Working Paper Series
ISSN 1170-487X

Informal Introduction to Starlog

John G. Cleary

Working Paper 93/10

October, 1993

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

Informal Introduction to Starlog

John G. Cleary

Introduction

This report provides an informal and gentle introduction to the logic programming language Starlog and is intended to eventually form the first chapter of a book on Starlog. Like Prolog (a widely known and common logic programming language), Starlog programs consist of sets of Horn clauses. Starlog differs from Prolog in the way it is executed and in the use of logical time to order execution. The style of programming that results tends to be different from Prolog and similar to programs for relational databases.

The Pool and Tuples

Execution of Starlog programs takes place in a **pool of tuples**. Each **tuple** is a single discrete piece of information, for example, `student('John',mathematics).` Tuples are the basic unit of information in relational databases and perform the same function as records or structures in applicative languages. The pool is dynamic with tuples being placed into it and later withdrawn. To show that a tuple is present in the pool at a particular time a number, its **timestamp**, is attached to it, for example, `student('John',mathematics)@23`. To show that a tuple is present over a range of time it can be written as:

```
student('John',mathematics)@T :- T≥20, T≤24.
```

This says that the student information about John is present in the pool from time 20 till time 24.

The letter `T` here is a **logical variable** usually referred to just as a **variable**. Variables are distinguished from other things such as the name of tuples (`student`) or constants (`24` or `mathematics`) by starting with a capital letter. One way of thinking about variables is that they are place holders that can take on any possible value. Thus `T` above might take on the value 1000 or -1 or even `mathematics` or `foo(bar)`. The tuple above can then be read as meaning "`student('John',mathematics)@T` is present in the pool for those values of `T` where `T` is greater than or equal to 20 and `T` is less than or equal to 24". Thus `student('John',mathematics)@1000` is not present in the pool because 1000≤24 is false but `student('John',mathematics)@20`, `student('John',mathematics)@21` etc. are in the pool. Such tuples which contain **constraints** on the variables in the tuple are referred to as **constrained tuples** as contrasted with **simple tuples**.

Execution and Rules

A **program** in Starlog consists of a number of **rules**. Each rule is like a constrained tuple in that it contains conditions that must be satisfied when the rule is to generate new tuples. Execution of programs in Starlog is a cycle where rules whose conditions are satisfied by tuples in the pool are executed which in turn puts more tuples back into the pool and so on ... The program below is one which will place the tuple `even` into the pool at the times, 0,2,4, ...

```
Program 1.1:
%Generate all even numbers

even@0.
even@T :- even@S, T is S+2.
```

The first rule, "`even@0.`" says that the tuple should be placed in the pool at time 0. This is a special case as the rule contains no conditions at all. The second rule says that if there is a tuple "`even@S`" in the pool (or to be more precise `even` is in the pool at time `S`) then compute `T` as `S+2` and put `even` into the pool at time `T`. The result is that `even` appears in the pool at times 0, 2, 4, ... but not at times 1, 3, 5, ...

There is an important principle in Starlog (the **principle of causality**) that tuples can only be inserted in the future (or at the current time). It is not possible to generate a tuple in the past (there are plenty of science fiction books about that will

explain the dangers of doing that). One other small restriction on times is that they must all be positive (greater than or equal to 0). This restriction is necessary so that Starlog knows at what time to start execution.

Output (`write`)

In order to see what tuples have been generated by the program it is necessary to somehow take them from the pool and display them on a terminal or print them on paper. Starlog does this via a special class of `write` tuples. The Starlog system recognizes these and ensures that they are displayed on your terminal (in a Unix system they are written to stdout). The program above can be modified as follows:

```
Program 1.2:
%Generate all even numbers and print them

write(T):- event@T.

even@0
even@T :- even@S, T is S+1.

Output from program 1.2
0
2
4
...
```

Each `even` tuple as it is placed in the pool fires the rule "`write(T):- event@T.`" which places a `write` tuple in the pool at the same time, this is seen by the Starlog system and causes the tuple to be printed.

Hamming Problem

So far we have seen only a few very simple constructs in Starlog. Even these are sufficient to construct a great many interesting programs. I will illustrate this with the Hamming problem.

The problem is to generate all numbers of the form $2^a3^b5^c$ where a,b,c ≥ 0 and the numbers are to be generated in order without duplicates. Thus $10=2^13^05^1$ is in the sequence as is $1 = 2^03^05^0$, but 14 is not.

As in the even numbers example above the Hamming numbers will be generated in time order with the tuple `hamming` being placed in the pool at the appropriate times. The basic idea of the algorithm is that if some number, say `I`, is a Hamming number then `2*I`, `3*I` and `5*I` will also be Hamming numbers. In fact, given that 1 is a Hamming number this rule on its own is sufficient to generate all the rest of the Hamming numbers. Translated into Starlog these two rules give the following program:

```
Program 1.3
%Generate all the Hamming numbers and print them
%(see program 1.4 for a better written version)

write(T) :- hamming@T.

hamming@1.
hamming@N :- hamming@I, N is I*2.
hamming@N :- hamming@I, N is I*3.
hamming@N :- hamming@I, N is I*5.
```

It is worth following the execution of this program to see what happens. Beginning at time 1 `hamming` is placed in the pool (I will ignore all the write tuples so as not to clutter up the example too much). This immediately generates three new tuples, `hamming@2`, `hamming@3`, and `hamming@5`. These are placed by the system in the pools which will be used at these future times. Now Starlog advances to time 2 which is the

next time where there is something in the pool. `hamming` is in the pool and it generates new tuples at times 4, 6 and 10. Again at time 3 `hamming` is in the pool and tuples are generated for times 6, 9 and 15. The execution continues in this way forever (or at least until the user runs out of patience). A couple of interesting events occur along the way.

The first is at time 6. The `hamming` tuple is generated for this time twice. Once from time 2 (2*3) and again at time 3 (3*2). However, the tuple pool is like a mathematical set, so that only one copy of a tuple is ever stored. Thus the two different tuples are merged into one. (This process of removing duplicates is very important in this example as otherwise the number of tuples at each time would grow exponentially with time rather than there being at most one tuple.) The second interesting event is at time 7. This is the first time at which no `hamming` tuple is present. The Starlog system skips past time 7 and moves to time 8.

The style of the Hamming program above is unsatisfactory. The same rule (take a `hamming` tuple and multiply its time) is repeated three times. As programmers we are always on the look out for some means of reducing the size of programs. This can be done here by introducing `multiply` tuples which list the multipliers 2,3 and 5. They are multipliers at all times so the rules for introducing these tuples take the form of immutable facts:

```
multiplier(2)@T.
multiplier(3)@T.
multiplier(5)@T.
```

(the `T` variables can take on any value from 0 upward).

The three multiplying rules above can then be replaced by one more general one that says that a new `hamming` tuple is to be generated from an old one by multiplying by any of the values in the `multiplier` tuples. The program compacted in this way becomes;

Program 1.4a

```
%Generate all the Hamming numbers and print them
%(this version avoids needlessly duplicating the second rule)

write(T) :- hamming@T.

hamming@1.
hamming@N :- multiplier(J)@T, hamming@I, N is I*J.

multiplier(2)@T.
multiplier(3)@T.
multiplier(5)@T.
```

Such immutable facts are so common that an alternative way of expressing them is provided. Both the facts and the conditions in rules that reference them can be written without any reference to time. The program after this further simplification becomes:

Program 1.4b

```
%Generate all the Hamming numbers and print them
%(this version avoids needlessly duplicating the second rule)

write(T) :- hamming@T.

hamming@1.
hamming@N :- multiplier(J), hamming@I, N is I*J.

multiplier(2).
multiplier(3).
multiplier(5).
```

Abbreviations such as these are referred to as **timeless rules** or **facts**.

Fibonacci Numbers

The Fibonacci numbers are the following sequence:

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

where each number is the sum of the preceding two numbers. To fully specify the sequence it must start with the two numbers 1, 1. A good way of representing such a sequence in Starlog is to generate the numbers at successive times. So the first two 1s are generated at time 0 and 1 respectively, 2 at time 2, the 3 at time 3 and the 5 at time 4. (Choosing the right representation in this way is an art form and hopefully the reader will be proficient at it before the end of this book). The tuples that will be generated look like this:

```
fibonacci(1)@0.
fibonacci(1)@1.
fibonacci(2)@2.
fibonacci(3)@3.
fibonacci(5)@4.
fibonacci(8)@5.
...
```

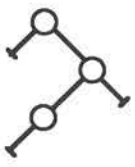
For the program we have three rules. The first two state that the Fibonacci numbers at times 0 and 1 are 1. The third says that the Fibonacci number at a particular time is the sum of the two Fibonacci numbers at the immediately preceding times.

```
Program 1.5.   Compute the Fibonacci numbers.

fibonacci(1)@0.
fibonacci(1))@1.
fibonacci(W)@T :- fibonacci(U)@R, S is R+1,
                  fibonacci(V)@S, T is S+1,
                  W is U+V.
```

Data Structures and Trees

So far we have seen a number of types of tuples; simple tuples such as `even@2`, and tuples with a single associated value such as `fibonacci(8)@5`. The number 8 here is a **parameter** of the tuple. Tuples can have any number of parameters which allows very complex tuples and programs to be built up. A parameter in a tuple can be a variable (e.g. `T`), an integer (e.g. `8`), as well as more complex objects called **functors**. The word functor is a fancy term that means roughly "the name of a record". Functors are like tuples in that they have a name which starts with a lower case letter and can themselves have one or more parameters. This last allows arbitrarily complex structures to be built up. It is not often necessary to do this in Starlog, as typically, complexity is built up by adding more simple tuples to the pool rather than making individual tuples more complicated. Sometimes though it is necessary. As an exercise in constructing complex data structures from functors consider the problem of generating all binary trees. A binary tree will be represented by two functors `leaf` and `tree`. `leaf` will be a functor with no parameters (sometimes referred to as an **atom**) and will represent a leaf node of the tree (for a useful tree it should have some parameters to carry information). `tree` will be a functor with two parameters the first being the left subtree of the tree and the second the right subtree. Thus `tree(leaf,tree(tree(leaf,leaf),leaf))` represents the tree:



Consider the problem of generating all possible trees. The rules for this are simple: first `leaf` is a tree and second from any two trees a new tree can be constructed by making them the left and right subtrees respectively. The trees will be packaged in a tuple called `is_a_tree`. At first sight it is not clear what this has got to do with time and

the following two Starlog like rules (but without any time!) might be thought to do the job.

```
Program: 1.6
%Generate all binary trees.
%Warning this is not valid Starlog.

is_a_tree(leaf).
is_a_tree(tree(Left,Right)) :-
      is_a_tree(Left), is_a_tree(Right).
```

The difficulty with this program is that it does not say what order the trees will be generated. Why is this a problem? The trees must be generated in some order (this is after all computing and not mathematics that we are doing). But if the wrong order is chosen then there is no guarantee that any particular tree will be generated. For example, all the trees with an empty left subtree might be generated first and then the rest. But there are an infinite number of trees with an empty left subtree so we never get around to the rest. There are orders for generating the trees so that every tree is eventually generated. For example, if all trees with depth 1 are generated first, then those with depth 2 and so on. There are only a finite number of trees of any particular depth and any given tree has a finite depth so eventually every tree will be constructed. It is in this ordering of computation that time in Starlog becomes important. Depth is a good parameter to use for ordering as the left and right subtree always have depths smaller than the whole tree. This is just a restatement of the causality principle that the time of a new tuple must not be earlier than the times of the tuples in the rule that generates it. Using depth as a time parameter requires setting the time (depth) of the newly generated tree to the maximum of the depth of the two subtrees plus 1 and gives the following (correct) Starlog program:

```
Program:  1.7
%Generate all binary trees.
%Use depth of tree to order tuple generation

is_a_tree(leaf)@0.
is_a_tree(tree(Left,Right))@N :-
      is_a_tree(Left)@L,
      is_a_tree(Right)@M,
      N is max(L,M)+1.
```

Figure 1.1 shows the successive contents of the pool and diagrams of the corresponding trees.

Choosing a time ordering for a program such as this is an art rather than an exact science and there may be many ways to do it. For example, the following modified program uses the number of nodes in the tree as the 'time' parameter:

```
Program:  1.8
%Generate all binary trees.
%Use number of nodes in tree to order tuple generation

is_a_tree(leaf)@0.
is_a_tree(tree(Left,Right))@N :-
      is_a_tree(Left)@L,
      is_a_tree(Right)@M,
      N is L+M+1.
```

The reader may enjoy working out what happens with the following mystery definition of time:

```
Program:  1.9
%Generate all binary trees.
%Use mystery definition to order tuple generation

is_a_tree(leaf)@0.
is_a_tree(tree(Left,Right))@N :-
      is_a_tree(Left)@L,
      is_a_tree(Right)@M,
      N is (L+M)*(L+M+1)/2 + M.
```

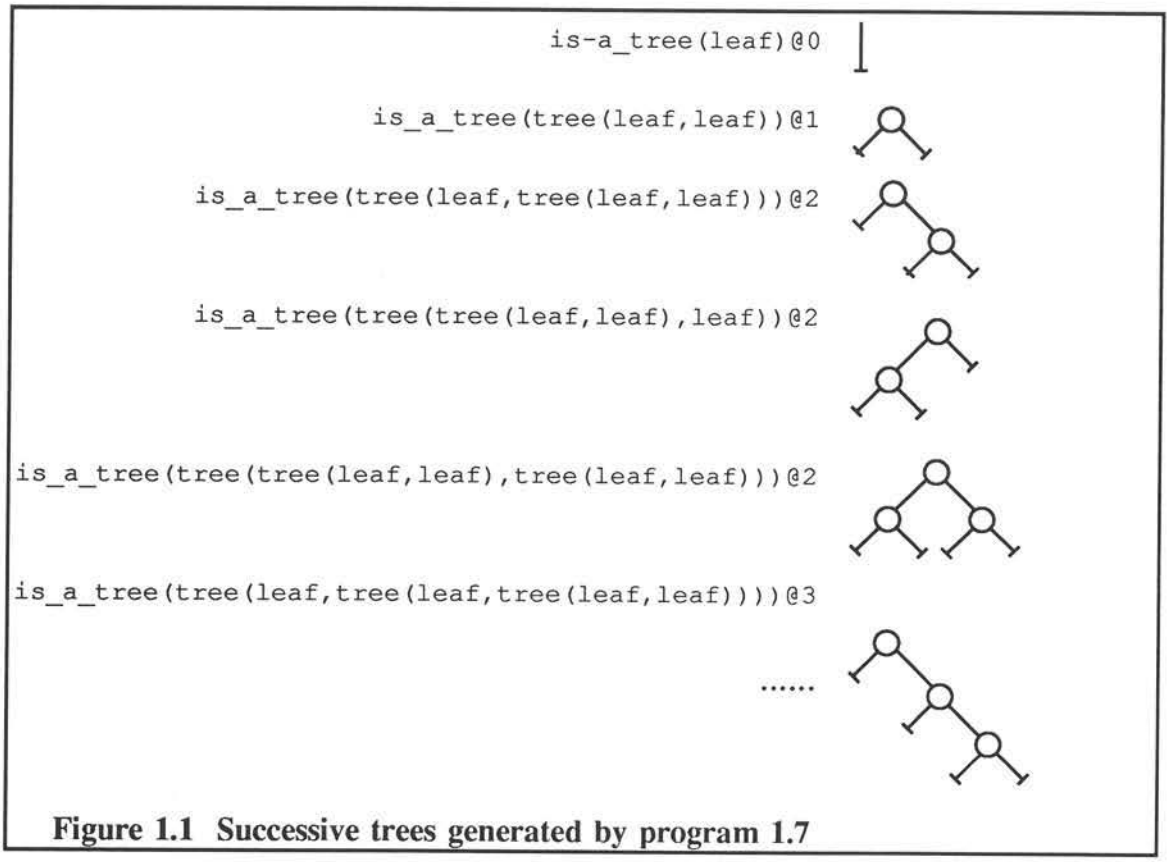


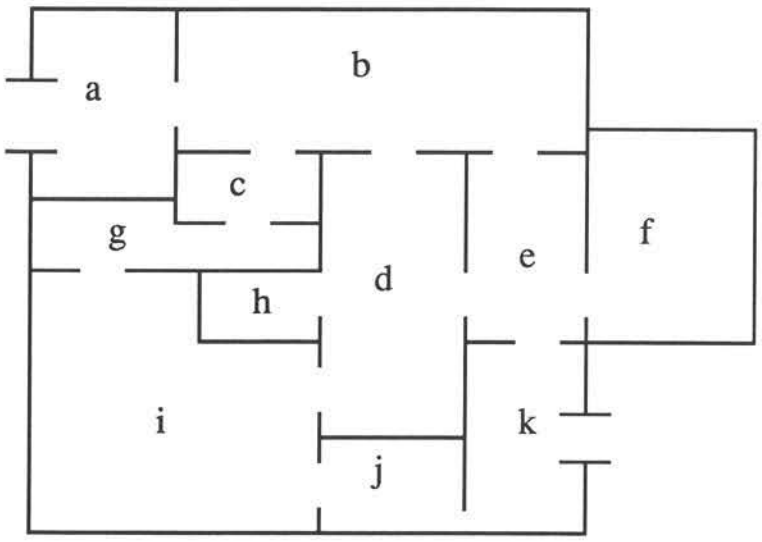
Figure 1.1 Successive trees generated by program 1.7

Negation

Searching

The next example will show how to do simple searching in Starlog. I have chosen this example for two reasons: first because searching is an interesting and important topic in computing and second because it will require that tuples be generated when other tuples are **not** present in the pool. All the rules so far have been facts or have generated a new tuple as the result of (one or more) other tuples in the pool. Sometimes this is not sufficient and it is necessary to be able to inhibit the generation of a new tuple or to generate a new tuple when something is not present.

The search problem I will use is to find a way through a house from the entrance to the exit. There are doors between rooms and the trick is to find some way through the maze of rooms and doors. To write a program for this, some way of representing the layout of the house must be devised as well as some way of walking through the rooms. The data structures for this might be represented in the same way as the binary trees in the previous section. However, it is more effective to use timeless facts of the form `door(a,b)` which say that there is a door from room a to room b. Because there may be one way doors (such as turnstiles or trapdoors) a door from room a to room b does not necessarily imply a door back from room b to room a. In a similar style the entrance room and the exit room are described by timeless `entrance` and `exit` tuples. Figure 1.2 shows an example house and its description.




```
entrance(a).

exit(k).

door(a,b).          door(b,a).
door(b,c).          door(c,b).
door(b,d).          door(d,b).
door(b,e).          door(e,b).
door(c,g).          door(g,c).
door(d,e).          door(e,d).
door(d,h).          door(h,d).
door(d,i).          door(i,d).
door(e,f).          door(f,e).
door(e,k).          door(k,e).
door(g,i).          door(i,g).
door(i,j).          door(j,i).
door(j,k).          door(k,j).
```

Figure 1.2 House Searched by Program 1.10

The search for a route will be done using tuples of the form `in(R)@Time` which say that at time `Time` it is possible for an intrepid explorer to be in room `R`. Time here is chosen to advance by one unit as a new room is entered. Note that at any one time there may be many `in(R)` tuples, as all possible ways of moving through the house are simulated at once. A first naive version of a search program follows:

```
Program  1.10
%Search from entrance to exit
%Warning this program does not terminate

%Start at the entrance
in(Room)@0 :- entrance(Room).

%In one time unit move to all adjacent rooms
in(Room)@T :- in(Old_Room)@S, T is S+1, door(Old_Room,Room).
```

Why is this program naive? Well for one thing it never stops, as it keeps repeatedly visiting the same rooms (Figure 1.3 shows the tuples generated). To prevent this the program should (at least) stop when the exit is found. What is needed is a rule which says "if the exit has been found then stop generating new `in` tuples" or to be more precise the following condition needs to be added to the second rule: "if at some previous time there has been an `in` tuple for the exit then do **not** generate a new `in` tuple". This is a new type of rule different from any we have seen before, for now something must not be done if a particular tuple is present. Starlog allows this to be done using **negations** in the body of a rule of the form `not(tuple@T)`. In this case the negation is formulated in two steps. First the condition "if at some previous time there has been an `in` tuple for the exit" is encapsulated in the tuple `done@T` using the rule:

```
done@T :- T>T0, in(Room)@T0, exit(Room).
```

This will generate a constrained tuple `done` which is true after time `T0`.

The second rule for `in` is then modified to :

```
in(Room)@T :-
      in(Old_Room)@S,
      T is S+1, not(done@T),
      door(Old_Room,Room).
```

This results in the following program:

```
Program  1.11
%Search from entrance to exit taking care to stop
%Warning: if the exit cannot be reached this may still not
%terminate

%Start at the entrance
in(Room)@0 :- entrance(Room).

%In one time unit move to all adjacent rooms
in(Room)@T :-
      in(Old_Room)@S,
      T is S+1, not(done@T),
      door(Old_Room,Room).

%Remember to stop after finding the exit
done@T :- exit(Room), in(Room)@T0, T>T0.
```

Figure 1.3 shows that this does the trick and execution does indeed stop after the exit is reached.

Unfortunately while it works for this example,the program is still not satisfactory as it visits the same room multiple times and will not terminate if there is no route from the entrance to the exit. For example, `in(b)` is generated at times 1, 2, and 3. If there are N rooms in the house the search may generate $O(N^2)$ tuples (and take $O(N^2)$ time) whereas O(N) algorithms are possible. Also in any house where it is possible to go from a room and get back to the room and where the exit cannot be reached the tuples for the rooms will be repeatedly generated and the program will not terminate.

Once a room has been visited there should be no need to go on searching for more routes to the same room. This is done by modifying the meaning of `in(Room)@T` to be "`Room` has been visited at time `T` or earlier"[1]. This requires the rules for `in` to be modified so that the condition `T>S` is used rather than `T is S+1`. This gives the following program which always terminates and only generates a single conditional tuple for each room visited:

```
Program  1.12
%Search from entrance to exit taking care not to enter any room
%more than once.

%Start at the entrance
in(Room)@T :- T≥0, entrance(Room).

%In one time unit move to all adjacent rooms that have not been
%visited before
in(Room)@T :- T>S
      in(Old_Room)@S,
      not(done@S),
      door(Old_Room,Room).

%Remember to stop after finding the exit
done@T :- exit(Room), in(Room)@T0, T>T0.
```

Figure 1.3 shows that this program generates a smaller number of `in` tuples than Program 1.11. Although each tuple is conditional and present in an infinite number of pools the Starlog system need only do work once to generate each of them. The program terminates whether or not the exit can be reached.

---

[1]Thanks to Bill Rogers for pointing out this elegant solution.

Despite this gentle introduction to negation it is difficult to overstate its importance. Without it there are a great many programs that could not be written in Starlog. Unfortunately, dealing with its correct implementation and semantics will occupy most of the rest of this book! To illustrate this point the following sections show the importance of negation by using it to implement assignment and then show some of the difficulties of implementation and meaning with a couple of rather tricky little toy programs.

Program 1.10
never terminates

```
in(a)@0.

in(b)@1.

in(a)@2.
in(c)@2.
in(d)@2.
in(e)@2.

in(b)@3.
in(d)@3.
in(e)@3
in(g)@3.
in(h)@3.
in(i)@3.
in(k)@3.
in(f)@3.

in(a)@4.
in(b)@4.
in(c)@4.
in(d)@4.
in(e)@4
in(f)@4
in(g)@4.
in(h)@4.
in(i)@4.
in(k)@4.
in(j)@4

in(a)@5.
in(b)@5.
in(c)@5.
in(d)@5.
in(e)@5
in(f)@5.
in(g)@5.
in(h)@5.
in(i)@5.
in(k)@5.
in(j)@5.
.....
```

Program 1.11
sometimes terminates

```
in(a)@0.

in(b)@1.

in(a)@2.
in(c)@2.
in(d)@2.
in(e)@2.

in(b)@3.
in(d)@3.
in(e)@3
in(g)@3.
in(h)@3.
in(i)@3.
in(k)@3.
in(f)@3.

done@T:- T≥4.
```

Program 1.12
visit each room once
always terminates

```
in(a)@T:- T≥0.

in(b)@T:- T≥1.

in(c)@T:- T≥2.
in(d)@T:- T≥2.
in(e)@T:- T≥2.

in(f)@T:- T≥3.
in(g)@T:- T≥3.
in(h)@T:- T≥3.
in(i)@T:- T≥3.
in(k)@T:- T≥3.

done@T:- T≥4.
```

See individul programs in text for details.

Fig 1.3 Contents of Pool for Different Search Programs

Assignment and Databases

Assignment is a familiar operation in many programming languages. For example we might say:

```
AVAR := 42;
```

This means that after executing the statement the location AVAR will contain the value 42 independent of whatever value it had before the execution of the statement. That is the contents of AVAR changes from some old value to 42. A similar process occurs in relational databases where tuples in a relation can be deleted or new ones inserted. The meaning of assignment clearly involves a before and after ordering of execution. Starlog implements assignment by using its temporal ordering so that a location has one value over a range of time and then for later times it has another value. To illustrate this I will use as an example a very simple database.

This database has two types of objects: keys and values. The keys are the letters x, y and z and the values are the letters a, b, c. Each key has associated with it exactly one value. I will say that a key has a particular value. The database is implemented in Starlog using tuples of the form value(Key,Value). For example value(x,a) says that "the key x has the value a". Thus there should never be more than one value tuple in the pool at any one time which has the same key.

To assign a new value to a key an assign(Key,Value) tuple will be used. For example assign(x,b)@10 says that from time 10 onward the key x will have the value b independent of any value it had before. Also if x had no value (there was no value(x,_) tuple in the pool) before, then a new one will be created. There is an arbitrary choice that needs to be made at this point. The assignment assign(x,b)@10 could reasonably give x it's value at time 10 and onward or at time 11 and onward. The convention used here is that the assignment will not take effect till the next time. That is, x has the value b from time 11 onward. Figure 1.4 shows what should happen in the pool for an example set of assign tuples.

assign(x,a)@1	value(x,a)@T ¨ T≥2, T≤5.
assign(y,b)@3	value(y,b)@T ¨ T≥4.
assign(x,c)@5	value(x,c)@T ¨ T≥6.

Figure 1.4. Example of Assignment in a Database.
Assignments are made to x and y at the times 1, 3 and 5. This leads to the key x having the value a from time 2 to 5 inclusive and then a value c from time 6 onward. The key y has no value until time 4 and from there onward will have the value b.

The problem now is to construct a program that takes assign tuples as inputs and generates value tuples as outputs. The program will be an implementation of the statement: "a key K has a value V from the time it is assigned until the next value is assigned to it". I have underlined the key part of this statement that is going to require the use of negation. To start gently the first part that is not underlined can be written in Starlog as:

```
value(K,V)@T :- assign(K,V)@T0, T>T0.    %Warning:incomplete code
```

That is each assign tuple generates a new value tuple which is present from the time following (T>T0) the time of the assignment. The problem with this rule as it stands is that the value tuples do not stop at the next assignment. So in the example of Fig. 1.4 from time 6 onward both the tuples value(x,a) and value(x,c) would be present in the pool. So the rule should have a condition that says "stop being true when the next

assignment occurs to the same key". To capture this stopping condition we generate tuples of the form `delete(K,T0)@T` which mean "at time `T` there has been an earlier assignment to the key `K` which occurred after time `T0`". In the form of a rule this is:

```
delete(K,T0)@T :- assign(K,_)@T1, T>T1, T1>T0.
```

Each `assign` tuple thus generates a `delete` tuple.which will have a definite value for `T1` and where `T` will be constrained to later times and `T0` to earlier times. `delete` will be used to make all assignments which occurred earlier than `T1` false after time `T1`.

The rule for generating `value` tuples can be written using `assign` and `delete` tuples as follows:

```
value(K,V)@T :- assign(K,V)@T0, T>T0, not(delete(K,T0)@T).
```

That is, "a key `K` has a value `V` from the time it is assigned until a `delete` tuple occurs for the same key". The complete program is thus:

Program 1.13

```
%Assign values in the value tuple using the assign tuple
value(K,V)@T :- assign(K,V)@T0, T>T0, not(delete(K,T0)@T).

%Delete tuples are used to terminate old value tuples
delete(K,T0)@T :- assign(K,_)@T1, T>T1, T1>T0.
```

Another way of understanding this program is to see what happens when it is used on the example of Figure 1.4. Each of the tuples generated is labelled with a number for easy reference.

At time 1 the tuple `assign(x,a)` appears in the pool. This generates the conditional tuple

```
value(x,a)@T:- T≥2, not(delete(x,1)@T).                    (1)
```

as well as the tuple

```
delete(x,T0)@T:- T≥2, T0≤0.                                (2)
```

(in fact, this tuple will never be used as there will be no corresponding `value(x,_)` tuples to terminate).

The `value` tuple is a conditional tuple true from time 2 onward. It is conditional because it will fail at any time after 2 that a matching `delete` tuple appears in the pool.. Starlog deals with this by waiting till time 2 and then checking if any `delete` tuples have appeared up to that time. Thereafter the tuple will be rechecked whenever a new `delete` tuple for the key `x` appears. Continuing execution, at time 3 two more tuples are generated for the key `y`:

```
value(y,b)@T:- T≥4, not(delete(x,3)@T).                    (3)
delete(y,T0)@T:- T≥4, T0≤2.                                (4)
```

These do not match with tuples for the key `x` and so nothing further happens till time 5 when two more tuples are generated:

```
value(x,c)@T:- T≥6, not(delete(x,5)@T).                    (5)
delete(x,T0)@T:- T≥6, T0≤4.                                (6)
```

Tuple number 6 matches with the `delete(x,1)@T` condition inside the negation in tuple number (1). This stops the tuple from being true from time 6 onward (`T≥6`). Thus (1) can be modified to the following form without a negation:

```
value(x,a)@T:- T≥2, T≤5.                                   (1')
```

No more tuples are generated after time 5 (as there are no more `assign`s) thus tuples (3) and (5) can be rewritten to be:

```
value(y,b)@T:- T≥4.                                        (3')
value(x,c)@T:- T≥6.                                        (5')
```

Integrity Constraints - Checking for Errors

Given the way that the `assign` and `value` tuples have been written above, care must be taken that no key is assigned two different values at the same time - if this happens then both these values will be present at one time in the database violating the

rule that each key could should be associated with just one value. One way to deal with this problem is to check for cases of multiple assignment and print a warning message whenever it occurs. Because Starlog allows any rule to access any tuple in the pool this checking can be done simply without modifying the programs above. The following program monitors the pool and prints a message whenever a key is assigned more than one value at the same time:

```
Program  1.14
%Print a warning message whenever a key is assigned two values
%at the same time
write(['Key',K,'assigned two values at time',T])@T :-
      assign(K,V1)@T,
      assign(K,V2)@T,
      not(V1=V2).
```

The following even more paranoid program checks that no key ever ends up with two different values (just in case we wrote our program incorrectly) and something slips past the check above:

```
Program  1.15
%Print a warning message whenever a key has two values at the
%same time
write(['Key',K,has two values at time',T'])@T :-
      value(K,V1)@T,
      value(K,V2)@T,
      not(V1=V2).
```

Checks such as these are referred to as integrity constraints and are commonly used in databases to check the continuing correctness of a database. They form a powerful tool for debugging Starlog programs. The next section looks at some other ways of debugging Starlog programs.

Debugging and `input`

As in any programming language it is necessary to test and debug Starlog programs. In the last two programs I showed how to check the usage and correctness of `assign` and `value`. As well, Starlog provides a number of very useful general tools to aid in debugging. The first is the following "Universal Printing Program", this prints out every tuple that enters the pool. It also uses negation to ensure that it does not print out its own `write` tuples (this can lead to an infinite number of messages being written):

```
Program 1.16 The Universal Printing Program
%Print all tuples in the pool (except write tuples!!)
write(G@T)@T :- G@T, not(G=write(_)).
```

In substantial programs the pool may be so large that it is not effective to print all of it but you still want access to it. The following program uses input tuples to provide a general way of querying the pool, even at previous times. Each Starlog system provides a way of injecting tuples into the correct pool interactively (see the manual for details). These tuples appear in the form: `input(thing@42)`, where the user typed "`thing`" at time 42. (The time is added by the Starlog system which slips the input into the current pool as soon as it receives it from the host operating system). The first version of the program receives the input matches it against the current pool and prints a message if anything matches:

Program 1.17

```
%Print requested tuples in the current pool
write(G)@T :- input(G)@T, G@T.
```

For example if the user typed `value(K,V)` at time 4 in the assignment program 1.13 and Fig 1.4 the following would be printed:

```
value(x,a)
value(y,b)
```

An even more sophisticated program than 1.17 is possible. This is done by allowing the user to type in a time with the query which prints out what was in earlier pools. The following program assumes that the user will type inputs of the form `G@S` which will request all tuples matching `G` in the pool at time `S`.

Program 1.18 Universal Interactive Debugger

```
%Print requested tuples in the pool at the specified time
write(G@S)@T :- input(G@S)@T, G@S.
```

So typing in `value(x,K)@3` at time 6 will print:

```
value(x,a)@3
```

If a time in the future is entered then the system will wait till that time before printing. So if `value(x,K)@6` is input at time 3 the system will wait till time 6 and then print:

```
value(x,c)@6
```

It is possible to use the program to print all instances of a term at any time and for `write` to print constrained tuples. For example, the input `value(x,K)@T` when input at time 6 gives the following output:

```
value(x,a)@T :- T≥2, T≤5.
value(x,c)@T :- T≥6.
```

Printing constrained tuples in this way can be a little tricky as the negation constraints change with time. The system does its best to put out immediate information and then later to update it. So if the same input had been inserted at time 3 the following would have been printed:

```
value(x,a)@T:- T≥2, not(delete(x,1)@T)   %Printed at time 3
value(x,a)@T :- T≥2, T≤5.                %Printed at time 5
value(x,c)@T :- T≥6.                     %Printed at time 6
```

Clearly this debugger is very powerful. There is one small penalty that must be paid for this. Normally Starlog can garbage collect tuples when they are no longer needed. However, including code such as the Universal Interactive Debugger requires that all tuples be kept in case someone wants to print them later. Debugging large programs in this way may be constrained by the available memory.

What Can Go Wrong With Negation

There is an old joke about the computer onboard an aircraft that announces to the passengers: "There is no pilot on board this aircraft. Do not worry as you are in the hands of the latest computer technology and nothing can go wrong, can go wrong, can go wrong, ...".

In every programming language there are particular ways that programs can go wrong. In traditional applicative languages with assignments these include infinite loops, not assigning variables values values before they are used, addressing outside the range of an array, dividing by zero, and so on. Starlog is free from some of these but there are a few particular ones associated with negation that users need to be aware of. In particular there is an insidious class of programs like the following:

Program 1.19

```
%A very bad program (who knows what it means)
creteans_are_liars@T :- not(creteans_are_liars@T).
```

This program says that "the tuple `creteans_are_liars` should be in the pool if it is not in the pool". As noticed some 2500 years ago it is not possible to get this right. Starlog neatly skips this problem by saying that you are not allowed to write programs like this. The rule is that you cannot write programs where any tuple depends on its negation to be present in the pool. This forbids more subtle versions of the program that hide the negation dependency:

Program 1.20

```
%Another more subtle illogical program
creteans_are_liars@T :- not(creteans_are_beautiful@T).
creteans_are_beautiful@T :- creteans_are_liars@T.
```

Starlog will do its best to tell you about such programs. Some it may be able to warn you about when the program is loaded. Others cannot be detected until the program is executed.

Unfortunately, the distinctions between valid programs with negations and ones which violate this rule can be quite subtle. For example the program below has a tuple depending upon its negation at an earlier time. Although at first glance it may seem heir to the paradox above it is in fact OK because the negation refers to a tuple at an earlier time not at the current time. The program generates all the even numbers in a rather tricky way by saying that every number is even if the immediately preceding number was not even. The resulting is:

Program 1.21

```
%even numbers generated using negation
%It may seem paradoxical but it is not
even@T:- T is S+1, not(even@S).
```

This program executes as follows. It starts at time 0, there are no `even` tuples at time -1 (all times are positive). So the negation cannot succeed and so at time 0 `even` must be in the pool. But `even@0` prevents the rule from succeeding with `S=0` and `T=1`. So there is no even tuple at time 1. But then at time 2 there is no tuple at time 1 and `even@2` is placed into the pool and so on.