Retry` and `Fail` in the debugging output; the messages are indented to correspond with the level of the predicate call. Detailed debugging of conditions like this is only available when a condition is called from the prompt. Debugging of condition evaluation during the recognise phase of trigger expansion will, however, produce some terse output. The debugger will automatically skip over compiled predicates.

8.4.1 Move

Once you have stopped at a spypoint, the following commands are available to move to a new point:

`n`	Nowait	Continue to the prompt
`c`	Creep	Step to next predicate
`s`	Skip	Skip over the full evaluation of this predicate
`l`	Leap	Leap to next spied predicate
`f`	Fail	Force evaluation of this predicate to fail

At present compiled predicates cannot be debugged. The debugger will automatically skip over such predicates.

8.4.2 Add/remove spypoints

Once you have stopped at a predicate, you can add a spypoint at it or remove a spypoint from it:

`+`	Add Spypoint	Place a spypoint on the current predicate
`-`	Remove Spypoint	Remove a spypoint from the current predicate

8.4.3 Display

`dp`	Display Predicate	Lists the current predicate
`di`	Display Instantiated Predicate	Lists with current variable bindings

8.4.4 Debugging Conditions Example

```
sc> spy(member).
Adding SCEPTIC spy point on condition: member/2
sc> member(b,[a,b]).
Call: member(b, [a, b])? [l] c
 Call: member(b, [b])? [c]
 Exit: member(b, [b])? [c]
Exit: member(b, [a, b])? [c]
member(b, [a, b])
Retry: member(b, [a, b])? [c]
 Retry: member(b, [b])? [c]
  Call: member(b, [])? [c]
  Fail: member(b, [])? [c]
 Fail: member(b, [b])? [c]
Fail: member(b, [a, b])? [c]
sc>
```

8.5 Act phase commands

8.5.1 Move

Once you have stopped at a spypoint, the following commands are available to move to a new point:

`n`	Nowait	Continue to the prompt (with optional tracking)
`c`	Creep	Step to next trigger or predicate
`s`	Skip	Skip over the full expansion of this trigger
`l`	Leap	Leap to next spied trigger
`sr`	Skip Recognise	Skip over recognise phase for this trigger (so you can see the actions generated and still skip afterwards)
`dr`	Debug Recognise	Debug recognise phase for this trigger

8.5.2 Add/remove spypoints

Once you have stopped at a trigger, you can add a spypoint at it or remove a spypoint from it:

`+`	Add Spypoint	Place a spypoint on the current trigger
`-`	Remove Spypoint	Remove a spypoint from the current trigger

8.5.3 Display

The source of information display in the act phase is a stack structure of pending triggers and actions (predicates), with information about which triggers generated them. The following display commands are available.

`f`	Full Stack	Display the full trigger/action stack
`w`	Waiting	Display the pending triggers/predicates only
`t`	Current	Display the current trigger or predicate (This is displayed automatically before pausing. This command is to recall it if other output has intervened.)

The stack is displayed with indentation to reflect the tree structure of the expansion, with the root of the tree at the bottom and most indented. In the full display, triggers which have already run their

recognise cycle and generated others, and so are no longer waiting, are shown in angled brackets. Separate instantiations in one generation are separated by `-`.

Consider, for example, the rewrite rules:

```
t1: c1(X) => t2(X), a1(X).
t2(X): true => a2(X).
```

where `a1(X)` and `a2(X)` are also triggers, and the database:

```
c1(fred).
c1(jo).
```

If we have a spypoint on `a2(_)`, and the user has entered the trigger `t1`, the structure displayed by the full display is

```
Full Stack :
a2(fred)
   < t2(fred) >
   a1(fred)
   -
   t2(jo)
   a1(jo)
   < t1 >
     < user >
```

i.e.,
`a2(fred)` is the current trigger, which was generated from
`< t2(fred) >` which is no longer pending
`a1(fred)` belongs to the same instantiation as `t2(fred)`
`t2(jo)` and `a1(jo)` are another instantiation
Both these instantiations were generated from
`< t1 >` which is no longer pending, which was generated from the user.

The corresponding waiting display is:

```
Waiting :
a2(fred)
   a1(fred)
   -
   t2(jo)
   a1(jo)
```

8.5.4 Track

The command to change the tracking level is just to type the level number. The available tracking levels are:

- `0` No tracking
- `1` Track spied triggers only
- `2` Track all triggers

The tracking level determines which *additional* points will be displayed between pause points. So, for example, if you continually 'leap', i.e., pause at all spied triggers, tracking level 1 will have no effect. However, if you use the 'nowait' command `n` with this level, you will see all spied triggers passed between then and the prompt.

The tracking level for the recognise and act phases can also be set from the Sceptic prompt via the `set_tracking_level/2` action. This takes a phase (`rec` or `act`) and a tracking level (0, 1, 2 or 3) as its arguments. This allows executing to be traced without stopping at any points. Note however that tracking will only occur if debugging is also on.

8.6 Recognise phase commands

You enter the recognise phase by typing `dr` at a paused trigger, to debug the recognise phase. In this phase, there are no spypoints as such. There are various distinguished points in the cycle which are available as pause points. These points are:

New Rewrite Rule	Start of a new rewrite rule for current trigger
Instantiation	Complete instantiation of current rewrite rule
New Condition	Start of a new condition of current rewrite rule
Success	Success of current condition
Failure	Failure of current condition
Retry	Retry of previous condition

On starting to debug the recognise phase the first pause is at the start of the first rewrite rule. The different commands and levels pause or display different subsets of these points.

8.6.1 Move

Once you have stopped at a point, the following commands are available to move to a new point:

`n`	Nowait	Continue to the end of the rec phase (with optional tracking)
`p`	New Rewrite Rule	Stop at the start of the next rewrite rule, or end of rec phase
`i`	Instantiation	Stop at the next instantiation, or new rewrite rule, or end
`s`	Success	Stop at the next condition success, instantiation, rewrite rule or end
`c`	Creep	Stop at next point

8.6.2 Display

There are no separate display commands at present.

8.6.3 Track

The command to change the tracking level is just to type the level number. The available tracking levels are:

- `0` No tracking
- `1` Track rewrite rules and instantiations
- `2` Track rewrite rules, instantiations and succeeding conditions
- `3` Track all points

The tracking level determines which additional points will be displayed between pause points. So, for example, if you continually use command `i` i.e., pause at all rewrite rules and instantiations, tracking level 1 will have no effect. However, if you use the 'new rewrite rule' command `p` with this level, you will see all instantiations passed between then and the next rewrite rule.

9 References

Clocksin, W. & Mellish, C. (1981). *Programming in Prolog*. New York: Springer-Verlag.

Fox, J. (1992). Techniques for developing distributed decision systems. Dilemma workshop #1.

Hajnal, S., Fox, J. & Krause, P. (1989). Sceptic user manual: version 3.0. Technical Report. Advanced Computation Laboratory, Imperial Cancer Research Fund, London.

O'Neill, M. (1992). The STOP demonstrator. Technical Report, Advanced Computation Laboratory, Imperial Cancer Research Fund, London.

Walker, N. S. (1991). *Biomedical Image Interpretation*. Ph.D. Thesis, Department of Computer Science, Queen Mary and Westfield College, London.

Zicha, D. & Fox, J. (1991). Symbolic Simulation of Complex Biological Processes. Technical Report, Advanced Computation Laboratory, Imperial Cancer Research Fund, London.

A Top Level Command Summary

`consult/1`	consult a Sceptic file.
`compile/1`	compile a Sceptic file.
`debug/0`	enter debug mode.
`halt/0`	exit from Sceptic.
`listing/0`	list all predicates/rewrite rules.
`listing/1`	list specified predicates/rewrite rules.
`nodebug/0`	exit debug mode.
`nospy/1`	remove spy point.
`noverbose/0`	exit verbose mode.
`sc/0`	return to Sceptic from Prolog.
`spy/0`	report spy points.
`spy/1`	add spy point on a predicate/trigger.
`verbose/0`	enter verbose mode.

B Debugging Command Summary

B.1 Condition phase commands

Move:

`n`	Nowait	Continue to the prompt
`c`	Creep	Step to next predicate
`s`	Skip	Skip over the full evaluation of this predicate
`l`	Leap	Leap to next spied predicate
`f`	Fail	Force evaluation of this predicate to fail

Spypoints:

`+`	Add Spypoint	Place a spypoint on the current predicate.
`-`	Remove Spypoint	Remove a spypoint from the current predicate.

Display:

`dp`	Display Predicate	Lists the current predicate
`di`	Display Instantiated Predicate	Lists with current variable bindings

Tracking:

There are no separate tracking commands at present.

B.2 Act phase commands

Move:

`n`	Nowait	Continue to the prompt (with optional tracking).
`c`	Creep	Step to next trigger.
`s`	Skip	Skip over the full expansion of this trigger.
`l`	Leap	Leap to next spied trigger.
`sr`	Skip Recognise	Skip over recognise phase for this trigger (so you can see the actions generated and still skip afterwards).
`dr`	Debug Recognise	Debug recognise phase for this trigger.

Spypoints:

`+`	Add Spypoint	Place a spypoint on the current trigger.
`-`	Remove Spypoint	Remove a spypoint from the current trigger.

Display:

`f`	Full Stack	Display the full trigger/action stack.
`w`	Waiting	Display the pending triggers/predicates only.
`t`	Current	Display the current trigger or predicate. (This is displayed automatically before pausing. This command is to recall it if other output has intervened.)

Tracking:

`0`	No tracking.
`1`	Track spied triggers only.
`2`	Track all triggers.

B.3 Recognise phase commands

Pause/display points:

New Rewrite Rule	Start of a new rewrite rule for current trigger.
Instantiation	Complete instantiation of current rewrite rule.
New Condition	Start of a new condition of current rewrite rule.
Success	Success of current condition.
Failure	Failure of current condition.
Retry	Retry of previous condition.

Move:

`n`	Nowait	Continue to the end of the rec phase (with optional tracking).
`p`	New Rewrite Rule	Stop at the start of the next rewrite rule, or end of rec.
`i`	Instantiation	Stop at the next instantiation, or new rewrite rule, or end.
`s`	Success	Stop at the next condition success, instantiation, rewrite rule or end.
`c`	Creep	Stop at next point.

Display:

There are no separate display commands at present.

Tracking:

`0`	No tracking.
`1`	Track rewrite rules and instantiations.
`2`	Track rewrite rules, instantiations and succeeding conditions.
`3`	Track all points.

C Error Messages

C.1 Sceptic Error Messages

undefined predicate Undef While debugging a predicate condition an undefined predicate, *Undef*, has been called. Only while debugging predicates will Sceptic catch undefined predicate calls, else where the underlying Prolog system will handle such calls.

Term is not an action. The named *Term* has been called on the RHS of a rewrite rule, but is not defined as an action (but possibly as a predicate condition).

consult/1: illegal argument: Structure. The *Structure* given as an argument to consult was neither an atomic file name nor a list of atomic file names.

compile/1: illegal argument: Structure. The *Structure* given as an argument to compile was neither an atomic file name nor a list of atomic file names.

library: illegal argument: Structure. The *Structure* given as an argument to `library` during a consult or compile was neither an atomic file name nor a list of atomic file names.

File does not exist: File The named *File* does not exist in any of the expected directories, or with an expected file extension.

None of these files exists: File List The named files in *File List* do not exist in any of the expected directories, or with an expected file extension.

illegal initialisation file Sceptic's initialisation file has been given a non-atomic name, possibly during the original installation procedure at your site.

Cannot classify input term: Input The named term *Input* entered at the prompt can not be classified as either an action or as a condition. Alternatively this may be the result of a declaration in a file, currently being loaded, which calls an unknown action or condition.

Ignoring unknown declaration: Declaration Any unknown declarations found while loading a file are reported.

Argument must be instantiated Uninstantiated arguments are not allowed for `spy/1`, `nospy/1`, and `listing/1`.

Argument must not be numeric A numeric argument has been given to `spy/1` or `nospy/1`.

Functor must be an atom Any functors to `spy/1`, `nospy/1` and `listing/1` must be atomic.

Arity must be an integer An arity must be expressed as an integer to `spy/1`, `nospy/1` and `listing/1`.

Cachable verification failed (this should not happen!) You should never see this error message.

Number of repeats must be a positive integer The second argument to `sc_repeat/2` was illegal.

C.2 Prolog Error Messages

The underlying Prolog system may report Prolog errors itself, the types of errors being dependent upon your version of Prolog.

`| ?-` If your Prolog exits to its prompt, usually `| ?-`, then you can restart Sceptic with `sc.`

D Backwards Compatibility

The prime difference between this version of Sceptic and prior versions is the clear distinction between conditions and actions and the treatment of each as equals. Thus conditions or actions can be entered at the Sceptic prompt, and each will be treated appropriately (functioning in an "ask" mode for conditions, generating all solutions, and in a "tell" mode for actions, propagating their consequences throughout the database). There are a few aspects of the current version of Sceptic which are incompatible with earlier versions. These are:

Breadth-first expansion: This version expands actions depth first only. In version 2 this was the default behaviour, and was used almost exclusively. The breadth-first option was dropped in version 3 to simplify development, and we have not, as yet, missed it. We may re-introduce it later if required, though we have no plans to do so at present. The `current_expansion_direction` parameter of version 2 is obsolete.

Rewrite Rule syntax: The syntax of versions 1 and 2 did not distinguish the trigger, other than by position. That is, in old syntax the rewrite rule

```
Trigger: Condition1, ..., ConditionN => Action1, ..., ActionN.
```

was written

```
Trigger, Condition1, ..., ConditionN => Action1, ..., ActionN.
```

In version 3, the current syntax was used except that the trigger `eoc` was optional, so a rewrite rule without a colon was an `eoc` rewrite rule. All eoc-rewrite rules, including "eoc-only" applications, must now explicitly include the `eoc` trigger.

Files in the old formats must be updated by hand before they can be used in the version 4. The special mode for loading old-format files available in version 3 has been dropped.

Abbreviations: Versions 1 and 2 contained a small number of built-in abbreviations. These have been removed, but similar functionality can be obtained by defining trivial predicates. See the truth maintenance case study for an example.

Configuration predicates deleted: Previous versions of Sceptic included a configurable prompt and a configurable file extension. These facilities are no longer available.

Default conditions/actions removed: Previous versions of Sceptic allowed default conditions and actions to be specified. This default behaviour has been removed on the grounds that fully explicit programs are generally easier to read.

Escape to Prolog: Escape to Prolog (via end-of-file) is no longer possible. Typing end-of-file at the Sceptic prompt now exits the Sceptic system. This behaviour is consistent with treating conditions and actions as equals: there should no longer be any need to escape to Prolog.

Debugging: Version 3 was the first version to have a Sceptic-oriented debugger. In version 4 this debugger was extended to allow debugging of conditions.

New commands to the Sceptic prompt: The following were new commands for version 3: `spy`, `nospy`, `lp`, `oload`. In version 4, the first two of these commands have been extended so that they may take functor/arity specifications for their arguments. The `lp` command has now been replaced with the `listing` command (similar to Prolog's `listing` command, but treating conditions and actions as equals). The `oload` command has been removed.

Execution of actions: In version 3, predicates which occurred as actions were repeatedly called to produce all solutions, but did not instantiate variables contained within them. One side effect of this was that there was no difference in version 3 between `retract/1` and `retractall/1` as an action. In version 4, this has been altered: primitive actions are executed just once (though again, variables are not instantiated).

Calling Sceptic from Prolog The predicate `sc_process/1`, whilst not mentioned elsewhere in this manual, may still be used to trigger a Sceptic rewrite rule from within a Prolog condition. `sc_process` cannot be called directly from the prompt, nor can it be called as a trigger.

E Porting Programs

The following examples of porting programs between Prolog and Sceptic may be used as guidelines. Note that whilst Sceptic code can be mechanically translated into Prolog code, the reverse translation is not so simple.

E.1 Porting Programs from Prolog to Sceptic

- Failure driven loops tend to easily map onto Sceptic rewrite rules. Care should be taken to ensure there is no dependency between the conditions and actions.

```
% Prolog
map :-
    data(X, old),
    map(X, New_X),
    retract(data(X, old)),
    assert(data(New_X, new)),
    fail.
map.
```

```
% Sceptic
map:
    data(X, old),
    map(X, New_X)
 => retract(data(X, old)),
    assert(data(New_X, new)).
```

- Procedural Clauses may also be easily rewritten in Sceptic.

```
% Prolog
start :-
    next_value(X),
    solve(X),
    map, !,
    start.
```

```
% Sceptic
start:
    next_value(X)
 => solve(X),
    map,
    start.
```

E.2 Porting Programs from Sceptic to Prolog

All Sceptic rules can be mapped into prolog using the following approach. The predicate `call_list/1` is required in addition to the clauses representing the rewrite rules. Predicate definitions from Sceptic programs naturally remain unchanged in Prolog.

```
% Sceptic
trigger1(X):
    c1(X),
    c2
 => a1,
    a2.

trigger1(_):
    c3
 => a3.
```

```
% Prolog
trigger1 :-
    findall((a1,a2),(trigger1(X)=trigger1(Y),c1(Y),c2),AR1),
    findall((a3),   (trigger1(_)=trigger1(_),c3),      AR2), !
    call_list(AR1),
    call_list(AR2).
```

```
% call_list/1 takes a list or goals and calls each goal in turn. Each is embedded in a
% double negation to prevent variable binding between goals (which here correspond to
% Sceptic actions).

call_list([H|T]) :-
    call(not(not(H))),
    call_list(T).
call_list([]).
```

E.3 Porting Programs Between Sceptic3 and Sceptic4

The new features of Sceptic are detailed in section D, for a quick conversion from Sceptic version 3 these points in particular should be considered.

Predicate abbreviations: If you wish to use abbreviations (like ~ for `member`) then they must be explicitly defined in your program.

Primitive actions are executed only once: Before Sceptic version4 `retract/1` and `retractall/1` did the same thing, they now behave as they do in Prolog, `retract/1` only removing it's argument from the database once.

Don't confuse conditions and action: Sceptic version 4 will expect only actions on the RHS of a rule. If your version 3 program calls predicates as if they were rewrite rules then you will have to convert them into rules. Sceptic version 4 is more rigorous than previous versions as to rule and predicate type checking during execution, and attempts to call a predicate as if it were an action will result in an error being reported.

It is possible to mislead Sceptic into executing a prolog predicate as an action by wrapping the predicate call in the genuine action `pl/1`. A call on the RHS of a rule would change from `pred` to `pl(pred)`, the action `pl/1` executes it's Prolog argument once. Use of `pl/1` is strongly discouraged as it crosses the boundary between declarative rules and logical predicates, however it could be used to quickly get less formal Sceptic applications up and running, and it's presence highlights where this distinction is not clear.

default_action: There is no longer a `default_user_action` or any other similar default settings in Sceptic. A very simple example of a command loop which does a similar job is shown here where `my_action/1` is the main trigger for the application.

```
loop:
    write('my system>'),
    read(X)
 => loop_action(X).

loop_action(X):
    X \== exit
 => my_action(X),
    loop.
```

By starting `loop` a customised prompt appears and only the argument for the main trigger need be typed, rather than the trigger name and the argument.

F Prolog-specific Issues and Portability

F.1 Relation to Prolog: Internal form

In the current implementation, Sceptic rewrite rules are translated into clauses for a single Prolog predicate. The rewrite rule:

```
T: C1, C2, ..., Cn => A1, A2, ..., An
```

becomes the clause:

```
'$rewrites'(T, [A1, A2, ... , An|T]-T) :- C1, C2, ..., Cn.
```

That is, there is a satisfied instantiation of a rewrite rule for `T` with actions `[A1, A2, ..., An]`, if conditions `C1, C2, ..., Cn` are satisfied. The use of difference lists to represent actions allows lists of lists of actions to be efficiently concatenated.

The top-level control mechanism drives the clauses to provide the distinctive features of the system: all-solutions instantiation and data-driven propagations of triggers.

This representation is not necessarily a permanent choice. It is a compromise, which attempts to retain simplicity of structure for explanation and debugging, whilst improving efficiency over previous versions by having conditions evaluated as 'raw' Prolog.

F.2 Sceptic availability and portability

Versions of Sceptic are available for Quintus Prolog and SICStus Prolog. Prologs with a compatibility library for either of these, like Eclipse, also run Sceptic. It is currently being ported to Poplog Prolog. We may consider attempting further ports if there is sufficient demand.

This version of Sceptic consists of a makefile and nine source files:

`sc_debug.pl`	debugger command interpreter
`sc_expand.pl`	condition/action execution
`sc_file.pl`	file handling
`sc_interface.pl`	user interface
`sc_lib.pl`	system library files
`sc_main.pl`	main declarations etc.
`sc_primitive.pl`	definitions of primitives
`sc_site.pl`	site specific definitions
`sc_prolog.<my_prolog>`	Prolog-specific code (e.g., `sc_prolog.quintus`)

The intention is that the `sc_prolog.<my_prolog>` file should contain all parts which vary between different Prologs, and that porting to a new Prolog should only involve providing suitable definitions of the predicates in this file. This is largely the case, although some Prologs may require more extensive alterations (e.g., those which require different comment conventions or other syntax changes).

The predicates defined in a Prolog-specific way are:

`not(+X)`: Succeed iff `X` fails.

`sc_command_line_arg(-L)`: Returns any command line arguments given to Sceptic.

`sc_machine_version(-MachineAtom)`: `MachineAtom` describes the machine architecture.

`sc_name(?Atom,?Chars)`: `Atom` converts to list of characters `Chars`.

`sc_op(+Precedence,+Type,+Name)`: Used for defining operators, since Prologs vary in their argument order for `op/3`.

`sc_predicate_property(:Head,?Value)`: Detects if the predicate corresponding to `Head` is compiled, interpreted or built-in.

`sc_prolog_version(-PrologAtom)`: `PrologAtom` describes the Prolog.

`sc_translate_primitive_condition(+Condition,-Translation)`: Defines the internal translation of a Prolog-specific primitive condition

`sc_translate_primitive_action(+Action,-Translation)`: Defines the internal translation of a Prolog-specific primitive action

`sc_garbage_collect`: Defined to garbage collect or succeed trivially if not applicable.

`sc_open(+File,+Mode,-Source)`: Open `File` in mode `Mode`, returning a handle `Source`.

`sc_read(+Source,?Term)`: Read `Term` from `Source`.

`sc_close(+Source)`: Close `Source` from which you have been reading.

`sc_eof(+Term)`: Recognise `Term` as the term returned by `sc_read` at end of file.

`sc_exists(+File,-AbsFile)`: Succeed if `File` exists, returning `AbsFile`, its absolute file name.

These descriptions are intended simply to indicate the purpose of the predicates. For more details, necessary to ensure correct behaviour in a port, refer to comments in the example files.

The code organisation described above is intended to facilitate portability of Sceptic *itself* between versions of Prolog. However this is insufficient to ensure portability of Sceptic *applications*. Sceptic allows full access to the underlying Prolog (via `pl/1`), and therefore does not prevent the application writer from using any aspect of that Prolog version. Although we strongly discourage the use of `pl/1`, Sceptic also makes available Prolog specific primitives. We have found these primitives to be very useful, and are reluctant to limit their use. Clearly applications which also employ these will be limited by the portability of the Prolog specific primitives.

F.3 The Quintus/SICStus Prolog ports

Quintus and SICStus Prolog provide several built-in predicates which are not available in many simpler systems. These predicates are also available, as primitive conditions/actions, in Sceptic when built under Quintus and SICStus Prologs.

Additional stream-based conditions:

`current_stream(?F,?M,?S)`: This succeeds with `F` instantiated to a file which is currently open in `M` mode (`read`/`write`/`append`) and attached to the stream `S`.

`current_input(?S)`: This succeeds with `S` instantiated to a term representing the current input stream.

`current_output(?S)`: This succeeds with `S` instantiated to a term representing the current output stream.

Additional miscellaneous conditions:

`atom_chars(?A,?C)`: This corresponds to `name/2` with its first argument restricted to being an atom.

`number_chars(?N,?C)`: This corresponds to `name/2` with its first argument restricted to being a number.

Additional stream-based actions:

`set_input(S)`: Set the current input stream to `S`.

`set_output(S)`: Set the current output stream to `S`.

Additional output actions:

`format(F,A)`: The `format` primitive allows output facilities similar to those given by the C `printf` function. `F` is a format string, containing control characters, and `A` is a list of arguments. (For full details see the relevant Prolog manual.)

`format(S,F,A)`: Analogous to `format/2`, with the first argument indicating the stream to which the output should be sent.

Saving the program state:

In Sceptic built under SICStus, the primitive action `save/1` will save the current state of the system to the file specified by its argument. Executing this file will restore the system and it will continue from where it was when the save command was issued.

In Sceptic built under Quintus, the program state may be saved with the primitive action `save/2`. This takes a filename as its first argument and a command (condition or action) as its second argument. When the file is restored the command will be executed. Note that if the second argument is an action it may be necessary to embed it in `sc_process/1`.

Index