Sceptic User Manual

September 1989, revised March 1990
Version 3.0, December 1990

Saki Hajnal
John Fox
Paul Krause

Advanced Computation Laboratory
Imperial Cancer Research Fund
61 Lincoln's Inn Fields
London
WC2A 3PX

Contents

1 Introduction

Sceptic draws upon experience using Prolog and Props2, the latter a language developed at ICRF which combined some features of a logic programming language with others more typical of production-rule interpreters.

Prolog is a practical AI programming language, but its roots lie in formal logic, theorem proving and databases. One of the main reasons for developing Props2 was the perception that a language for building intelligent automata should also support opportunistic computation (computation in response to events occurring unpredictably in time). Prolog applications can be programmed to have this characteristic (indeed Props2 is written in Prolog) but Prolog's design emphasises formal logical properties above any other properties considered to be desirable for autonomous systems.

Production systems, which have been widely used for modelling human cognition, emphasise event-oriented rather than state-oriented computations. Like Prolog, however, production systems have to compromise with the requirements of practical programming. OPS5, for example, has acquired many features and data structures during its development which have obscured its underlying computational model, made it a somewhat difficult language to use and its applications difficult to maintain. Consequently, since Props2 was also intended as a serious engineering tool, a second aspect of its design was the preservation of a sound, declarative style of programming.

Props2's emphasis on data-driven computation was supported by a truth-maintenance system, a default mechanism, and a number of experimental mechanisms for logical reasoning about control. However the specific form of these features was not appropriate for all applications. Sceptic has been designed to support a data driven style of computation, but without imposing any constraints on the exact truth maintenance and default mechanisms which may be implemented.

Sceptic interprets a production syntax and executes productions in a forward or data-driven control cycle. Sceptic is currently implemented in Prolog, and provides full access to the underlying Prolog system. As a consequence, Sceptic does not in principle do anything that cannot be achieved with Prolog, but it does provide a valuable syntactic differentiation between the control and procedural aspects of Prolog on the one hand, and the theorem proving (Prolog as a logic language) and calculation part of Prolog on the other. This is illustrated in considering how a computer model of a rational autonomous agent may be programmed. Sceptic would be used to express the control strategies and meta rules for deciding when a decision need be made, theorem proved, or beliefs about the environment updated. However, the underlying resolution and unification of Prolog would still be used to prove theorems about the agent's environment on demand.

So, for example, in encoding the pursuer-evader problem[1] in Sceptic, whenever an action occurs, Sceptic productions will be fired which control the revision of the agent's beliefs about the surroundings and the corresponding actions taken as a consequence of the agent's perceptions.

---

[1]Pursuers have sensors with a given range, and can capture evaders that are within a second given range. Evaders also have sensors with a finite range and have set tactics for evading detected pursuers. The problem is an interesting vehicle for exploring the behaviour of autonomous and perhaps co-operating agents. See Appendix B.

Similarly, a truth maintenance system may be succinctly expressed in Sceptic, with the productions describing the belief revision operations to be carried out when some new information is asserted into or retracted from a knowledge base. Prolog, as mentioned above, will be used to prove any theorems about the world upon which the firing of a production may be conditioned, or to obtain any information that is required.

The syntax of Sceptic will be described more fully in the next section. By way of introduction, Sceptic production rules consist of triggers, conditions and actions. When a trigger T_1 is generated, the rules prefixed by T_1 (T_1: C => A) will be fired if the conditions C are satisfied. In using Sceptic, it is intended that such a rule have an imperative reading. If a trigger T_1 is signalled and the antecedent C is satisfied at that moment, then do A. In keeping with this imperative style and the transient nature of the validity of A, it is natural to read A as specifying a course of action(s), such as assert(P), retract(P), or fire trigger T_2. (Triggers may also be generated by a user.) As an example, we may consider the first production in the truth maintenance system to be described later:

```
la(X): not(X) => assert(X), fw(X).
```

This may be read: "To logically assert X (the trigger la(X)) check first that X is not already known (the condition not(X). If this is the case, assert the fact X into the database (assert(X)) and then forward chain all consequences of X (fw(X), where fw(X) is another trigger)".

Sceptic does not at present provide any automated theorem proving capability beyond that available in the underlying Prolog. However, it is our belief that Sceptic provides a natural and expressive syntax for programming control strategies and belief update procedures. In principle, any available theorem proving or evaluation strategy may be called within the execution of a Sceptic production. The value of Sceptic is in programming the event oriented activities of an autonomous system.

This manual describes version 3.0 of Sceptic. For changes from previous versions refer to Appendix D.

2 Syntax

The Sceptic notation consists of productions of the following form:

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

where `T` is a trigger, `C1`–`Cn` are conditions, `A1`–`An` are actions, `=>` is a reserved symbol, and fullstop is a terminator. Triggers, conditions and actions may contain embedded variables, and there is an implicit universal instantiation, i.e., a production fires for all instantiations of its conditions. A condition may (in the current implementation) be any Prolog predicate, with a configurable default condition. This may be used, for example, to provide an 'object-level match', i.e., any condition which is not a defined predicate is taken to be a match on some database, where the precise definitions of the match function and database form are configurable. An action may also be any Prolog predicate, a further trigger, or a default action which is again configurable.

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

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

3 Using Sceptic

3.1 Starting the system

To start the system, type:

```
% sceptic
```

to the unix prompt (here `%`). The system starts with a message stating the current version, and a prompt (here `sc>`, but configurable).

```
Sceptic Version 3.0
sc>
```

This is an 'empty' system. The first step is normally to load a Sceptic application, consisting of one or more files of productions (or Prolog code).

3.2 Loading a file of productions

To load a file called test.sc, type:

```
sc> sc_load(test).
```

The file extension `.sc` will be used automatically, unless the file `test` exists. The choice of extension is configurable. 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> sc_load('test.1').
```

An alternative command `vload` will give verbose feedback on the translation of the input.

An input file may consist of a mixture of productions in the syntax described above and ordinary Prolog code. In particular, even if the application is 'pure' Sceptic with no significant Prolog code, you are likely to need some minimal predicates, and possibly some declarations specific to your version of Prolog (although common ones such as dynamic declarations and operator definitions have been made portable where possible). Comments use the normal Prolog conventions (`%` or `/*    */`).

The productions loaded can be inspected using the commands:

`lp.`	list all productions
`lp(Term).`	list productions matching Term

For the moment, the command `lp(Term)` requires its argument to be a structure. In the future, it may be made to accept just a functor name or functor/arity combination. The productions are displayed in approximately the original syntax. 'Approximately' here means that some conditions may be wrapped by `sc_one_condition/1` (see section 5.1 Internal Form).

Here is a very small application:

```
% File: mini.sc

% Declare f/2 as dynamic so that it will be recognised
% as a predicate even before there are any clauses.
% sc_dynamic should simulate this behaviour in the current Prolog.

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

4

```
% 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 production needing no conditions other than the trigger
% has the single dummy 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 Version 3.0
sc> sc_load(mini).                           { load application        }
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 productions for showf fire }
sc> input(jo).
sc> showf.
items entered so far:
   fred
   jo
sc>
sc> listing(f).              { an example of calling a Prolog predicate }
f(fred,u).                         { Prolog shows the 'raw' clauses       }
f(jo,u).
sc>
```

3.3   Escaping to Prolog

The example above is running in Sceptic mode, with a top loop (prompt `sc>`) reading triggers and processing them according to Sceptic's execution model. This loop can be terminated (by <end-of-file>) and restarted (by the Prolog call `sc.`).

```
sc> ^D                   { type control-D = <end-of-file> }
yes
| ?- X is 2 + 3.         { this is raw Prolog }
X = 5
| ?- sc.                 { re-start the Sceptic loop }
Sceptic Version 3.0
sc>
```

To exit all the way to the operating system, type <end-of-file> twice.

3.4   Typing in extra productions

The normal way to load productions is from a file. Occasionally it may be convenient to type them directly, and a mode is provided for this:

5

```
sc> pmode.                                { go into production-mode    }
scp> test: true => write(testing), nl.    { type in a valid production }
input: test:true=>write(testing),nl       { this is verbose feedback   }
sc_satis_prod(test,[write(testing),nl]):-true  { showing the internal form }
scp> ^D                                    { type control-D             }
sc> test.                                  { trigger the new production }
testing
sc>
```

NOTE: All productions for a single trigger must be entered at once, either from a file or from the keyboard. Do not use pmode to add an extra production for an existing trigger — the existing definition would be overwritten.

3.5   Configuring the system

At startup, the system loads a file called 'sceptic.ini'. It searches for this file in a sequence of directories: current directory, user's home directory, central Sceptic system directory, and loads the first one found. Thus the user can provide configuration information for an application (current directory), for all their applications (home directory) or rely on the system's default configuration:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% sceptic.ini - Default Sceptic configuration file
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% ESSENTIAL predicates - you must have these definitions or replace them

sc_prompt(top, 'sc> ').
sc_prompt(load, 'scp> ').

sc_file_extension("sc").

sc_default_condition(X,sc_error('Invalid condition: ', X)).
sc_default_action(X,sc_error('Invalid action: ', X)).
sc_default_user_action(X,sc_error('Invalid user action: ', X)).

% OPTIONAL predicates - you need these if your application uses them.
% They were built-in abbreviations in previous versions of Sceptic

% o(X) :- write(X).
% a(X) :- assertz(X).
% r(X) :- retract(X).
% '~'(X,Y) :- member(X,Y).
```

The supplied behaviour for `sc_default_condition`, `sc_default_action` and `sc_default_user_action` is to print an error message, on the assumption that all conditions should be defined predicates and all actions should be defined predicates or triggers. These configuration parameters can be used in an application-specific way to define non-trivial default behaviours.

For example, `sc_default_condition` can be used to specify that all conditions which are not recognised existing predicates should be interpreted as matches on some object-level database, by replacing the line in `sceptic.ini` by:

```
sc_default_condition(X, objmatch(X)).
```

6

and providing the definition of `objmatch/1`. For example, for a system which expects an object-level database of facts represented by clauses of the form `f(Patt,Source)` an appropriate definition may be:

```
objmatch(not(X)) :- !, \+ f(X,_).
objmatch(X)  :- f(X,_).
```

Similarly, the definition

```
default_user_action(X,user(X)).
```

means that any user input which is not recognised as a defined predicate or trigger will be handled by `user/1`. The definition of `user/1` is up to the application-builder, e.g., if we add to the example `mini.sc` (above) the production:

```
user(X): true => input(X).
```

the user no longer has to type `input(Patt)`, and the session becomes:

```
sc> showf.
items entered so far:
sc> fred.                                      { will call user(fred) }
sc> showf.
items entered so far:
    fred
sc>
```

4   Execution

4.1   The cycle

The execution of Sceptic productions is as follows. For any trigger, each production 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. Note that

a. All productions matching a trigger are used. There is no notion of a 'cut' to indicate that the 'right' production 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 queued depth-first. (In previous versions, this order could be changed to breadth-first. The effect of expansion order is an open issue, which relates to that of potential parallelism.)

Queued actions are interpreted one at a time. If they are predicates they are called. If they are triggers, they are executed as above, causing another burst of expansion. Unrecognised actions are handled by `sc_default_action`.

4.2   End-of-cycle processing

A distinguished trigger `eoc` is generated at the end of the cycle. If there are productions 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 an `eoc` trigger generates no new tokens, the cycle terminates. (This may occur either if there are no `eoc`-driven productions, or if the productions have conditions which produce no instantiations.) Note that an `eoc` production with no other conditions will cause looping. It is the user's responsibility to ensure that suitable conditions are present.

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 production will execute the agenda until there is nothing left on it, and then terminate.

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

Syntactically the `eoc` trigger is optional. The above production can also be written:

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

(This is why productions with only a trigger need the dummy condition `true`.)

Another use of `eoc` is to construct continuous applications which do not return to the prompt. See Appendix B for an example.

4.3 Triggers

Triggers are not conditions 'becoming true'. They are only active as they pass through the execution cycle and are then lost. The sequence

```
T: C1, ... Cn
```

is *not* to be interpreted as a logical conjunction of conditions (with associated expectations of being notified when conditions change). Its force is that the conditions C1–Cn are checked (and may generate instantiations) at the moment when trigger T passes through. This is not to say that Sceptic cannot be used to implement logical dependencies and truth maintenance (see Appendix A), only that the raw Sceptic engine is a simpler forward propagation machine driven by 'events' called triggers.

5 Relation to Prolog

5.1 Internal form

In the current implementation, Sceptic productions are translated into clauses for a single Prolog predicate. The production:

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

becomes (approximately) the clause:

```
sc_satis_prod(T, [A1, A2, ... , An]) :- C1, C2, ..., Cn.
```

That is, there is a satisfied instantiation of a production for `T` with actions `[A1, A2, ..., An]`, if conditions `C1`, `C2`, `...`, `Cn` are satisfied. The word 'approximately' is used because the conditions `C1` to `Cn` will only appear unchanged, as shown, if they are already recognised as Prolog predicates at the time the production is loaded. Unrecognised conditions will by 'wrapped' by the predicate `sc_one_condition(C)`, since they require a runtime check to see if they are predicates or should be handled by the `sc_default_condition` mechanism. This wrapping will be needed either if the condition in question is a predicate not yet defined (which will be loaded later) or if it is a variable to be instantiated during execution. Because of the former order-sensitive case, efficiency can be increased by loading predicates before productions which use them. In the future, a multi-pass load might be provided to handle this case automatically.

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.

5.2 Calling Prolog from Sceptic

Any Prolog call can be typed to the Sceptic prompt. However it will be executed using the Sceptic control which (in the current implementation) means that all solutions will be generated, but without the variable bindings being returned to the user. Therefore this method of call is only useful for predicates with side-effects (a handy example is `listing/1`).

A means of calling normal Prolog (i.e., first solution only and echoing the variable instantiation found) is under consideration.

5.3 Calling Sceptic from Prolog

The top-level Sceptic loop can be started by the predicate `sc/0` as described above, putting the system in Sceptic-mode. However it may be desirable, from within a normal Prolog application, to trigger a single Sceptic cycle. This is achieved by calling the predicate `sc_process/1`. For example, suppose the `input(X)` trigger in `mini.sc` above was the entry point to a more sophisticated Sceptic application such as a truth-maintained logical dependency system. A Prolog application could submit a pattern to that application by the call `sc_process(input(Pattern))`. If the Sceptic-maintained database in that example were as before (`fred` and `jo` already input):

```
?- sc_process(input(mary)).  { we are in Prolog but want Sceptic to handle this }
yes
| ?- listing(f).                                { check what happened }
f(fred,u).
f(jo,u).
f(mary,u).
yes
| ?- sc_process(showf).              { showf is also for Sceptic, not Prolog }
items entered so far:
    fred
    jo
    mary
yes
| ?
```

6 Debugging

6.1 The Sceptic debugger

This section describes the Sceptic-oriented debugger. The facilities for Prolog debugging depend on the underlying Prolog system.

The debugger has two phases, recognise (rec) and act. By placing spypoints and using moving/displaying commands and tracking levels you can trace the trigger expansion at various granularities, remaining in the 'act' phase. At any trigger at which you have paused, you can choose to debug the 'rec' phase, and trace the evaluation of production conditions, also 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.

To start debugging, place a spypoint on a trigger. The commands are:

```
spy(Term).
nospy(Term).
```

In this version, these commands require their argument to be a structure, e.g., `spy(la(_))`. In the future, they may be made to accept just a functor name or functor/arity combination.

The system will stop and prompt at the first occurrence of a spied trigger. The prompt is of the following form:

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

where:

> `Phase` is `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

You may input commands (see below) to the prompt. Note that debugging commands are terminated by RETURN and do not require a fullstop. 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.

6.2 Initial command settings

At the start of each 'act' phase debugging cycle (i.e., when you have input something to the prompt), the initial command is 'leap' (`l`). At the start of each 'rec' phase debugging cycle (i.e., when you have chosen the command `dr`), the initial command is 'new production' (`p`).

6.3 Act phase commands

6.3.1 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

6.3.2 Add/remove spypoints

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

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

Note that these commands (and `spy(Structure)` or `nospy(Structure)` to the Sceptic prompt) will add Sceptic or Prolog spypoints as appropriate. However, mixed debugging may be confusing.

6.3.3 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 productions:

```
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 instantion
> 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)
```

6.3.4   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
> 3   Track all triggers and predicates

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.

6.4   Rec phase commands

You enter the rec 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 Production   Start of a new production for current trigger
> Instantiation   Complete instantiation of current production
> New Condition   Start of a new condition of current production
> Success   Success of current condition
> Failure   Failure of current condition
> Retry   Retry of previous condition

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

6.4.1 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 Production	Stop at the start of the next production, or end of rec phase
i	Instantiation	Stop at the next instantiation, or new production, or end
s	Success	Stop at the next condition success, instantiation, production or end
c	Creep	Stop at next point

6.4.2 Display

There are no separate display commands at present.

6.4.3 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 productions and instantiations
2	Track productions, 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 productions and instantiations, tracking level 1 will have no effect. However, if you use the 'new production' command p with this level, you will see all instantiations passed between then and the next production.

7 Caveats

7.1 Robustness

The system as it stands is very flexible, since productions are simply translated into Prolog, and conditions and actions may be arbitrary Prolog calls. There are few safeguards against clashing predicate names, accidental abolition of dynamic declarations, etc. This can lead to unexpected results. However, a system with more sophisticated checking mechanisms would probably require more complete and careful declarations from the user. This is felt to be premature.

7.2 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. Nonetheless, it may turn out that 'natural' expression in Sceptic productions and efficient execution are in conflict, at least in some cases. (This is not to say that having such an expression of behaviour would be useless in those cases, only that other means might be required to give efficient performance.)

7.3 Why not just use Prolog?

Sceptic does nothing (in this implementation) which cannot be coded directly in Prolog. The potential interest in the Sceptic notation and control mechanism depends on the extent to which it naturally expresses a range of applications. If it does, then the corresponding direct coding would obscure the common structure underlying those applications. If not (e.g., if an application does the 'meat' of its work in complex special conditions coded in Prolog) then the extra layer of notation may add nothing.

7.4 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, if any, are unclear. However, Sceptic may 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 productions has provided a means of discussing alternative behaviours in a way that (larger) raw Prolog programs or (vaguer) English descriptions have not.

8   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 programs in Sceptic.

Non-instantiation of variables in actions.   Variables which are uninstantiated when an
action is called will not be bound after the call. For example the production

```
write_sum(A,B): true => C is A+B, write('A+B='), write(C), nl.
```

will give the following response:

```
sc> write_sum(1,2).
A+B=_1700
```

whereas the production

```
write_sum(A,B): C is A+B => write('A+B='), write(C), nl.
```

will respond as intended:

```
sc> write_sum(1,2).
A+B=3
```

This is a result of the way in which Sceptic is implemented and is a possibly controversial feature.
Currently, predicate actions are called one at a time for 'all-solutions', so that although the
addition takes place and succeeds, it fails on retry and the variable C is once again uninstantiated
by the time the next action is reached. The rationale for this behaviour is that instantiation
of variables should remain an aspect of checking the satisfaction of conditions. Read the above
rule as "to write out the sum of two numbers, if C is their sum, then output C".

Dynamic Predicates.   Predicates which are not in existence when an application is loaded
but which will be asserted during the Sceptic session must be declared as dynamic, since Sceptic
behaviour relies on distinguishing predicates and non-predicates.

Multiple use of Triggers.   All productions with the same trigger name must be loaded from
a single file. Loading a second file containing productions with already used trigger names
will result in the original productions being overwritten. This is analogous to the behaviour
of 'reconsult' in Prolog. No notion of ownership of triggers by a file is recorded when a file is
consulted. Thus, if all productions of a given trigger are deleted from a previously loaded file
and that file then reloaded, the corresponding productions will not be removed.

Mixing default uses.   As mentioned in Section 6.1, there are only a few built-in constraints
or checks on the usage of predicate or trigger names (the load will only detect clashes between
triggers and predicates of the same functor *and* arity). This can lead to difficulties, for example,
if the same predicate name is used in a condition and in an action, with a default Sceptic action
declared. If the predicate has been asserted into the database so that a straight call to it will
succeed as a condition, the use of the same predicate name as an action may not work as
intended. The action will be recognised as an existing predicate so the default Sceptic action
will not be called.

For example, the default Sceptic action may be declared as revise which removes the current
predicate and asserts an updated value:

```
        sc_default_action(R,revise(R)).
```

The following Sceptic production is intended as part of an update procedure which calls a
`last_count` predicate, increments the value of the associated argument and revises the value
of the `last_count` predicate's argument:

```
        update: last_count(N), N1 is N+1 => last_count(N1).
```

What will actually happen is that when the action is called, `last_count` will be recognised as
an existing predicate, the default Sceptic action will not be called and instead a straight call
`last_count(N1)` will be made. This last call will fail as the current value of the argument of
`last_count` is not equal to the value of `N1`.

Old format and 'eoc-only' applications.   In Version 3.0 the trigger `eoc` is optional, so a
production without a colon is an eoc production. This is syntactically indistinguishable from
an ordinary production in the syntax of previous versions. In an attempt to help existing
users, this version detects a load in which no productions have triggers, assumes it is an old-
format application, and warns the user accordingly. This means that it is *not* possible to load
a new-format application which contains *only* `eoc` productions.

A Truth Maintenance Example

As remarked above, Sceptic productions are not logical dependencies, and there is no notification of changes in conditions, or truth maintenance.

The following example uses Sceptic to build up in stages to a logical forward chaining and truth maintenance application, similar to part of Props2.5.

Stage 1: Logical assert (object-level forward chaining)

This example will use the structure `Antes->>Conses` to represent a logical dependency rule, where `Antes` is a list of antecedents and `Conses` is a list of consequents. The functor `->>` can be used infix with a suitable operator definition. It is emphasised that this is just a structure in the database, and the choice of functor is arbitrary (the more natural choice `->` is a Prolog built-in in some Prologs). We will use database assertions of the form `f(X)` to hold 'facts'. A suitable definition of `sc_default_condition` will cause conditions to default to matching on this structure. We wish to define a trigger `la(X)` which will perform logical assertion of a proposition `X`, i.e., add `X` to the database together with all its consequences as implied by the dependency rules present.

[A1] `la(X): not(X) => assertz(f(X)).`

[A2] `la(X): not(X), Antes->>Conses, member(X,Antes), Antes, member(C,Conses) => la(C).`

Production [A1] primitively asserts the fact `X` (structure `f(X)`) if it is not already present in the database. The check for absence (suppression of duplicates) is a feature of Props2.5, and corresponds to the notion that once a fact is known, it makes no sense to re-assert its truth.

Production [A2] does the forward-chaining on object-level rules. The use of `member/2` corresponds to the representation of antecedents and consequents as lists. Note that patterns unify as in Prolog, so that if `member(X,Antes)` succeeds (`X` matches one of the Antes) any embedded variables in that antecedent will be unified with the values in `X`. The condition `Antes` on its own works as a means of producing an instantation of all the antecedents because a list as a single condition is handled as a list of conditions, and in each case the default condition is called.

Both these productions contain the check `not(X)`, so that no further processing is done if the fact is already present. This suggests a re-structuring, where the check is done once, and triggers two actions: primitive assertion and forward chaining.

[A1a] `la(X): not(X) => assertz(f(X)), fw(X).`

[A2a] `fw(X): Antes->>Conses, member(X,Antes), Antes, member(C,Conses) => la(C).`

Stage 2: Justification maintenance

The above productions recursively produce all consequences of a fact which is logically asserted, but do not keep track of the dependency relationships between the facts. We can simply add to the database a structure which records that a consequence `C` is supported by an instantiated list of antecedents `Antes`. Here the structure used is `C<-Antes` where again `<-` is an arbitrary functor.

```
[A2b] fw(X): Antes->>Conses, member(X,Antes), Antes, member(C,Conses)
                       => la(C), assertz(C<-Antes).
```

This keeps supports (which are accessible in the database) but does not yet use them for truth maintenance.

Stage 3: Logical retract

We can use the supports structures added above to define logical retract with two analogous productions. 'uw' stands for unwind, and is the opposite of 'fw' for forward chain.

```
[R1] lr(X): X => retract(f(X)), uw(X).
```

```
[R2] uw(X): C<-Antes, member(X,Antes) => lr(C), retract(C<-Antes).
```

Stage 4: Negation

The above productions only handle dependency rules with 'positive' antecedents. We can introduce negation-as-failure as follows.

```
[A3] fw(X): C<-Antes, member(not(X),Antes) => lr(C), retract(C<-Antes).
```

```
[R3] uw(X): Antes->>Conses, member(not(X),Antes), Antes, member(C,Conses)
                         => la(C), assertz(C<-Antes).
```

Note the symmetry between [A3] and [R2], and that between [R3] and [A2b].

Stage 5: Multiple supports

The above system so far implements logical forward chaining and truth maintenance symmetrically over negated antecedents. However, productions [R2] and [A3] logically retract a consequence as soon as one of its support structures is found to be no longer valid, i.e., they assume that any fact only has one support. We can enhance the system to allow for multiple supports, so that a fact is only retracted when its last support is invalidated.

```
[R2a] uw(X): C<-Antes, member(X,Antes) => retract(C<-Antes), uws(C<-Antes).
```

```
[A3a] fw(X): C<-Antes, member(not(X),Antes) => retract(C<-Antes), uws(C<-Antes).
```

```
[R4] uws(C<-Antes): not((C<-Other, Other \== Antes)) => lr(C).
```

The use of extra parentheses to wrap 'not' round several conditions is to enable Prolog to parse the structure correctly as not/1.

The complete example

This is the complete example, with some command definitions and a pair of object-level rules. It also demonstrates the use of simple predicates and operator definitions to provide 'abbreviations', for example to allow the syntax X~Antes for member(X,Antes).

Note the various parts of the application

    a. Application-specific sceptic.ini with configuration predicates

b. Prolog declarations

c. Sceptic productions for the main application, in this case logical forward chaining/tms system

d. Sceptic productions for procedural commands

e. Object level database

This is a small application, and all parts are in `sceptic.ini` plus a single file. For larger applications it may be better to separate the parts into separate files.

```
%---------------------------------------------------------------------------
% File: sceptic.ini
%---------------------------------------------------------------------------

%  STANDARD - unchanged from system default file

sc_prompt(top, 'sc> ').
sc_prompt(load, 'scp> ').

sc_file_extension("sc").

sc_default_action(X,sc_error('Invalid action: ', X)).
sc_default_user_action(X,sc_error('Invalid user action: ', X)).

%  APPLICATION-SPECIFIC

sc_default_condition(X,objmatch(X)).

o(X) :- write(X).
a(X) :- assertz(X).
r(X) :- retract(X).
~(X,Y) :- member(X,Y).

objmatch(not(X)) :- !, \+ f(X).
objmatch(X) :- f(X).


%---------------------------------------------------------------------------
% File: demo.sc
%---------------------------------------------------------------------------
% Prolog declarations and configurable predicates

:- sc_op(800,xfy,<-).
:- sc_op(800,xfy,~).
:- sc_op(800,xfy,'->>').

:- sc_dynamic(f/1).
:- sc_dynamic('<-'/2).
:- sc_dynamic('->>'/2).


%---------------------------------------------------------------------------
% Sceptic productions for logical forward chaining/tms system

% Logical assert. [] references to text.
```

21

```
% [A1a] [A2b] [A3a]

la(X): not(X)  => a(f(X)), fw(X).

fw(X): Antes->>Conses, X~Antes, Antes, C~Conses => la(C), a(C<-Antes).

fw(X): C<-Antes, not(X)~Antes => r(C<-Antes), uws(C<-Antes).

% Logical retract. [] references to text.

% [R1] [R2a] [R3] [R4]

lr(X): X  => r(f(X)), uw(X).

uw(X): C<-Antes, X~Antes => r(C<-Antes), uws(C<-Antes).

uw(X): Antes->>Conses, not(X)~Antes, Antes, C~Conses => la(C), a(C<-Antes).

uws(C<-Antes): not((C<-Other, Other \== Antes)) => lr(C).

%--------------------------------------------------------------------------
% Sceptic productions for procedural commands

% Show current facts

facts: true => o(facts), nl.
facts: f(X) => o('    '), o(X), nl.

% Show rules

rules: true => o(rules), nl.
rules: Antes->>Conses => o('    '), o(Antes->>Conses), nl.

% Show supports for X

supports(X): true => o('supports for '), o(X), nl.
supports(X): X<-Antes => o('    '), o(Antes), nl.

%--------------------------------------------------------------------------
% Object level database

% Two 'dependency rules' according to the chosen convention

[p(X,Y), q(Y,Z)] ->> [r(X,Z)].

[aa(X), not(b(X))] ->> [c(X),d(X)].

%--------------------------------------------------------------------------
```

A sample run

Comments in {...} do not appear on the screen.

```
    Sceptic Version 3.0
```

```
sc> sc_load(demo).
sc> rules.
rules
    [p(_704,_705),q(_705,_710)]->>[r(_704,_710)]
    [aa(_922),not(b(_922))]->>[c(_922),d(_922)]
sc> facts.
facts                                                    { none so far }
                                        { first try the rule with negation }
sc> la(aa(1)).
sc> facts.                          { check conclusions from absence of b(1) }
facts
    aa(1)
    c(1)
    d(1)
sc> la(b(1)).                          { b(1) should cause retractions }
sc> facts.
facts
    aa(1)
    b(1)
sc> lr(b(1)).                      { retraction of b(1) reinstates conclusions}
sc> facts.
facts
    aa(1)
    c(1)
    d(1)
sc>lr(aa(1)).
sc> facts.
facts
sc>
                                        { now test multiple supports }
sc> la(p(1,2)).
sc> la(p(1,3)).
sc> facts.                                       { no conclusions so far }
facts
    p(1,2)
    p(1,3)
sc> la(q(2,4)).
sc> la(q(3,4)).
sc> facts.
facts
    p(1,2)
    p(1,3)
    q(2,4)
    r(1,4)
    q(3,4)
sc> supports(r(1,4)).                              { r(1,4) has two supports }
supports for r(1,4)
    [p(1,2),q(2,4)]
    [p(1,3),q(3,4)]
sc> lr(q(2,4)).
sc> facts.
facts
    p(1,2)
    p(1,3)
```

23

```
        r(1,4)
        q(3,4)
sc> supports(r(1,4)).
supports for r(1,4)
    [p(1,3),q(3,4)]                { one support has gone, but fact still there }
sc> lr(p(1,3)).
sc> facts.
facts
        p(1,2)
        q(3,4)
sc>                                { last support has gone, fact retracted }
```

B   Pursuer-Evader Example

This is a simple demonstration of the data driven style of computation that underlies Sceptic.
Agents have certain attributes, and act as their beliefs about the world are updated as a result
of changes in the state of the world. In this specific example agents belong to one of two classes:
Pursuers or Evaders. Pursuers have attributes speed, range and reach. They can detect any
agent that is within range, target on the nearest detected evader, and eat any evaders that are
within reach. If they detect no evaders they will move about at random, trying to search for
one. Evaders have attributes speed and range. They can detect any agent within range. If
they detect any pursuers, they will turn and run in the opposite direction to the nearest one.
If they detect no pursuers, then they will stay put, thinking they are safe . . .

The agents' beliefs are logically maintained using the truth maintenance system that is described
in the previous section. Each agent has an associated location (in Euclidean coordinates), and
the movement of an agent triggers a revision of the agent's beliefs using the TMS described in
Appendix A:

```
move_agent(N,Speed,Dir):
      location(N,OldX,OldY),
      pos_polar(Speed,Dir,OldX,OldY,NewX,NewY)
   => lr(location(N,OldX,OldY)), la(location(N,NewX,NewY)).
```

The above rule states that to move an agent `N` with speed `Speed` in direction `Dir` from location
`(OldX,OldY)` to a new location that is calculated to be `(NewX,NewY)`, then logically retract the
old location and logically assert the new location.

This action will result in a belief revision using the object rules given at the end of the program
listing. These describe the agents' abilities given above, viz that they can detect any agent
that is within range, target on the nearest detected evader, and eat any evaders that are within
reach. For example, the first ability is described by the rule:

```
    [location(A,HereX,HereY), location(O,Ox,Oy), A \== O, range(A,Range),
     pos_polar(Dist,Dir,HereX,HereY,Ox,Oy), Dist < Range]
 ->> [detects(A,O,Dist,Dir)].
```

After the belief revision cycle, the distinguished trigger `eoc` will trigger any actions that are
now appropriate given the Evaders' updated world view. For example, if an Evader detects any
Pursuers, it will move away from the nearest Pursuer as fast as it can:

```
eoc:
      detects(E,P,Dist,Dir),
      evader(E), pursuer(P),
      not([detects(E,P2,Dist2,Dir2), pursuer(P2), Dist2 < Dist]),
      speed(E,Speed),
      NewDir is Dir+180
   => move_agent(E,Speed,NewDir), evading_move(E,P).
```

If an action of moving an agent is initiated, a brief report is triggered which outputs to the
screen the details of the movement. Similarly, if a Pursuer eats an Evader, this fact is also
reported.

Note that some caution is required with the use of `eoc` triggers. If the condition of a rule
triggered by `eoc` is always satisfied, there will be a continuous looping of Sceptic execution
cycles. Thus in this particular example, the presence of Pursuers will cause the rule which
moves Pursuers randomly:

```
eoc:
        started,
        pursuer(P),
        not(target(P,E)),
        random(X), Dir is 360*X,
        speed(P,Speed)
    => move_agent(P,Speed,Dir), random_move(P).
```

to be executed repeatedly before the locations of the Evaders are logically asserted. There will
be no break in the execution cycle during which the user may assert the appropriate locations.
Thus the extra condition `started` is included in this rule so that nothing will happen until the
user is ready and fires the trigger `initialise`, which results in the assertion of `started` and
the initial locations of the Evaders.

A complete annotated listing of the demonstration follows. Three files are required. The file
`pursuer_evader.sc` contains the Sceptic productions and object rules for the simulation. The
file `polar.pl` is automatically consulted when `pursuer_evader.sc` is loaded, and contains the
Prolog rules which relate the coordinates of one agent to those of another by the distance from
and direction of the first to the second. The third file, `world.pl`, contains the details of the
agents' attributes and must be loaded before the demonstration is initialised.

```
%-----------------------------------------------------------------
% File: pursuer_evader.sc
% Requires sc_default_condition(X,objmatch(X)).
%-----------------------------------------------------------------

/*
 Pursuers have attributes speed, range, reach. They can detect any agent that
 is within range, target on the nearest detected evader, and eat any evaders
 that are within reach. If they detect no evaders they will move about at
 random, trying to search for one.

 Evaders have attributes speed and range. They can detect any agent within
 range. If they detect any pursuers, they will turn and run in the opposite
 direction to the nearest one. If they detect no pusuers, then they will stay
 put, thinking they are safe ...
*/

% Prolog declarations and configurable predicates.

:- sc_op(800,xfy,<-).
:- sc_op(800,xfy,~).
:- sc_op(800,xfy,'->>').

:- sc_dynamic f/1.
:- sc_dynamic '<-'/2.
:- sc_dynamic '->>'/2.

objmatch(not(X)) :- !, \+ f(X).
objmatch(X) :- f(X).


%-----------------------------------------------------------------
/*
```

26

The file world.pl (see later) contains details of the agents attributes.

Load this file before doing anything else. Then the agents need to be placed
in the world by logically asserting their positions. Pursuers will move
about at random even if they detect nothing, so "started" is asserted on
initialisation to tell them when to start looking. Can of course be modified
by editing the coordinates of the locations, or by adding locations for new
agents (but remember to add appropriate attributes to the file "world.pl").

Once "started" has been asserted, and the locations logically asserted, one
or more of the productions triggered by \verb"eoc" will always be satisfied
when this trigger is generated at the end of the execution cycle. The
execution of the program can then only be interrupted by ^C.
*/

```
initialise:
        true
    => assertz(started),
        la(location(p1,0,0)),
        la(location(p2,12,12)),
        la(location(p3,-10,5)),
        la(location(e1,5,5)),
        la(location(e2,-6,6)),
        la(location(e3,15,16)).
```

```
%----------------------------------------------------------------
% Actions:
```

```
move_agent(N,Speed,Dir):
        location(N,OldX,OldY),
        pos_polar(Speed,Dir,OldX,OldY,NewX,NewY)
    => lr(location(N,OldX,OldY)), la(location(N,NewX,NewY)).
```

```
% Moves depend on current beliefs. An evader moves as fast as it can
% in a direction opposite to that of the nearest pursuer.
```

```
% eoc:
        detects(E,P,Dist,Dir),
        evader(E), pursuer(P),
        not([detects(E,P2,Dist2,Dir2), pursuer(P2), Dist2 < Dist]),
        speed(E,Speed),
        NewDir is Dir+180
    => move_agent(E,Speed,NewDir), evading_move(E,P).
```

```
% Pursuers chase after evaders they are targeted on.
```

```
% eoc:
        target(P,E),
        detects(P,E,Dist,Dir),
        speed(P,Speed)
    => move_agent(P,Speed,Dir), pursuing_move(P,E).
```

```
% and if they have no targets, move about at random just looking.
% This is satisfied merely by the presence of pursuers, so the extra
```

```
% condition "started" is required to guard against continuous cycling
% of this rule until the user is ready and asserts "started".

% eoc:
        started,
        pursuer(P),
        not(target(P,E)),
        random(X), Dir is 360*X,
        speed(P,Speed)
    => move_agent(P,Speed,Dir), random_move(P).


% A Pursuer will eat all the Evaders it could eat. One could of course
% have a more complex rule, whereby it selected one, for example, from
% all the possible eatables.

% eoc:
        could_eat(P,E)
    => la(is_eaten(E)), lr(location(E,X,Y)), digestion_report(E,P).


%-------------------------------------------------------------
% These reports are generated by the appropriate actions.

evading_move(E,P):
        location(E,Xe,Ye)
    => write('Evader - '), write(E), write(' detects Pursuer - '),
        write(P), nl, write('and moves to ('), write(Xe), write(','),
        write(Ye), write(').'), nl.

pursuing_move(P,E):
        location(P,X,Y)
    => write('Pursuer - '), write(P), write(' is targeted on Evader - '),
        write(E), nl, write('and moves to ('), write(X), write(','),
        write(Y), write(').'), nl.

random_move(P):
        location(P,X,Y)
    => write('Pursuer - '), write(P), write(' moves randomly to ('),
        write(X), write(','), write(Y), write(').'), nl.

digestion_report(E,P):
        true
    => write('Evader '), write(E), write(' says "Eeek!"'), nl,
        write('Pursuer '), write(P), write(' says "Yummy!!"'), nl.

%-------------------------------------------------------------
% Sceptic productions for logical forward chaining/tms system

% This provides a generic belief maintenance system for all the agents.
% Whenever an agent moves, all agents' beliefs about the world are updated.

% logical assert

la(X):
        not(X)
```

```
      => assertz(f(X)), fw(X).

fw(X):
        Antes->>Conses, member(X,Antes), Antes, member(C,Conses)
     => la(C), assertz(C<-Antes).

fw(X):
        C<-Antes, member(not(X),Antes)
     => retract(C<-Antes), uws(C<-Antes).

fw(X):
        C<-Antes, member(not(L),Antes), member(X,L), L
     => retract(C<-Antes), uws(C<-Antes).

% logical retract

lr(X):
        X
     => retract(f(X)), uw(X).

uw(X):
        C<-Antes, member(X,Antes)
     => retract(C<-Antes), uws(C<-Antes).

uw(X):
        Antes->>Conses, member(not(X),Antes), Antes, member(C,Conses)
     => la(C), assertz(C<-Antes).

uw(X):
        Antes->>Conses, member(not(L),Antes), member(X,L), L, Antes, member(C,Conses)
     => la(C), assertz(C<-Antes).

uws(C<-Antes):
        not((C<-Other, Other \== Antes))
     => lr(C).

%--------------------------------------------------------------
% Object level database

% Agents detect all other agents within range.

    [location(A,HereX,HereY), location(O,Ox,Oy), A \== O,
     pos_polar(Dist,Dir,HereX,HereY,Ox,Oy), range(A,Range), Dist < Range]
 ->> [detects(A,O,Dist,Dir)].

% Pursuers target on the nearest Evader they can see.

    [detects(P,E1,Dist1,Dir1), pursuer(P), evader(E1),
     not([detects(P,E2,Dist2,Dir2), evader(E2), Dist2<Dist1])]
 ->> [target(P,E1)].

% and could eat any Evader that is within reach.

    [detects(P,E,Dist,Dir), pursuer(P), evader(E), reach(P,Reach), Dist < Reach]
```

```
 ->> [could_eat(P,E)].
```

The following file is required which contains the Prolog rules which relate the coordinates of
one agent to those of another by the distance from and direction of the first to the second.

```
%----------------------------------------------------------------
% File: polar.pl
%----------------------------------------------------------------
% Quintus specific.

:- use_module(library(math)).

pos_polar(Dist,Dir,Hx,Hy,Tx,Ty):
        var(Dist), var(Dir),!,
        Xdiff is (Tx-Hx), Ydiff is (Ty-Hy),
        GDist is (Xdiff*Xdiff)+(Ydiff*Ydiff),
        sqrt(GDist,Dist),
        (
         (Xdiff = 0, Radians is 3.14159/2.0);
         (Xdiff \== 0, D is Ydiff/Xdiff, atan(D,Radians))
         ),
        factor(Xdiff,Ydiff,Factor),
        Dir is Factor + ((Radians/3.14159) * 180).

pos_polar(Dist,Dir,Hx,Hy,Tx,Ty) :-
        var(Tx), var(Ty),!,
        IDir is integer(Dir),
        Radians is (IDir/180)*3.14159,
        cos(Radians,CDir),
        sin(Radians,NDir),
        Incx is Dist*CDir,
        Incy is Dist*NDir,
        Tx is Hx+Incx,
        Ty is Hy+Incy.

factor(Xdiff,Ydiff,-180) :-
    Xdiff < 0, Ydiff < 0, !.
factor(Xdiff,Ydiff,180) :-
    Xdiff < 0, Ydiff > 0, !.
factor(_,_,0).
```

The file world.pl contains the details of the agents' attributes. This may be customised by
altering the values of the attributes, and by adding or deleting agents.

```
%----------------------------------------------------------------
% File: world.pl
%----------------------------------------------------------------

pursuer(p1).           pursuer(p2).           pursuer(p3).
speed(p1,10).          speed(p2,10).          speed(p3,5).
range(p1,10).          range(p2,10).          range(p3,8).
reach(p1,3).           reach(p2,3).           reach(p3,2).

evader(e1).            evader(e2).            evader(e3).
```

```
speed(e1,7).              speed(e2,8).              speed(e3,6).
range(e1,9).              range(e2,9).              range(e3,10).
```

A sample run

The agents' attributes and initial locations are as in the listings of the files `pursuer_evader.sc` and `world.pl` given above. It will not be possible to reproduce exactly the program run logged here since a random number generator is used for some of the moves.

As before, comments in {...} do not appear on the screen.

```
Sceptic Versoin 3.0
sc> sc_load('world.pl').
sc> sc_load(pursuer_evader).
...                                           { messages from Prolog whilst    }
...                                           { compiling and consulting files }
sc> initialise.               { No more user interaction after initialisation }
Evader - e1 detects Pursuer - p1
and moves to (9.94974,9.94974).
Evader - e2 detects Pursuer - p3          { Evaders move away from nearest Pursuer }
and moves to (1.76236,7.93537).
Evader - e3 detects Pursuer - p2
and moves to (18.6109,20.7918).
Pursuer - p1 is targeted on Evader - e1
and moves to (7.07106,7.07105).
Pursuer - p3 is targeted on Evader - e2       { Pursuers target on nearest Evader }
and moves to (-5.14853,6.2096).
Pursuer - p2 is targeted on Evader - e3
and moves to (18.0182,19.9863).
Evader - e1 detects Pursuer - p1
and moves to (14.8995,14.8995).
Evader - e2 detects Pursuer - p1                       { The chases continue ... }
and moves to (-6.11609,9.3246).
Evader - e3 detects Pursuer - p2
and moves to (22.2217,25.5836).
Pursuer - p1 is targeted on Evader - e1
and moves to (14.1421,14.1421).
Pursuer - p3 is targeted on Evader - e2
and moves to (-0.297058,7.4192).
Pursuer - p2 is targeted on Evader - e3
and moves to (24.0363,27.9727).
Evader e3 says "Eeek!"                         { Until a Pursuer gets within    }
Pursuer p2 says "Yummy!!"                      { reach of an Evader and eats it }
Evader - e1 detects Pursuer - p1
and moves to (19.8492,19.8492).
Evader - e2 detects Pursuer - p3              { Now only Evaders e1 and e2 left }
and moves to (-13.6802,11.9292).
Pursuer - p1 is targeted on Evader - e1
and moves to (21.2132,21.2131).
Pursuer - p3 is targeted on Evader - e2
and moves to (-5.02464,9.04706).
Pursuer - p2 moves randomly to (26.2858,37.7163).
                                   {p2 detects no Evaders, so moves randomly}
```

```
Evader e1 says "Eeek!"                                          { whilst p1 eats e1}
Pursuer p1 says "Yummy!!"
Pursuer - p1 moves randomly to (14.5218,13.7817).  {e2 is the only Evader left, but}
Pursuer - p2 moves randomly to (26.9832,27.7407).  {no Pursuer can detect it, so    }
Pursuer - p3 moves randomly to (-9.01779,12.0562). {all Pursuers move randomly      }
Evader - e2 detects Pursuer - p3                   {until p3 moves close to e2      }
and moves to (-21.679,11.7896).
Pursuer - p3 is targeted on Evader - e2         { and p3 detects and targets on e2 }
and moves to (-14.0147,11.8816).
Pursuer - p1 moves randomly to (15.2194,23.7574).
Pursuer - p2 moves randomly to (22.2886,36.5701).
Evader - e2 detects Pursuer - p3     { e2 can move a little bit faster than p3 ... }
and moves to (-29.679,11.7896).
Pursuer - p3 is targeted on Evader - e2
and moves to (-19.014,11.7943).                     { ... and try as it might ... }
Pursuer - p1 moves randomly to (20.955,15.5658).
Pursuer - p2 moves randomly to (18.2211,27.4348).
Pursuer - p1 moves randomly to (19.7364,25.4912).
Pursuer - p2 moves randomly to (22.9156,18.6052).
Pursuer - p3 moves randomly to (-19.4497,16.7753).
...                                                { ... p3 can't quite catch it. }
...
```

The random movements continue until one of the faster pursuers targets on e2, chases and eats it. Then, although there are no more Evaders left, the Pursuers will continue to move about at random searching in vain. The user must type ^C to interrupt the program once terminal boredom has set in.

C Command Summary

C.1 Top level commands

`sceptic`	to start Sceptic from UNIX.
`sc_load/1`	load a file.
`vload/1`	load a file, verbose.
`oload/1`	load a file, old format (see Appendix D).
<end-of-file>	escape to Prolog from Sceptic.
`sc/0`	restart Sceptic from Prolog.
`pmode/0`	enter production entry mode (<end-of-file> to return to Sceptic).
`sc_process/1`	execute a Sceptic cycle from within Prolog.
`batch/1`	run a file as if typed to the prompt.
`lp/0, lp/1`	list productions.

C.2 Configuration parameters (normally in sceptic.ini)

`sc_prompt/2`	configures top level and production mode prompts (default values `sc>` and `scp>` respectively).
`sc_file_extension/2`	standard extension for file load (default `.sc`).
`sc_default_condition/2`	specifies default condition to be used when a condition is not recognised as an existing predicate.
`sc_default_action/2`	to be used when an action is not recognised as an existing predicate or trigger.
`sc_default_user_action/2`	to be used when a user command is not recognised as an existing predicate or trigger.

C.3 Debugging commands

C.3.1 Act phase commands

Move:

`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.

Spypoints:

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

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.
3	Track all triggers and predicates.

C.3.2   Rec phase commands

Pause/display points:

New Production	Start of a new production for current trigger.
Instantiation	Complete instantiation of current production.
New Condition	Start of a new condition of current production.
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 Production	Stop at the start of the next production, or end of rec.
i	Instantiation	Stop at the next instantiation, or new production, or end.
s	Success	Stop at the next condition success, instantiation, production or end.
c	Creep	Stop at next point.

Display:

There are no separate display commands at present.

Tracking:

0	No tracking.
1	Track productions and instantiations.
2	Track productions, instantiations and succeeding conditions.
3	Track all points.

D Backwards Compatibility

In most respects other than production syntax, this version is compatible with the previous versions. Some changes are documented below.

D.1 Old functionality (now removed)

D.1.1 Breadth-first expansion

This version expands actions depth first only. This used to be the default behaviour, and was used almost exclusively. The breadth-first option was dropped to simplify development at this stage. We may re-introduce it later if required. The `current_expansion_direction` parameter is obsolete.

D.1.2 Production syntax

The previous syntax did not distinguish the trigger, other than by position. That is, in old syntax the production

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

was written

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

The `eoc` trigger had to be included explicitly.

The normal Sceptic `sc_load` will expect productions in the new format. You can load an old-format file using the command `oload(Filename)`.

At the moment there is no 'save', so you cannot load in old format and save in new format automatically, but you can use the new command `lp` to see the productions in approximately new format (and 'stuff' them into a file?). 'Approximately' here means that some conditions may be wrapped (see section 5.1 Internal Form).

D.1.3 Abbreviations

Previous versions contained a small number of built-in abbreviations. These have been removed, but similar functionality can be obtained by defining trivial predicates, probably in sceptic.ini. See Appendix A for an example.

D.1.4 Renamed configuration predicates

The following configuration predicates have been renamed.

Old Name:	New Name:
pprompt	sc_prompt
default_sceptic_condition	sc_default_condition
default_sceptic_action	sc_default_action
default_user_action	sc_default_user_action

D.2 New functionality (not present in previous releases)

D.2.1 Debugger

Version 3.0 is the first version to have a Sceptic-oriented debugger.

D.2.2 Default filename extension

Version 3.0 is the first version to have a default filename extension.

D.2.3 New commands to the Sceptic prompt

The following are new commands: `lp`, `spy`, `nospy`, `oload`.

E   Prolog-specific Issues and Portability

E.1   Prolog portability

This version of Sceptic consists of three source files:

`sc_main.pl`	main program
`sc_mud.pl`	debugger
`sc_<prolog>.pl`	Prolog-specific code (e.g., `sc_quintus.pl`)

The intention is that all parts which vary between different Prologs should be localised in the `sc_<prolog>.pl` file, 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:

`sc_prolog_version(PrologAtom)`	`PrologAtom` describes the Prolog.
`sc_machine_version(MachineAtom)`	`MachineAtom` describes the machine architecture.
`sc_init_files(List)`	`List` is a list of initialisation files to be tried in order.
`sc_atom_chars(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_abolish(Functor/Arity)`	Remove all trace of the predicate `Functor/Arity`.
`sc_retractall(Term)`	Retract all matching clauses but leave the predicate defined.
`sc_garbage_collect`	Defined to garbage collect or succeed trivially if not applicable.
`sc_trimcore`	Defined to trimcore or succeed trivially if not applicable.
`sc_open(File,Source)`	Open `File` for reading, 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)`	Succeed if `File` exists.
`sc_dynamic(Functor/Arity)`	Simulate a dynamic declaration, so that Sceptic will recognise `Functor/Arity` as a predicate even if no clauses exist. Other uses are Prolog-specific.
`sc_is_predicate(Term)`	Succeed iff `Term` matches a defined predicate, whether compiled, built-in, dynamic, etc.

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.

E.2   Application portability

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, and therefore does not prevent the application writer from using any aspect of that Prolog version, including (possibly unwittingly) non-portable aspects. If such aspects are identified, and if application portability is an issue, an approach analogous to the above can be taken, using file(s) of 'library' predicates, with versions coded for different Prologs.