Temporal Logic Programming in *Tokio*

Albert C. Esterline
Danny Kilis

University of Minnesota

DRAFT

CHAPTER 1

Introduction

1.1 Theoretical Background

CHAPTER 2

Temporal Variables

2.1 Bindings of Temporal Variables: `$t`-lists

A temporal variable is bound to a Prolog structure similar to a list, which we shall call a $t-list. The head of a $t-list is the value of the corresponding variable for the current time. Its tail may again be a $t-list; thus the $t-list contains the values of the variable for all future times, as far as these values are known. As execution of a Tokio program proceeds from one time to the next, successive elements of the $t-list for a variable become the current value of the variable until, perhaps, the last element of the $t-list is reached, and then this last element is the value for the variable from that time on. Furthermore, as execution proceeds, a $t-list may be extended into the future if the last part of a $t-list is uninstantiated.

A $t-list is a structure with two arguments and with $t as its functor; the second argument may be uninstantiated or may itself be a $t-list. For example, suppose the following is associated with the temporal variable `I`:

```
$t(1, $t(2, _))                    .........(1)
```

This indicates that the current value of `I` is `1`, the value of `I` for the next time is `2`, and the value of `I` for later times is undetermined. This $t-list may be compared with the Prolog (and hence Tokio) list

```
[1, 2]                             .........(2.1)
```

Now recall that the primitive list functor in Prolog is `.`, which takes two arguments; the second normally is either itself a list or the special atom [ ]. Thus **(2.1)** may be represented as

```
.(1, .( 2, []))                    .........(2.2)
```

This looks very much like **(1)** with $t replaced by `.`, except the last part is `[]` instead of `_`.

In fact, it is quite easy in Prolog to create and to use lists with uninstantiated last parts, that is, tails. Thus the list made with `.` as follows

```
.(1, .(2, _))                      .........(3.1)
```

is written in list notation as

```
[1, 2 | _]                         .........(3.2)
```

Now consider the usual `member` predicate in Prolog:

```
member(E, [E|_]).
member(E, [_|T]) :-  member(E,T).
```

And consider the query

```
?- member(3,[1,2]).
```

or, equivalently,

```
?- A = [1,2], member(3,A).
```

where the list is that shown in **(2.1)** and **(2.2)**. This finds `3 ≠ 1`, and so tries `member(3, [2])`. It then finds `3 ≠ 2`, and tries `member(3, [])`. The latter fails, so all goals depending on it, back to the original, fail. Next consider the query

```
?- A = [1, 2|_], member(3, A)
```

where the list is that shown in **(3.1)** and **(3.2)**. This finds `3 ≠ 1`, then `3 ≠ 2`, and then tries `member(3, [_])`. But then the first, non-recursive clause in the definition of `member` succeeds: `[_]` is unified with `[E|_]`, but `E` is unified with `3`; thus `[_]` is unified with `[3|_]`. The goals depending on this subgoal thus succeed, and we get, at the top level, `A=[1,2,3 | _]`. Thus this definition of `member`, when given an instantiated first argument and a second argument that is a list with uninstantiated tail, succeeds whether or not the first argument appears in the list. If the first argument is not in the list, then it is added to the list as the last element before the uninstantiated tail.

We have made this digression to give the reader a feel for how Tokio may "differentiate" $t-lists to store future values for a variable. Now suppose variable `A` is associated with $t-list **(1)**, and `B` is uninstantiated. Then

```
A = B
```

or

```
B = A
```

instantiates `B` to `1` (the order of the operands makes no difference). That is, `=` looks at only the head (or leftmost term) of a $t-list. Similarly, if `C` is bound to the $t-list

```
$t(1, $t(3, _)),
```

then

```
A = C
```

succeeds, but neither `A` nor `C` changes. If `D` is bound to

```
$t(2,_)
```

then

```
D = A
```

fails. These results are corollaries of the meaning of `=` inherited from Prolog and of the fact that `=` looks only at the head of $t-lists. Recall that generally a Prolog predicate may have different arguments instantiated or uninstantiated in various uses.

The Tokio function `@` (read *next*) accesses the tail of a $t-list; thus `@` used with `=` allows one to refer to the value of a variable at the next time. For example, if `A` is bound to **(1)** and if `B` is uninstantiated,

```
@A = B
```

or


```
B = @A
```

instantiates B to 2. If C is uninstantiated and D is bound to

```
$t(1, $t(2, $t(3, _))),
```

then

```
@C = @ @D
```

(the space between repeated @'s is required) binds C to the $t-list

```
$t(_, $t(3, _)).
```

If E is bound to

```
$t(1, _)
```

then

```
@E = E + 1
```

differentiates E so that it becomes

```
$t(1, $t(2, _)).
```

Finally, if A is as defined above, then

```
@A = A
```

fails since @A is 2, and A is 1, so there is no way @A and A can be made equal.

To give illuminating examples, we introduce the *always* operator, #. If P is some goal, then #P is executed at each time, from the first to the last, of the interval in which it occurs. write(A) is just as it is in Prolog, except that, instead of writing what A is bound to, which may be a $t-list, it writes only the current value of A. Suppose our Tokio program is simply,

```
test(A) :- @A = A+1, #write(A).
```

Suppose that after, this program is compiled, we issue the following query:

```
?- tokio I = 1, test(I).
```

The output will then read:

```
t0: 1
t1: 2
I = $t(1, $t(2, _9)
yes
```

where _9 is the internal name for some unbound variable. That is, the current (at t_0) value of A is 1, and the value of A at the next time (t_1) is 1, and nothing is determined for any time after that, so Tokio draws the interval to a close.

2.2 One-Time Unification vs. Unification Over All Time

The unification we have considered so far, by means of =, is a "one-time unification". The unification called "unification over all time" is achieved by unification with the head of a clause. For example, let us suppose A is bound to **(1)**, and that foo is defined by


```
foo(X) :- write(X), Y = @X, tab(3), write(Y).
```

Then evaluating

```
tokio foo(A).
```

binds `X` to `$t(1, $t(2, _))` and produces the following output

```
t0: 1    3
t1:
A = $t(1, $t(2,_9))
yes
```

As another example, suppose our program is

```
test(A) :- @A = A+1, #disp(A).
disp(X) :- write('=>'), write(X).
```

If we use the query

```
?- tokio I = 1, test(I).
```

we get

```
t0: =>1
t1: =>2
I=$t(1, $t(2, _1))
yes
```

Note that `X` in `disp` must get bound to the same $t-list as `A` in `test` for all information required to be available to `disp`.

Thus far, we have considered $t-lists whose last elements are uninstantiated variables, `_`. We now consider cases where the last element is bound to a value. Subsequent examples will make use of the `length` operator.

Recall that Tokio will discontinue the current interval as soon as everything in that interval is satisfied. One explicit way to extend or to bound an interval is by using the temporal predicate `length`. The goal `length(n)`, where `n` is a positive integer, forces the current interval to be of length `n`, i.e., to extend from some time t_i to some time t_{i+n} . For example,

```
test :- length(1), A = 1, @A = 2, @ @A = 3, #write(A).
```

output only 1 and 2.

```
test :- length(3), A = 1, @A = 2, @ @A = 3, #write(A).
```

will output 1, 2, and 3, for times t_0, t_1, and t_2 respectively, and then will indicate an uninstantiated variable for t_3.

We begin with the degenerate case, where a temporal variable, say `A`, is bound to a *simple* value, that is, a value other than a $t-list. (A *simple* value in this sense could be a list or structure of any degree of complexity.) Then the value of `A` for the present and all future times is the *simple* value to which it is bound. For example, given the program


```
p(a).
t(A) :- length(3), p(A), #write(A).
```

evaluation of `t(X)` will produce

```
t0: a
t1: a
t2: a
t3: a
X = a
```

The next greater degree of complexity arises when a temporal variable `A` is bound to a two-element $t-list, where both elements are instantiated. Then the first element is the value of A for the current time, and the second element is the value of `A` for all future time. For example, suppose we have the program

```
q(b).
rz(X) :- X=a, @q(X).
t(A) :- length(3), rz(A), #write(A).
```

then evaluating `t(Y)` produces:

```
t0: a
t1: b
t2: b
t3: b
Y = $t(a, b)
yes
```

Here the variable `Y` in the initial goal shares with `A` in the third clause; the unification is unification over all time. Again, `A` in the third clause shares with `X` in the second clause because of the goal `rz(A)` in the third clause; again the unification involved a unification over all time. In the second clause, the `X=a` goal causes the current value at `X` (and thus for `A` and of `Y`) to be `a`; and the `@q(X)` goal causes the value for `X` (and thus for `A` and for `Y`) at the next time to be `b`. Since `q(b)` is interpreted as asserting that `q(b)` is always true, it is reasonable that `@q(X)` should cause X (and thus `A` and `Y`) to have the value `b` for all future time. This is indeed the case `X` (and `A` and `Y`) is bound to `$t(a, b)`. The `a` is for the current time. At the next time point, the value for `X` is `b`, and, from the context of *this* time point, it is as if we start with `X` bound to the simple value `b`. Thus this case reduces to the previous case, but at the next point in time.

More generally, a $t-list is one of two forms:

1) `$t(`S_0`, $t(`S_1`, ... $t(`S_n`, _) ...))`

2) `$t(`S_0`, $t(`S_1`, ... $t(`S_n`, v) ,,,))`

Here S_i, $0 \leq i \leq n$, is either a variable or a simple (i.e., non-$t-list) value, and v is a simple value. Suppose variable X is bound to **1)** and the current time is t_0. Then, because the last member of the $t-list is _, the value of X is undetermined after t_n; that is, X is undifferentiated for t_i, $i > n$. Next suppose X is bound to **2)** and the current time again is t_0. Then, because the last member of the $t-list is v, a value, the value of X is v for all t_i, $i \geq n$.

Note that any of the S_i, $i \leq n$, in **1)** or **2)** may be an unbound variable. For example, given the program

```
q(b).  rz(X) :- @@X = a, @@@q(X).
t(A) :- length(4), rz(A), #write(A).
```

evaluation of t(Y) produces:

```
t0: _
t1: _
t2: a
t3: b
t4: b
Y = $t(_, $t(_, $t(a, b)))
yes
```

Sometimes Tokio must fill in gaps in a $t-list. For example, given

```
p(a).
q(b).
r(X) :- length(3), p(X), @@q(X).
```

evaluating r(Y) produces

```
t0: a
t1: a
t2: b
t3: b
Y = $t(a, $t(a, b))
yes
```

The second a in the $t-list to which Y is bound is the direct result of no unification, but it must be present to record the fact the value of Y (which shares with X of the third clause) is a at t1.

Finally, an attempt to output (using write) and undifferentiated element results in that element being differentiated. For example, given

```
p(a).
q(X) :- X = b.
```



```
rq(A) :- length(3), p(A), @q(A).
```

evaluating `rq(Y)` produces

```
Y = $t(a, $(b, _))
```

Here the undifferentiated `_` indicates that `Y` has no value for t_i, `i > 1`. On the other hand, if `#write(A)` is added to the `rq` clause giving

```
rq(A) :- length(3), p(A), @q(A), #write(A).
```

the result of evaluating `rq(Y)` is

```
t0: a
t1: b
t2: _
t3: _
Y = $t(a, $t(b, $t(_, $t(_,_))))
```

Here the final, undifferentiated `_` indicates that `Y` has no value for t_i, `i > 3`; the unbound elements for t_2 and t_3 occur explicitly because `write` was evaluated at t_2 and t_3.

2.3 Summary

Value of a Prolog variable	Value of a Tokio variable	Comment
`a`	`a`	Value `a` holds at all t_i, `i` $\geq$ `0`.
`[a,b|_]`	`$t(a, $t(b,_))`	Value `a` holds at t_0, value `b` holds at t_1, and there is no value for t_i, `i > 1`.
`[a,b|c]` i.e., `.(a, .(b,c))`	`$t(a, $t(b,c))`	Value `a` holds at t_0, value `b` holds at t_i, and value `c` holds at all t_i, `i > 1`.
`[a,b]`, i.e., `.(a, .(b, []))`	Nothing	There is no analog of `[]`.

Table 2.1: Analogy between Prolog (and Tokio) lists and Tokio $t-lists.

CHAPTER 3

Temporal Predicates and Operators

3.1.1 Always (#) and Sometimes (<>) We have already seen the `always` operator, `#`. Recall that `#P`, where `P` is some Tokio predicate, requires `P` to succeed at each time in the current interval. In declarative terms, `#P` states that `P` is (in the current interval) always `true`. The dual of `#P` is `<>P`, read *sometimes* `P`. `<>P` is `true` if `P` is `true` sometime in the future, i.e., at some t_i, `i > 0`, in the current interval. In procedural terms, `<>P` requires `P` to be a goal at each time in the interval but the first until `P` succeeds. Note that success of `P` (i.e., success of `P` at the current time) does not count to success of `<>P`. In declarative terms, `P` does not imply `<>P`. However, for `#P` to succeed, `P` must succeed at each time in the current interval, so `#P` implies `P`. Thus `#P` and `<>P` are not quite the duals they are claimed to be. Note that, if there is no way `P` can succeed, then `<>P` will cause the interval in which it occurs to be extended indefinitely unless that interval is closed. Thus it is good practice to use the combination

```
length(n), <>P,
```

where `n` is some positive integer, rather than simply `<>P`. Generally, `<>P` is used when we wish to extend the current interval far enough into the future for `P` to succeed at sometime in the interval, but when we also do not know in advance how far into the future that might be. Thus one use of `<>` could be in defining the predicate

```
sometimes_equal(A,B) :- <>A = B.
```

This will succeed if `A` and `B` have the same value for some time in the current interval. Note that it will not succeed if `A` and `B` have the same value only at a time outside the current interval.

3.1.2 Next: @ (function) and @ (operator) We have already seen the *next* function, `@`. Recall that `@I` accesses the tail of the $t-list to which `I` is bound; in particular, if `J` is uninstantiated, then

```
J = @I
```

instantiates `J` at the current time to the value for `I` at the next time. Now, there is also a `next` temporal operator, also symbolized by `@`. The goal `@P` succeeds if the goal `P` succeeds at the next time. It is important to distinguish the function `@` from the operator `@`, even though certain analogies between the two warrant emphasis. As an example,

```
test :- I = 1, @(I = 2).
```

is the same as

```
test :- I = 1, @I = 2.
```

Note that

```
test :- length(2), I = 1, @I = 2, @ @I = 3, @ #write(I).
```

will output `2` and `3` as the values for `I` at t_1 and t_2, but will not output the value for `I` at t_0. Also note that

```
@ @I = 3
```


is the same as

```
(@ @I) =  3
```

We have considered the `@` operator as delaying evaluation of a goal to the next time point. Equivalently, we could view the `@` operator as creating a subinterval of the current interval, where the subinterval begins at the next time. The two views are equivalent because all goals in an interval are executed at the beginning of the interval.

Note that a necessary condition for the goal `@P` to succeed in the current interval is that the length of this interval be at least one, i.e., there must be a time point after the current time — a next time — at which `P` can succeed. Similarly, a necessary condition for `@ @P` to succeed is that the length of the interval be at least two (from $\mathtt{t_i}$ to $\mathtt{t_{i+2}}$), and, in general, a necessary condition for

`@ @...@ P` (n `@`'s)

to succeed is that the length of the interval be at least n.

3.1.3 Weak Next: `next` The operator `next` (*weak next*) is similar to `@` except that `next P` may succeed even when there is no next time in the current interval, i.e., it is *not* a necessary condition for the success of `next P` that the length of the interval be at least one. If the goal `next P` is evaluated and there is no next time in the interval, then `next P` succeeds no matter what `P` may be. Now, an interval consisting of only one point (the current time) has length zero. A suffix subinterval of an interval is any subinterval containing that interval's last point. As a Tokio execution in an interval proceeds, successively smaller suffix subintervals of the original interval become the new current interval. When the last point in the original interval is reached, the current subinterval has length zero. Thus we may say that, when `@P` is evaluated at the last point in an interval, it must fail, and, when `next P` is evaluated at the last point in an interval, it must succeed; all of this independent is of what `P` may be.

3.1.4 Interval Termination: `empty` and `notEmpty` To clarify the semantics of `next` and other operators, we discuss a zero-argument predicate `empty` that succeeds when evaluated at the last point in an interval (or equivalently, that succeeds when evaluated in an interval of length zero). Then, in declarative terms, `next P` is `true` if and only if `empty` **v** `@P` is `true` (where **v** means *or*). Also, `length(n)` is `true` if and only if

`@ @...@ empty` (n `@`'s)

is `true`. Thus we may always replace `length(n)` with `empty` preceded by n `@`s. Now let `notEmpty` be `true` when and only when the current time is not the last time point in the current interval. Then `@P` is `true` implies `notEmpty` is `true`.

Further discussion of `empty` and `notEmpty` requires some understanding of how intervals are maintained by Tokio. If neither `empty` nor any predicate definable in terms of `empty`, such as `length`, has been evaluated, then the current interval lacks a fixed endpoint; in such a case, we say the current interval is *open-ended*. Evaluation of a goal of the form `@P` in an open-ended interval requires execution to advance at least as far as the next time, and, in general, evaluation in an open-ended interval of `P` preceded by n `@`'s requires execution to advance at

least *n+1* time points into the future. When all goals resulting from the use of `@` have been evaluated, there is no need to extend the current open-ended interval, so Tokio terminates the interval. (Tokio, however, ensures that the top-level interval has length at least one.)

A *closed* interval is an interval with a fixed endpoint, and so with a fixed length. An interval is closed because `empty` or some predicate (such as `length`) definable in terms of `empty` has been evaluated. In a closed interval of length n, a goal of the form `P` preceded by *n+1* `@`'s fails because, at the last point in the interval, and attempt is made to evaluate `@P` — but there is no next time in which `P` may succeed.

Now, there are two cases in which the goal `empty` succeeds:

1) if the current interval is open-ended, or

2) if the current interval is closed and the current time is the last time in the current interval.

There is one case in which the goal `empty` fails:

3) if the current interval is closed and the current time is not the last time in the current interval, i.e., there are times in the current interval after the current time.

In case **1)**, evaluation of `empty` causes the current, open-ended interval to be closed, with the current time as its last time; evaluation of `@empty` causes the next time to be the last time; and, in general, evaluation of `empty` preceded by n `@`'s causes the time n units after the present to be the last time of the current interval.

In an opposite manner, the goal `notEmpty` succeeds in cases **1)** and **2)**, and fails in case **2)**. Note that either `empty` or `notEmpty` will succeed in case **1)** — but with different effects. Evaluation of `notEmpty` in case **1)** causes the current, open-ended interval to extend at least to the next time (while remaining open-ended; evaluation of `@notEmpty` causes the current interval to extend at least two time points into the future; and, in general, evaluation of `notEmpty` preceded by n `@`s causes the current interval to extend at least *n+1* time points into the future.

The meaning of `empty` and `notEmpty` may be explained in terms of the interval variables maintained internally by Tokio. Suppose `I` is the interval variable for the current interval. Then `I` may be thought of as bound to the last time in the current interval if this interval is closed; if the current interval is open-ended, then `I` is unbound. For example, if the current time is $\mathtt{t_i}$ and the end of the current interval is two time points hence, then we imagine `I` bound to $\mathtt{t_{i+2}}$. We may now reformulate the conditions under which the goals `empty` and `notEmpty` succeed or fail. Let `I` be the interval variable for the current interval and let `tsuni` be the current time. Then `empty` succeeds if

1') `I` is unbound, or

2') $\mathtt{I} = \mathtt{t_i}$,

and fails if

3') $\mathtt{I} > \mathtt{t_i}$;

and notEmpty succeeds in cases **1')** and **3')** and fails in case **1')**. Note that the goal notEmpty is not the same as the goal not(empty) since the latter fails in case **1')**. In case **1')**, evaluation of empty binds I to t_i and evaluation of notEmpty places the requirement in I that, if it becomes bound, it must be bound to some $t_j > t_i$. Evaluating @empty at the current time t_i causes empty to be evaluated at t_{i+1}, thus (still assuming case **1')**) binding I to t_{i+1}. Similarly, evaluating @notEmpty at t_i will force I to be unbound at t_{i+1}.

Notice that the goal @true has the same effect as the goal notEmpty. This is because evaluating @true causes true to be evaluated at the next time point. If I is unbound, then evaluating @true results in execution reaching at least the next point in time, where true is evaluated; but evaluating true imposes no further requirements. If $I > t_i$, then @true succeeds, again with no further conditions. Finally, if $I = t_i$, @true fails since there is no next time at which true could be evaluated. It follows that the goal consisting of notEmpty preceded by *n* @'s has the same effect on the goal consisting of true preceded by *n-1* @'s. It should be evident that in, for example, the complete goal

```
notEmpty, @notEmpty, @@Empty,
```

the goals notEmpty and @notEmpty are superfluous, and the composite goal

```
@....@empty, @....@empty,
```

where the number of @'s in the two subgoals are different, always fails.

3.2 Reduction Along the Current- and Future-Time Axes

The concept of a suffix subinterval and the next operator together allow us to characterize the execution of a Tokio program as it advances from one time point to the next. The term "reduction" is used in logic programming to describe the replacement of goals by subgoals in an attempt to satisfy a query. In Tokio, reduction is done in two directions: along the current-time axis and along the future-time axis. Reduction along the current-time axis is identical to reduction in Prolog. Reduction along the future-time axis occurs when a new subinterval is generated, for example, when a goal of the form @P is evaluated. At any time point, reduction along the future-time axis is done after reduction along the current-time axis is complete. Thus Tokio must keep a list of goals to be evaluated at the next time point; this list is called the 'next queue'.

In general, reduction of a Tokio goal will produce subgoals to evaluate at the current time and subgoals to evaluate at the next time. The latter are put into the next queue, and the former are evaluated as part of the reduction along the current-time axis; thus all the former are evaluated before any of the latter. For example, when @P is evaluated, the current-time subgoal notEmpty and the future-time subgoal P are generated. If the current-time is the last time in an interval, then notEmpty is false, and so @P fails, and thus backtracking along the current-time axis is initiated; this backtracking is identical to the backtracking in Prolog. If there is a next time, then notEmpty succeeds and, assuming the rest of the reduction along the current-time axis succeeds, evaluation proceeds to the next time point, where P and other goals previously put on the next queue are evaluated. If P and these other goals succeed with respect to reduction along the current-time axis at the next time, then, in general, a new next queue will have been built, and the goals in this will be evaluated in the following time (two time points after the original time point considered). If P contains no temporal operators, then there are no goals descended from @P when reduction along the current-time axis occurs two time points after the original time. (This is not to say,

however, that there are no effects of `@P` at that point. Indeed, the effects of `@P` may propagate indefinitely in the future because of variable bindings caused by the evaluation of `P`.)

If, on the other hand, reduction along the current-time axis fails at the time after the original time, then backtracking along the future-time axis is initiated. This type of backtracking is not derived from Prolog, and will be discussed further below. Two major points should emerge from this discussion of reduction in Tokio. The first is that reduction as a whole is driven by reduction along the future-time axis, not reduction along the current-time axis. The second major point is that each step along the future-time axis may be viewed as evaluating the goal formed by applying the `next` operator to the conjunction of the subgoals in the next queue. This is because any commitment to the existence of a next time is covered by generating the current-time subgoal `notEmpty`.

The current-time and future-time subgoals generated in evaluating a goal containing a temporal operator are similar to the head and tail of a $t-list bound to a temporal variable. The heads of the $t-lists of the program variables provide the environment for the evaluation of the current-time subgoals and the tails of the $t-lists of the program variables provide the environment for the evaluation of future time-goals.

3.2.1 `#` and `<>` Revisited This discussion may be applied to the evaluation of subgoals of the form `#P` and `<>P`. Thus `#P` succeeds if `P` succeeds and `next #P` succeeds. And `<>P` succeeds if `@P` succeeds or `next <>P` succeeds. Thus `<>P` posts two subgoals, `P` and `<>P`, into the future, only one of which need succeed at the next time. Also, `<>P` can extend an open-ended interval in search of a time when `P` succeeds; this is because it evaluates the goal `@P` before the goal `next <>P`, and `@P` evaluated at the last point in an open-ended interval extends the interval by one point. In contrast, `#P` evaluates one subgoal, `P`, at the current time, and posts another subgoal, `#P`, into the future if there is a future. If there is a future, both subgoals must succeed; obviously, the second subgoal ensures that `P` is evaluated at all points of the current interval. If there is no future, then only the `P` subgoal at the current time need succeed for the `#P` to succeed; this is because the second subgoal from the point of view of the current time is `next #P`, and `next Q`, where `Q` is any goal, always succeeds when there is no future. Thus `#P` cannot extend an open-ended interval.

3.2.2 Interval Length: `length` and `skip` Next, `length(n)` is equivalent to

`@ @...@ empty` (*n* `@`'s)

This succeeds if

`@...@ empty` (*n-1* `@`'s)

succeeds at the next time. There is no subgoal for the current time, or, rather, the subgoal for the current time is simply `true`. Obviously,

`@ @...@ empty` (*n* `@`'s)

extends an open-ended interval to have a length of *n* and (because of the `empty`) makes it closed; it also fails in a closed interval of length other than *n*, but succeeds (but without effect) in a closed interval of length *n*. A goal


```
length(1)
```

is equivalent to the zero-argument goal

```
skip.
```

3.3 Interval Diagrams: Graphical Representation of Goals and Variable Values

We now present a unified graphical representation of the evaluation of Tokio goals and the values of temporal variables. We draw a discrete time-line with the points labeled t_0, t_1, ...:

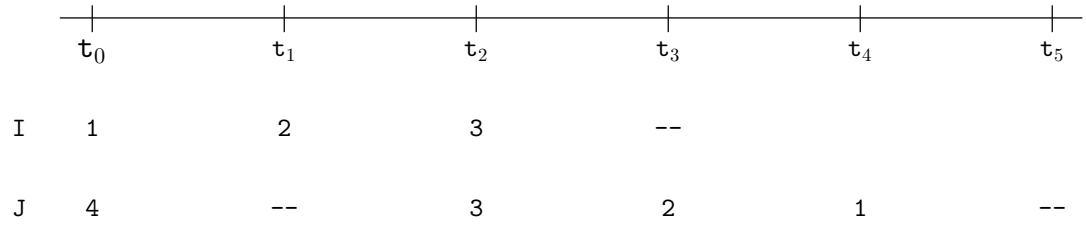
Beneath this line, we include a line for each temporal variable on which we write its value for each time. Thus, if the $t-lists for `I` and `J` are

```
$t(1, $t(2, $t(3, _)))
```

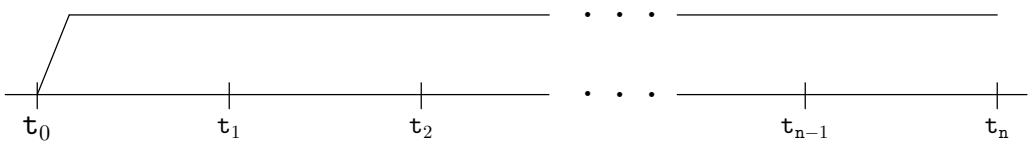
and

```
$t(4, $t(_, $t(3, $t(2, $t(1, _)))))
```

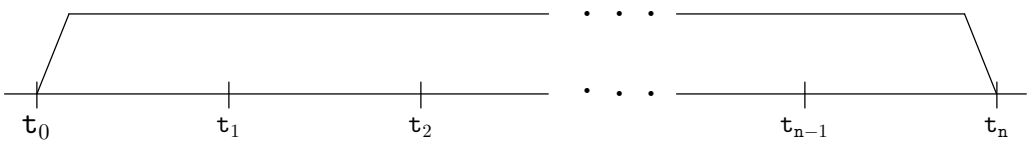
respectively, we draw



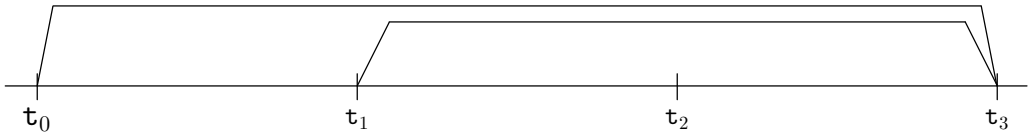
We indicate an open-ended interval if length `n` as follows:



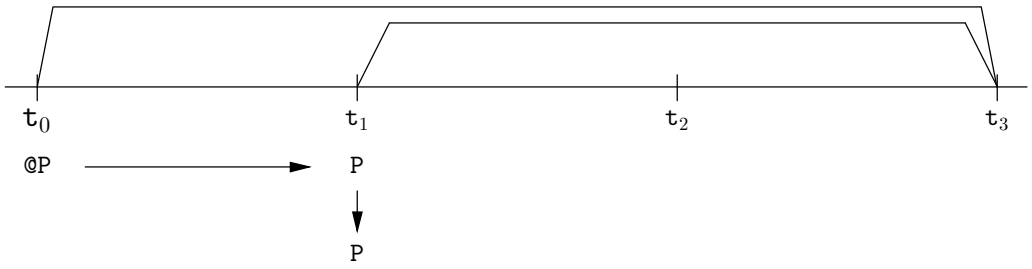
A closed interval of length `n` is indicated by

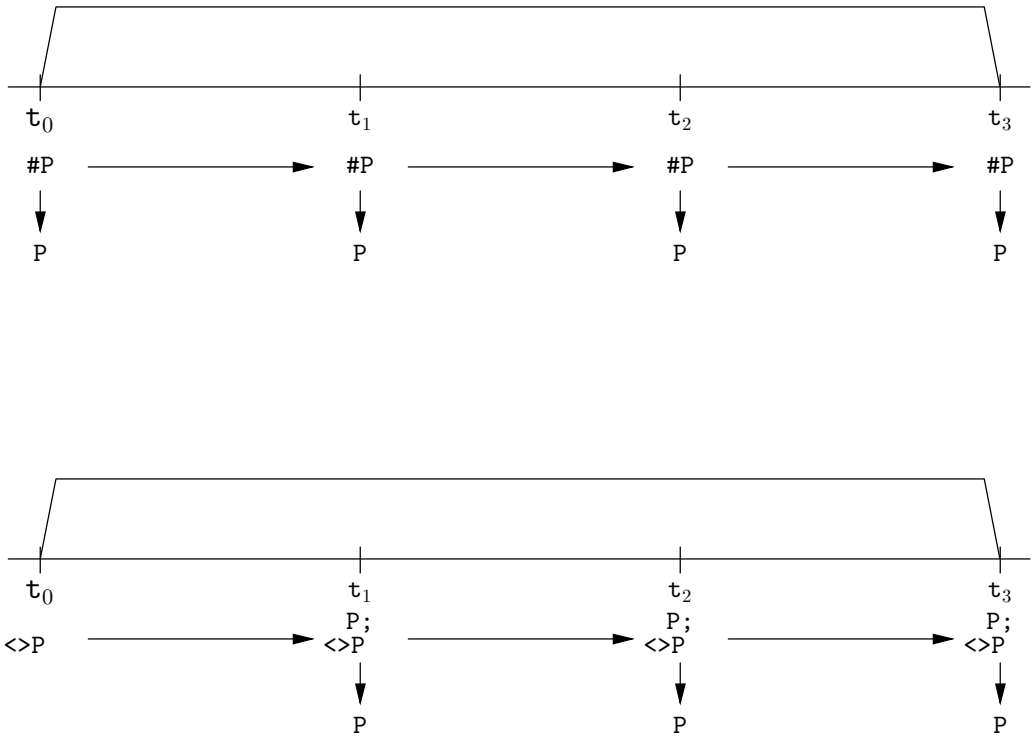


Subintervals are shown below the intervals containing them. Thus, if the original goals include `@P` and `length(3)`, then:



The goals to evaluate at time t_i are shown directly below the t_i label. An arrow from these goals points down to their current-time subgoals, and another arrow points to the right, to their future-time subgoals, which are the goals shown directly below t_{i+1}. Thus we have the following, where we assume the original interval is closed and of length `3`:

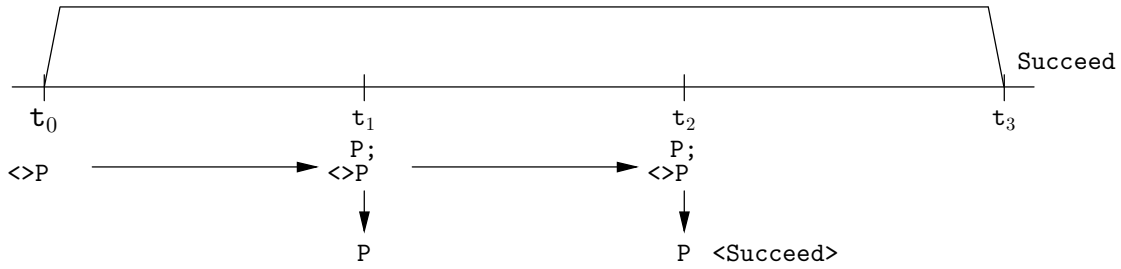




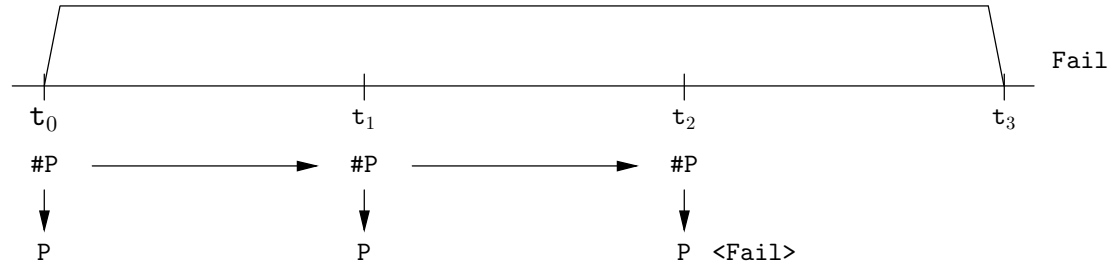
Note that no current-time subgoal is shown at t_0 in the case of `@P`. We could have put `true` as such a goal, but we shall keep things simple. Likewise, no current-time subgoal is shown at t_0 for `<>P`. We could show `@P` as a current-time subgoal at t_0, but this is the same as having `P` as a goal at t_1. We have shown the future-time subgoals of `<>P` separated by the Prolog *or* operator `;`

```
P;
<>P
```

to indicate that an attempt is made to satisfy `P` and only if this attempt fails is an attempt made to satisfy the subgoal `<>P`. Also, at t_i, `i > 0`, in the `<>P` case, `P` is included both in the goals at t_i and in the current-time subgoals at t_i. In general, a goal at t_i that includes no temporal operators will also appear as a current-time subgoal at t_i. Finally, we have assumed that `<>P`, as of t_3, has not yet succeeded since we have shown `<>P` at t_i, $0 \leq i \leq 3$. If, for example, `P` succeeds at t_2, then `<>P` succeeds, and we get the diagram



Similarly, we have assumed that `#P` has not yet failed as of t_3 since we have shown `#P` as a goal at t_i, `0 ≤ i ≤ 3`. If `P` fails at t_2, then `#P` fails, and we get the diagram

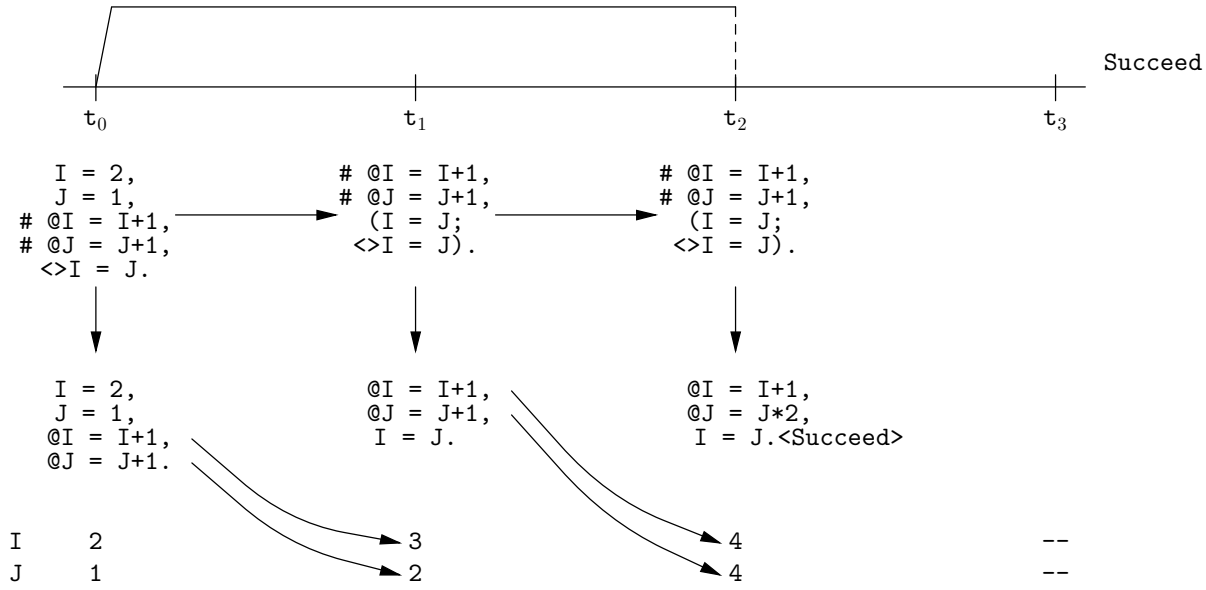


We shall henceforth indicate whether evaluation of a query succeeds or fails by writing "succeed" or "fail" to the right of its diagram. We shall also indicate for certain critical current-time subgoals whether they succeed or fail by writing "<succeed>" or "<fail>" next to them.

We may combine the representation of the values of temporal variables and the representation of current-time and future-time subgoals in a diagram. For example, suppose we have

```
test :- I = 2, J = 1, # @I = I+1, # @J = J+1, <> I = J.
```

We represent the evaluation of this with the following diagram:



We have used arrows from current-time subgoals to variable values when the subgoals affect values at times later than the present current-time. The current interval in this example terminates at t_2, when all the original subgoals have succeeded; we indicate this with a dashed line at t_2 closing the interval. Note that neither of the current-time subgoals `@I = I+1` and `@J = J+2` at t_2 were evaluated; this is because, as soon as `I=J` was found to be true at t_2, all original goals were known to be satisfied, and so evalustion could terminate.

3.4 More on Interval Lengths and Interval Diagrams

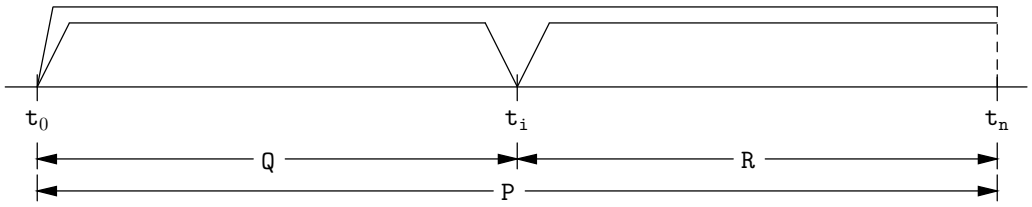
3.4.1 And: , (parallel), && (sequential -- *chop*), and & (neutral) The `and` of Prolog, that is `,`, is used in Tokio to indicate parallel conjunction of subgoals. Thus, for example, if `P` is defined by the clause

```
P :- Q, R.
```

then evaluating `P` involves evaluating `Q` and `R` in parallel. Tokio also has a sequential `and`, `&&`, read *chop*. Suppose, for example, P is defined by the clause

```
P :- Q && R.
```

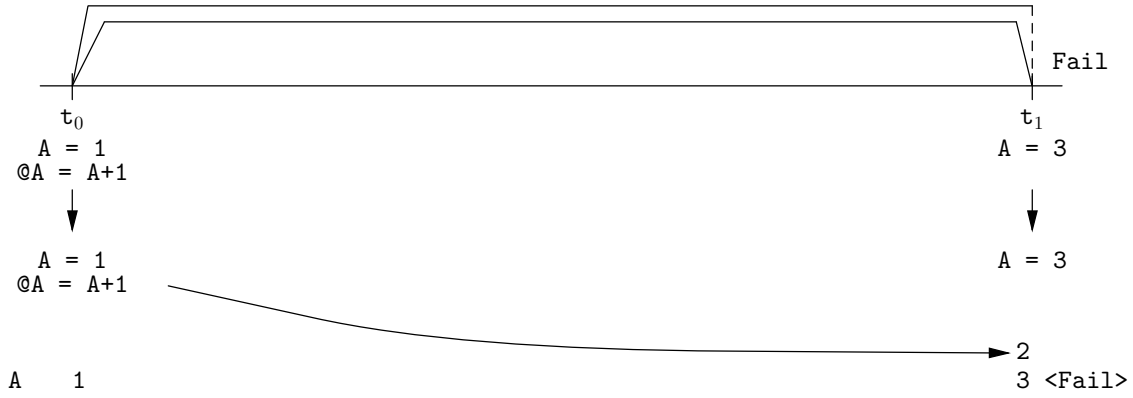
Then the interval in which `Q` can be satisfied and the interval in which `R` can be satisfied are both subintervals of the interval in which `P` can be satisfied:



Note that the last time point in the interval for `Q` is the first time point in the interval for `R`. Also note that `&&` has lower precedence than the parallel `and` (`,`). Thus, the following will always fail:

```
A = 1, @A = A + 1 && A = 3.
```

We may represent this as



The two sequential subgoals here are

```
A = 1, @A = A + 1
```

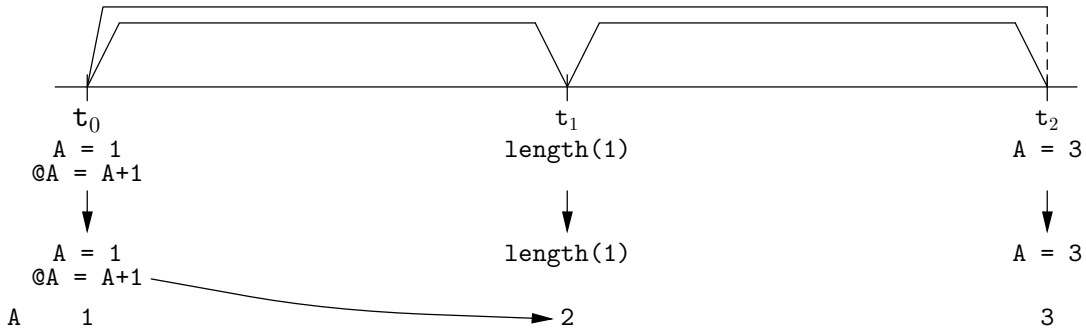
followed by

```
A = 3
```

Notice that the interval for subgoal `A=3` has length zero. In general, the only intervals that cannot have length zero are the top-level interval and the first of two subintervals resulting from a `chop`. This example fails because the first sequential subgoal requires `A` to be `2` at t_1, while the second sequential subgoal requires `A` to be `3` at t_1. Also note that, ignoring the effects of backtracking and assuming no `@` operation (as opposed to function) occur, the first of two subintervals resulting from a `chop` has length zero.

The above example should be compared with the following, which succeeds:

```
A = 1, @A = A + 1 && length(1) && A = 3
```

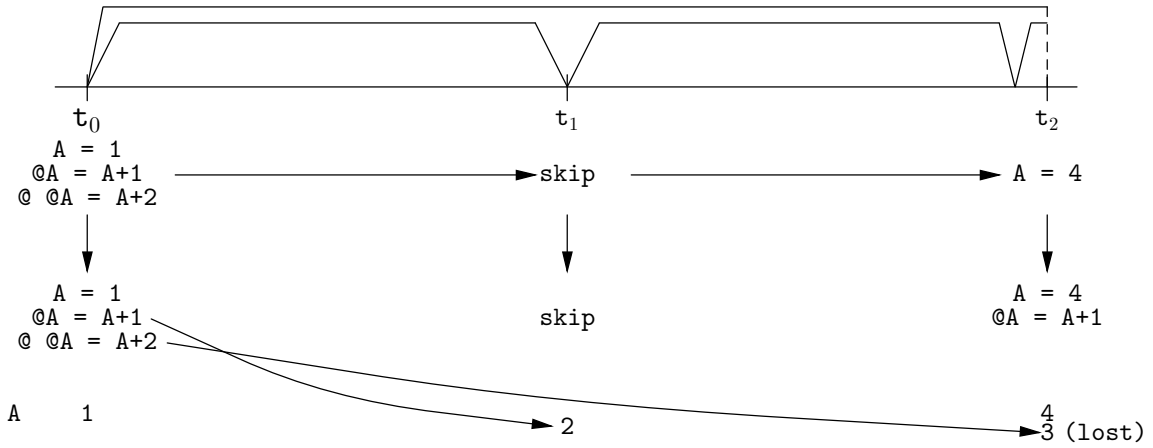



Since `&&` is associative, we consider the subintervals generated by multiple occurrences of `&&` as immediate subintervals of a single parent interval; in this case, the parent interval is divided into three subintervals, the first two of length one, and the third of length zero. This example succeeds because the second subinterval adds a time point after the point (at the end of the first subinterval) when `A` is 2; this added point is then the time when `A = 3`. The Tokio predicate `skip` is identical with `length(1)`, so the above could be rewritten as

```
A = 1, @A = A + 1 && skip && A = 3
```

Instantiations of temporal variables are not carried over from one subinterval to the next (except at the common time point terminating the one and initiating the next). Consider, for example,

```
A = 1, @A = A + 1, @ @A = A + 2 && skip && A = 4, @A = A+1
```



Here again the parent interval is divided into three subintervals, the first two of length one and the third of length zero. In the first subinterval, `A` is bound to the $t-list

```
$t(1, $t(2, $t(3, _))).
```

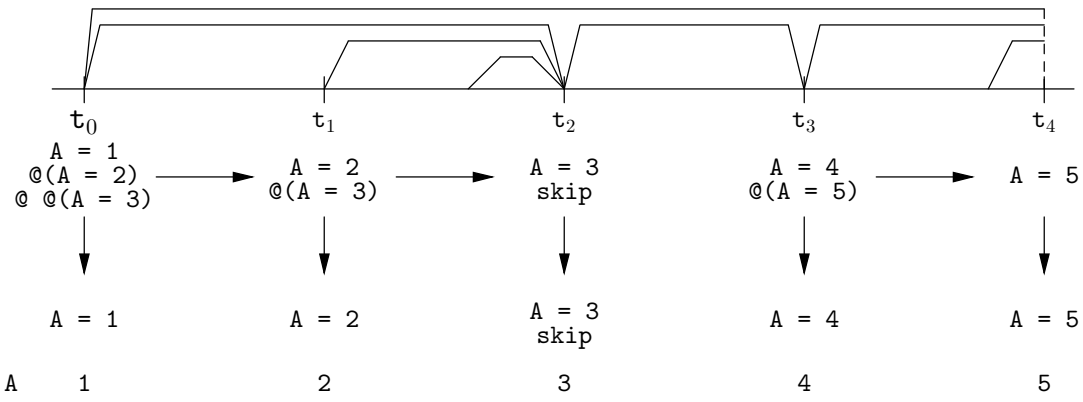

This records that at t_0 `A` is `1`, at t_1 `A` is `2`, at t_2 `A` is `3`, and `A` is undefined at t_i, `i > 2`. However, the only time points included in the first temporal interval are t_0 and t_1, so the only values of any significance are `1` at t_0 and `2` at t_1. In the third subinterval, `A` is bound to

```
$t(4, $t(5, _))
```

Since this subinterval begins (and ends) at t_2, the `4` is the value of `A` at t_2. Since `A` does not appear in the subgoal for the second subinterval, we have a consistent sequence of values for `A` throughout the parent interval, from t_0 to t_2. The $t-list records that `A` is `5` at t_4. But t_4 is past the third subinterval (and the entire parent interval), so the fact that `A` is `5` at t_4 is of no significance. Note that the `@` function (as opposed to the `@` operator) does not extend intervals.

In contrast, consider the following, where the `@` is now the operator `@`, not the function `@`:

```
A = 1, @(A = 2), @ @(A = 3) && skip && A = 4, @(A = 5)
```



Here the parent interval is again divided into three subintervals, but now the first is of length two, the second is of length one (as before), and the third is of length one. The first subinterval must be of length two because the occurrence of the subgoal `@ @(A = 3)` at t_0 implies the occurrence in the same (sub)interval of `A = 3` at $t_{0+2} = t_2$. Two suffix subintervals of the first subinterval are distinguished: one of length one from t_1 to t_2 corresponding to the subgoal `@(A = 2)` and the other of length zero at t_2 corresponding to the subgoal `@ @(A = 3)` in the subinterval or, equivalently, to the subgoal `@(A = 3)` in the suffix subinterval $<t_1\ t_2>$. In the part of the first subinterval not covered by a suffix subinterval, that is, at t_0, `A` is bound to

```
$t(1, _).
```

Since this subinterval begins at t_0, this $t-list records that `A` is `1` at t_0. In the part of the suffix subinterval $<t_1\ t_2>$ of the first subinterval not covered by the suffix subinterval $<t_2>$, that is, at t_1, `A` is bound to

```
$t(2, _).
```

Since this subinterval begins at t_1, this $t-list records that `A` is `2` at t_1. Finally, in the suffix subinterval $<t_2>$ of the first subinterval, that is, at t_2, `A` is bound to


```
$t(3, _).
```

Since this subinterval begins (and ends) at t_3, this $t-list records that `A` is `3` at t_3. Similarly, the third subinterval must be of length two, and one suffix subinterval of this subinterval is distinguished. In the part of the third subinterval not covered by the suffix subinterval (i.e., at t_3), `A` is bound to `$t(4, _)`, and in the final suffix subinterval of the third subinterval `A` is bound to `$t(5, _)`. Since `A` does not occur in the second subinterval, $<t_2\ t_3>$, we have a consistent sequence of values for `A` throughout the parent interval $<t_1\ t_2\ t_3\ t_4>$.

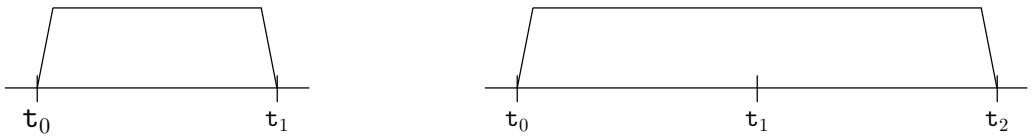
3.4.2 Discussion of Interval Lengths Often the length of an interval is not obvious, but depends on the length of time required to satisfy subgoals formulated with user-defined predicates. In such cases, one must take care that two different lengths are not specified for the same interval, which is an inconsistency. For example, suppose that the predicate `p` is defined by the clause

```
p :- r, s.
```

Now suppose `r` and `s` are defined, respectively, by the clauses

```
r :- length(1).
s :- length(2).          ...(*)
```

Then two different diagrams are specified:

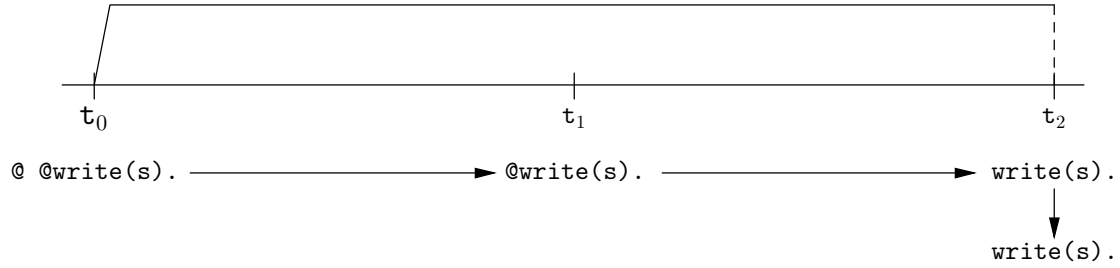


Both intervals are closed, but they have different lengths — there is no way they may be viewed as one and the same interval, and so the goal `p` must fail. In contrast, suppose `r` and `s` are defined by

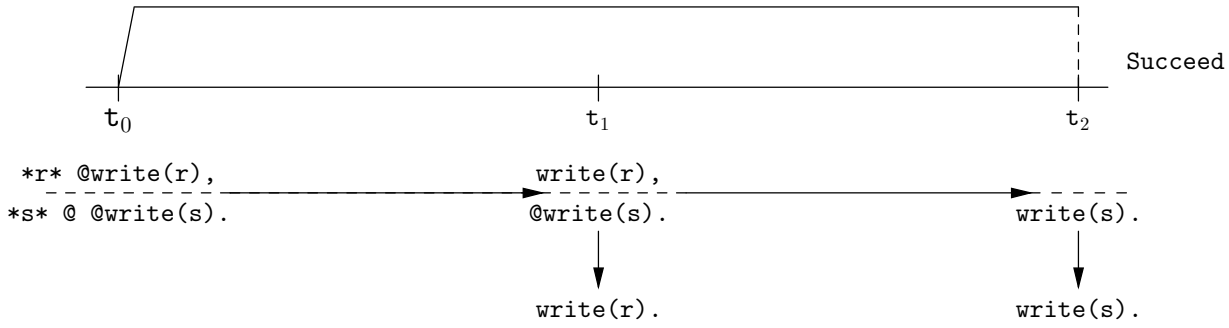
```
r :- @write(r).
s :- @ @write(s).       ...(**)
```

Then the diagram for `r` shows an open-ended interval of length one

and the diagram for `s` shows an open-ended interval of length two.



Since both intervals are open-ended, they may be viewed as one and the same interval. Specifically, since the interval for `r` is open-ended, it may be seen as the first part of an (open-ended or closed — in this case, open-ended) interval of length two. Thus, combining the diagrams for `r` and `s`, we get



Since Tokio terminates an interval when all subgoals are satisfied, the length of the interval for `p` is two. Notice that we have not bothered to indicate the suffix subintervals generated by the `@` operators.

3.4.3 Interval Diagrams and User-Defined Predicates In the last example, there are two subgoals, `r` and `s`, for the goal `p`. The subgoals `r` and `s` themselves have subgoals `@write(r)` and `@ @write(s)`, respectively. When we drew the diagram for `p`, we wrote subgoals as if `r` and `s` were replaced with their subgoals in the definition of `p`. In general, we shall draw diagrams so that each user-defined predicate is eliminated in favor of the subgoals in its definition. In the last diagram, we identified the immediate, user-defined subgoals of `p` with `r` and `s` enclosed in `*`'s to the left of where these subgoals come into play; a dotted line was used to separate the subgoals of `r` from those of `s`. We shall continue with this convention. This convention runs into difficulties when a recursive predicate appears as a subgoal. If a recursive predicate posts no subgoals into the future, then we shall treat it as a primitive. If it does post subgoals into the future, we shall treat it as we treated `#` and `<>`. Finally a disjunction of subgoals will be indicated with the Prolog *or*, `;`, as was done when we discussed `<>`.

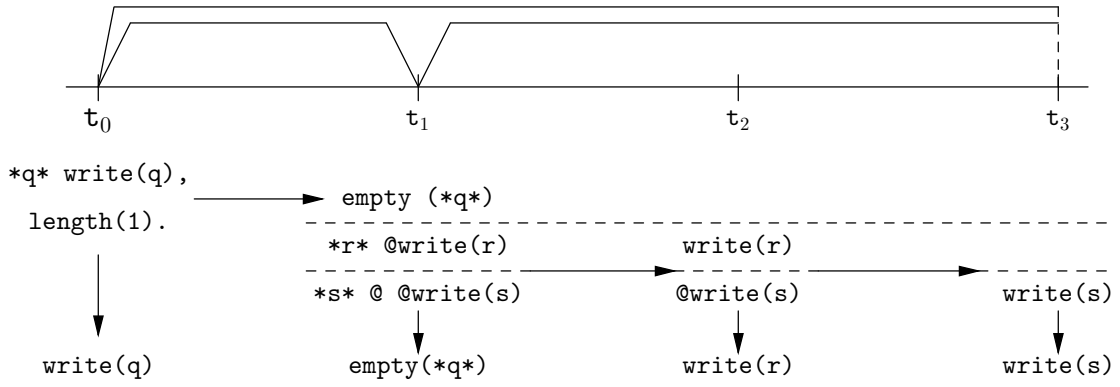
Consider, for example, the predicate `p1` defined by the clause

```
p1 :- q && r, s.
```

where `r` and `s` are defined by `(**)` and `q` is defined by

```
q :- write(q), length(1).
```

and `r` and `s` are defined as above. The diagram for this is



We have parameterized `empty` to indicate that the interval for `q` is of length zero at t_1.

3.5 Other Temporal Predicates and Operators

3.5.1 Enforcing a Constraint Throughout an Interval: `<--` With the groundwork on intervals behind us, we may now describe the remaining basic temporal operators of Tokio. Evaluation of

```
A <-- B
```


causes the value of A throughout the current interval to be the value of B at the beginning of the interval. Thus for example, instead of

```
#(A = 1),
```

we may use

```
A <-- 1
```

Notice that

```
length(3), B = 1, @B = 2, A <-- B.
```

will cause B to be bound to $t(1, $t(2, _)) and A to $t(1, $t(1, $t(1, _))), that is, A is 1 (the value of B at t_0) throughout $<t_0\ t_1\ t_2>$. In contrast,

```
length(3), B = 1, @B = 2, #(A = B).
```

will cause A to be 1 at t_0, 2 at t_1, and unbound at t_2. Finally,

```
A <-- 1 && skip && length(1), A <-- 2.
```

will cause A to have the value 1 at t_0 and t_1, and 2 at t_2 and t_3, while

```
A <-- 1 && length(1), A <-- 2.
```

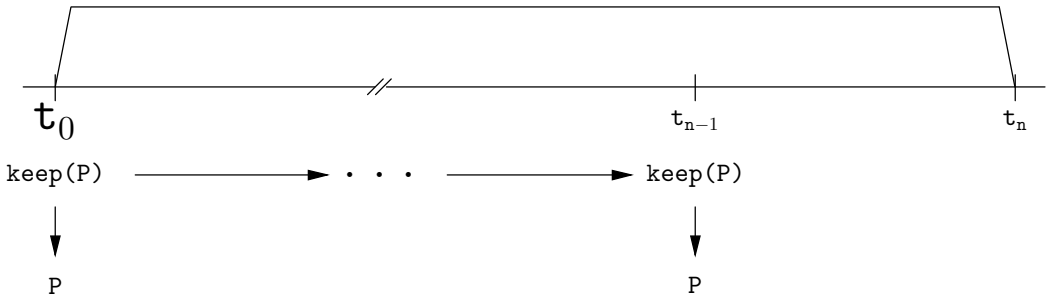
will fail since it tries to bind A to 1 and 2 at t_1.

3.5.2 Evaluating a Goal Until, or Only At, the End of an Interval: `keep` and `fin`

Evaluation of the goal

```
keep(P) :-
```

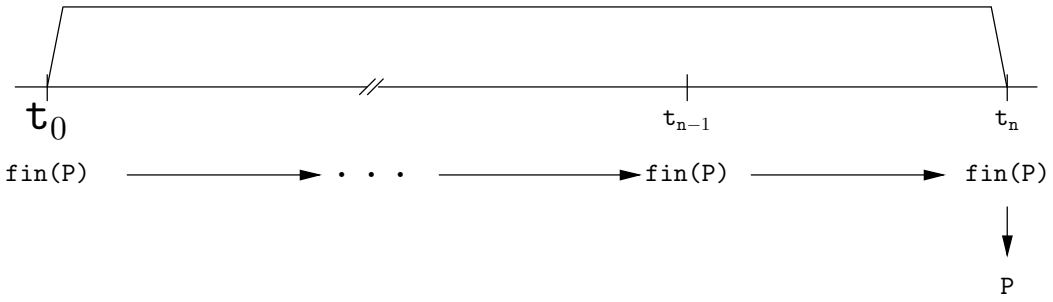
causes P to be evaluated at every point but the last in the current interval:



Evaluation of the goal

```
fin(P)
```

causes P to be evaluated at only the last point in the current interval.



Thus

```
length(3), keep(A = 1), fin(A = 1).
```

has the same effect as

```
length(3), A <-- 1.
```

However, `keep` and `fin` are much more versatile than `<--` since their arguments are goals, not variables. Thus, for example, we may have

```
keep(write(a)), fin(write(b)).
```

In practice, the arguments of `keep` and `fin` usually involve user defined predicates and produce more interesting results.

We may use `keep` to avoid introducing an interval whose sole goal is `skip` since, for example, `keep(A = 1)` does not give `A` a value the last point in the current interval, where it overlaps with the next interval. Thus,

```
length(2), keep(A = 1) && A = 2
```

is consistent, and gives `A` the value `1` for t_0 and t_1 and the value `2` for t_2. We may use `fin` to avoid `&&` in this example:

```
length(2), keep(A = 1), fin(A = 2).
```

We may describe the meanings of `keep` and `fin` very succinctly in terms of `if ... then ...` and the zero-argument predicates `empty` and `notEmpty` we introduced earlier:

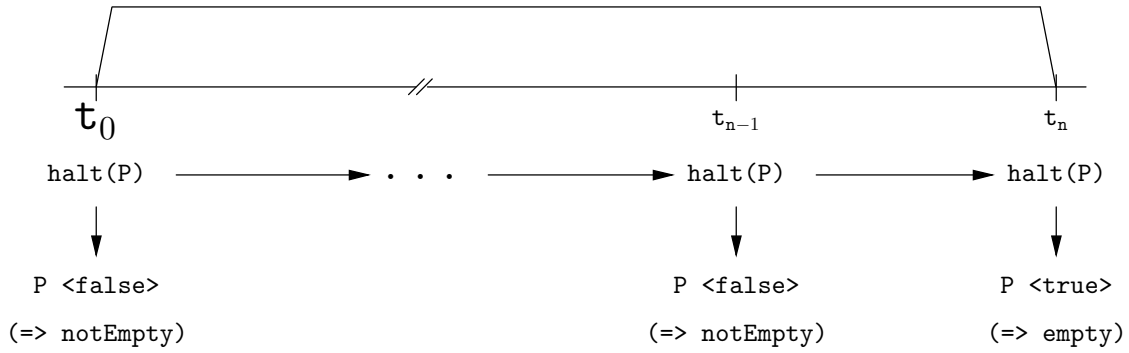
```
keep(P) :- #(if notEmpty then P).

fin(P) :- #(if empty then P).
```

Recall that `empty` is true only at the last time point of an interval and not `notEmpty` is true at all points in the interval but the last. (We shall come to `if ... then` and `if ... then ... else` later. In fact, in Tokio, `if ... then` is not a more primitive notion than `keep` or `fin`.)

3.5.3 Dependence of Interval Termination on a Goal: `halt` We may introduce `halt(P)`, where `P` is some goal, in a similar fashion:

```
halt(P) :- #(if P then empty else notEmpty).
```

Thus, if the current interval is open-ended, then `halt(P)` closes the current interval when `P` becomes true.



For example, the goal

```
halt(I = J).
```

causes the current, open-ended interval to be closed at the time t_i when `I` and `J` have the same value. If the current interval is already closed at some time t_j, then `halt(I = J)` succeeds if and only if `I` and `J` have the same value for the first time at t_j.

If `I` or `J` (or both) is unbound, then the goal `I = J` must succeed; in this case, then, `halt(I=J)` means the same as

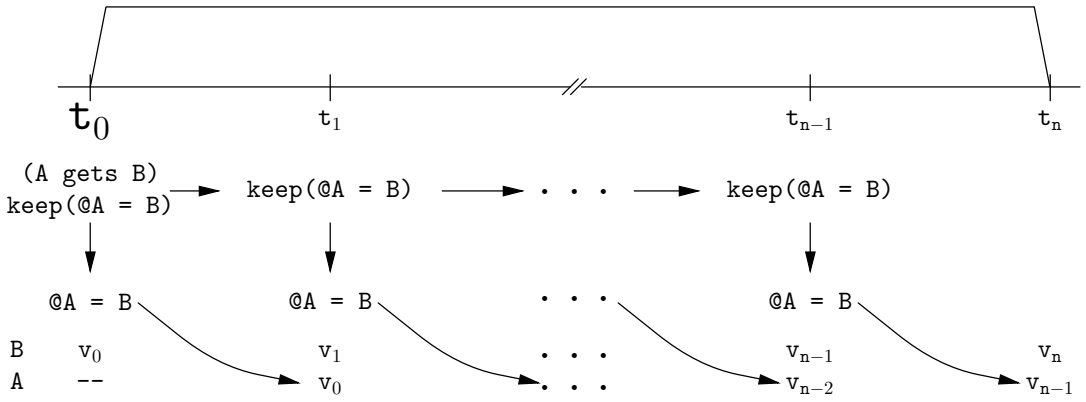
```
empty, I = J
```

From the above definition of `halt`, it follows that `halt(true)` means the same as `empty` and `halt(fail)` means the same as `notEmpty`.

3.5.4 Repeated Assignment: `gets` We may define further temporal operators in terms of `<--`, `keep` and `fin`. Thus `gets` is defined as

```
A gets B :- keep(@A = B).
```

For an interval of length `n`, if B has the values v_0, ..., v_i, ..., v_n at times t_0, ..., t_i, ..., t_n, we may diagram `A gets B` as



Note that no value is hereby determined for `A` at the beginning of the interval, t_0, and `@A = B` is not evaluated at the end of the interval t_n. The most salient fact is that the value of `A` lags that of `B` by one time unit — `gets` has an obvious application in modelling a device with unit delay.

3.5.5 Holding a Variable's Value Constant: `stable` Again, `stable` is defined as

```
stable(A) :- keep(@A = A).
```

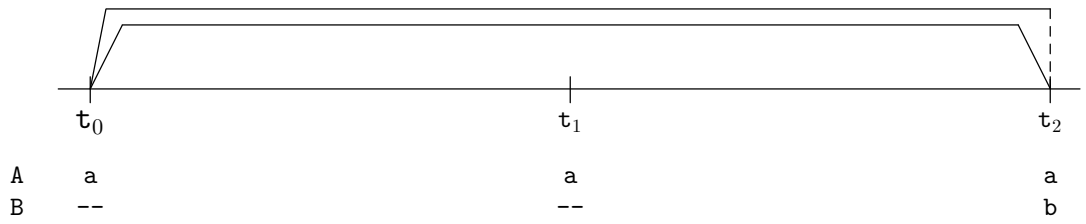
or, equivalently, as

```
stable(A) :- A gets A.
```

Thus `stable(A)` forces `A`'s value to remain constant, at the value it has at t_0, throughout the interval. An obvious use of `stable` is to carry a value from the beginning of one interval to the beginning of the next. For example, if `p(A)` determines the value of `A` only at the current time and `q(A,B)` uses the current value of `A` to determine the value of `B`, then

```
length(2), p(A), stable(A) && q(A, B)
```

makes available to `q` at t_2 the value of `A` determined by `p` at t_0. Thus:



where `a` is the value of `A` determined by `p(A)` and `b` is the value of `B` determined by `q(a,B)`. The reason `stable` can carry a value from the beginning of one interval to the beginning of the next is that the last point in the first interval is

identical to the first point of the next, and `stable` carries a value from the first to the last point of an interval. A common error in Tokio programming is to assume that the value for a variable remains stable. In fact, however, future-time values do not exist unless they are explicitly determined by means of temporal operators.

3.5.6 Temporal Assignment: `<-` Using `stable`, we may propagate the value that a variable has at the beginning of an interval to all future points in that interval. Frequently, however, as just discussed, we are really interested in setting the value of a variable at the end of an interval in terms of the value of that variable or other variables at the beginning of the interval. In that case, `<-` should be used; it is defined as

```
B <- A  :-  C <-- A, fin(B = C),
```

or, equivalently, as

```
B <- A  :-  A =C, stable(C), fin(B = C).
```

In either definition, variable `C` is a dummy variable introduced for the sake of the definition; the specification of Tokio does not require Tokio actually to introduce this third variable. The clearest way to think of `A <- B` is as `A <-- B`, but with the value of `A` determined only for the last time in the interval, not for the entire interval. As with `<--`, the second argument may be a constant, variable, or arithmetic expression. If it is a variable, its value at the beginning of the interval is used; if it is an arithmetic expression, it is evaluated at the beginning of the interval, and the resulting value is used. An interesting application of `<-` is to model the interchange of values in two registers. Suppose registers `A` and `B` each require one time unit to stabilize. Then

```
length(1), A <- B, B <- A
```

will cause `A` at t_1 to have the value `B` had at t_0 and `B` at t_1 to have the value `A` had at t_0. Note that no intermediate variable is needed. The state of affairs described is consistent since: although `A` (or `B`) has different values at different times, it never has different values at the same time.

3.6 Summary

CHAPTER 4

Backtracking into the Past

4.1 Backtracking into the Past: Simple Cases

Recall that, in Tokio, reduction (the replacement of goals by subgoals in an attempt to satisfy a query) is done in two directions: along the current-time axis, and along the future-time axis. Thus there are two sorts of backtracking: backtracking along the current-time axis, and backtracking along the future-time axis. We shall call the latter "backtracking into the past" since it moves backwards in time looking for a fresh choice at an earlier time. Reduction and backtracking along the current-time axis are identical to reduction and backtracking in Prolog. We have already extensively discussed reduction along the future-time axis. Backtracking into the past is easily understood in terms of reduction along the future-time axis. The only difficulty occurs with the division of the interval into subintervals by means of a `&&` operator. When backtracking into the past returns to a chop point, Tokio advances the chop point to the next time. Since the initial division of an interval makes the first subdivision of minimal length (length one), Tokio is thus able to generate all possible divisions of an interval into two subintervals. To explain the advance of a chop point on backtracking into the past, we shall introduce interval variables, which are internal variables marking the ends of (sub)intervals. However, we begin discussing the simple cases of backtracking into the past, those that do not encounter chop points.

When backtracking into the past returns to a time point t_i, it checks the subgoals that were executed when execution last was at t_i. The first subgoal that can be resatisfied, if there is one, is resatisfied, and execution then proceeds in the normal, forward fashion. If there is no goal at t_i that can be resatisfied, the backtracking returns to t_{i-1}. If backtracking into the past reaches t_0 and finds no resatisfiable subgoal there, then the entire query fails. We shall refer to a resatisfiable subgoal as a "choice point". We may then describe backtracking into the past as a movement back through time in an attempt to find a choice point.

When backtracking returns to some time t_i, the search for a choice point at t_i proceeds as in Prolog. The catch is, however, that the subgoals at t_i are in general generated by evaluation of subgoals at t_{i-1} — the subgoals evaluated at t_i were put into the next queue at t_{i-1}. As an example, suppose our database contains

```
p(1).
p(2).
q(1).
q(2).

r(X,Y) :- length(3), @p(X), @ @q(Y), fin(X = Y),
#write((X,Y)).
```

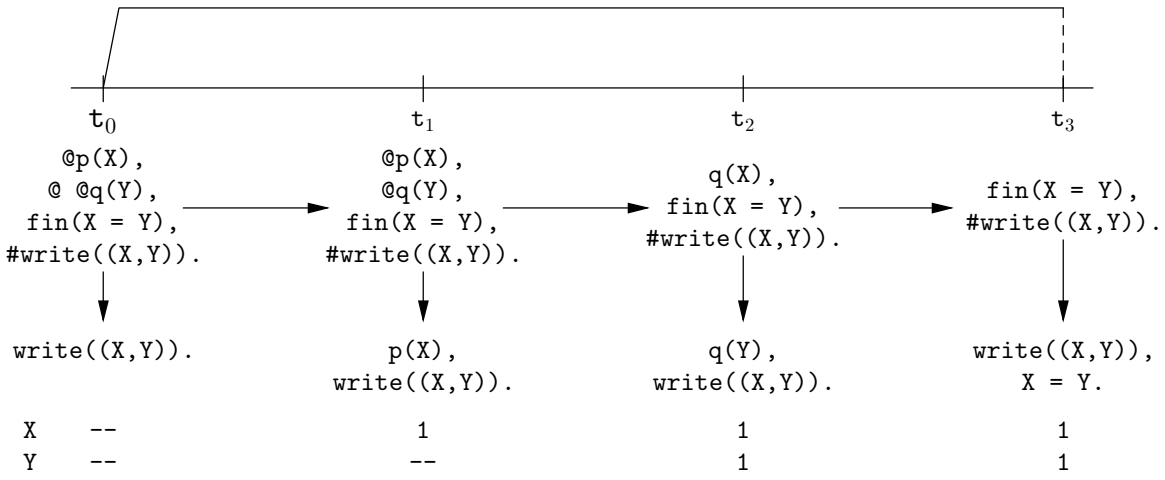
If we now issue the query

```
tokio r(X,Y).
```

then, at t_0, neither `X` nor `Y` has a value. At t_1, the subgoal `p(X)` is evaluated. It matches the `p(1)` fact in the database, and `X` unifies with `1`. This is a case of unification across all time, and `X` is `1` from t_1 on. At t_2, `q(Y)` matches the `q(1)` fact, again a unification across all time, and so `Y` is `1` from t_2 on. Thus, at t_3, `fin(X = Y)` succeeds, and the entire query succeeds. The terminal output is:


```
t0: _100,_104
t1: 1,_108
t2: 1,1
t3: 1,1
3 clock and 0.133335 sec.

X = $t(_100,1)
Y = $t(_104,$t(_108,1))
```

The `1` in `X`'s $t-list represents the value of `X` for t_1 and subsequent times, and the `1` in `Y`'s $t-list represents the value of `Y` for t_2 and subsequent times. The diagram for the calculation of this query is



Now suppose we force a failure by typing a ';' before the carriage return. There is no choice point at t_3, so the backtrack returns to t_2. Here it finds an alternative for `q(Y)`, which matches the fact `q(2)`. Thus `Y`'s value at t_2 and subsequent times becomes `2`. Execution now proceeds in the normal, forward fashion. The `write` at t_2 is evaluated, then the `write` at t_3, and finally the `X = Y` at t_3 fails, forcing another backtrack. The output generated thus far since the forced failure is:

```
b2: 1, 2
t3: 1, 2
```

Note that ´b_i:´ is output when a transition from t_{i+1} to t_i is made, while ´t_i:´ is output when a transition from t_{i-1} to t_i is made. Also notice that only the current-time goals (below the arrows) need to be considered on backtracking into the past and when execution returns forward; there is only one way these subgoals may be generated. (We shall find that this is no longer the case when we consider backtracking that encounters a chop point). Backtracking through the

current-time goals at a particular time, which is backtracking along the current-time axis, proceeds as if they formed the body of a Prolog clause. Note that their order is determined partly by the order in which subgoals appear in the Tokio clause and partly by the reordering done by Tokio.

After the failure of `X = Y` at t_3 (where `X` is `1` and `Y` is `2`), another backtrack into the past is initiated. This finds the choice point at t_2 exhausted (both `q` facts in the database have been used), so it returns to t_1. Here it finds an alternative for `p(X)`, matching it with the fact `p(2)` in the database. Thus the value of `X` is `2` from t_1 on. Execution now proceeds forward. The `write` at t_1 is evaluated, then the `q(Y)` goal at t_2 is encountered. Since this is encountered anew, it presents the same two alternatives it did when first encountered. The first alternative is taken; that is, `q(Y)` is matched with the fact `q(1)`, and so `Y` is `1` from t_2 on. Next, the `write` goal at t_2 is evaluated, then the `write` at t_3, and finally the `X = Y` goal at t_3 fails. Thus the output generated since the last failure is:

```
b2:
b1: 2,_108
t2: 2,1
t3: 2,1
```

The failure of the `X = Y` subgoal at t_3 initiates another backtrack. This finds an alternative solution for `q(Y)` at t_2: `q(Y)` is matched with `q(2)`, so `Y` is `2` from t_2 on. With execution now advancing forward, the `write` goal at t_2 is evaluated, then the `write` at t_3, and `X = Y` at t_3 succeeds, so the top-level query now succeeds for a second time. The new output is

```
b2: 2,2
t3: 2,2
3 clock and 0.400002 sec.

X = $t(_100,2)
Y = $t(_104, $t(_108,2))
```

Note that the $t-lists for `X` and `Y` are as they were the last time the query succeeded except that, in place of `1`, we now have `2`. If we now force another failure by typing ';', the backtrack thus initiated searches back through t_3, t_2, t_1, and t_0 for a choice point, but finds none, so the query now fails. The output thus produced is

```
b2:
b1:
--fail--

X = _0
Y = _1
```

The example just presented was chosen so that all choice points occur at t_1 or later. This was done because of an anomaly in the b_i lines and labels when

backtracking into the past reaches t_0. For then the line recording the transition from t_2 to t_1, which should be labeled b_1, does not appear, and the line recording the transition from t_1 to t_0 and any subsequent evaluations at t_0, which should be labeled b_0, is labeled b_1. To illustrate, we shall take the previous example and replace `length(3)` with `length(2)`, `@p(X)` with `p(X)`, and `@ @q(X)` with `@q(X)`. This amounts to shifting the time line left one unit, and discarding the original t_0, where nothing takes place. The database now contains

```
p(1).
p(2).

q(1).
q(2).

r(X,Y) :- length(2), p(X), @q(Y), fin(X = Y),
#write((X,Y)).
```

The following is a transcription of a query evaluation that exactly parallels the evaluation and re-evaluations discussed at length above:

```
| ?- tokio r(X, Y).
t0: 1,_91
t1: 1,1
t2: 1,1
2 clock and 0.100001 sec.

X = 1
Y = $t(_91, 1);

b1: 1, 2
t2: 1, 2
b1: 2, _91   <--
t1: 2, 1
t2: 2, 1
b1: 2, 2
t2: 2, 2
```



```
2 clock and 0.283334 sec.

X = 2
Y = $t(_91, 2);

b1:
-- fail --

X = _0
Y = _1
```

The arrow indicates the anomalous line. In place of this line, we should expect the following two lines:

```
b1:
b0: 2, _91
```

Indeed, Tokio never outputs the labels ´b0:´. It does not like to admit that it backtracks back to t0.

4.2 Backtracking into the Past: Cases Involving &&

Backtracking is more difficult to understand when a chop point is encountered. Associated with each interval is an interval variable, which is not visible to the programmer. If an interval is closed, the interval variable for that interval marks its last point. If an interval is open, its interval variable has no value. In either case, the interval variable for an interval is stored at its last point - this is where backtracking encounters it, and all backtracking can do is advance the interval variable forward to the next time. Recall that a && divides a parent interval into two subintervals. The first subinterval initially has length one, unless an @ operator occurs in it; this is the smallest length allowed for the first of two intervals generated by &&. We shall assume for now that no @ operator occurs in the first interval. The last point of the first subinterval is the first point of the second interval. We consider here only the simple case of two subintervals, that is, one && operator. Thus, if the parent interval is $< t_0 \ldots t_n >$, then initially the first subinterval is $< t_0 \ t_1 >$ and the second subinterval is $< t_1 \ldots t_n >$. If backtracking returns to t_1, then the interval variable for the last subinterval is advanced to t_2, which now becomes the chop point, and the two intervals are now $< t_0 \ t_1 \ t_2 >$ and $< t_2 \ldots t_n >$. Each time backtracking encounters the chop point, the chop point is advanced one unit, and so the first subinterval's length increases by one and the second subinterval's length is decreased by one. If the only choice backtracking ever finds is that afforded by the chop point, and the query never succeeds, then eventually the first subinterval is $< t_0 \ldots t_n >$ and the second is $,< t_n >$ — recall that the second subinterval, unlike the first, may have length zero. If the query is not satisfied at this stage, then it fails, for there are no other ways to divide the parent interval into two subintervals.

To introduce a succinct notation, we shall use `I` for the interval variable of the first subinterval. If there are only two subintervals, then the interval variable for the parent interval and that for the second subinterval always mark the same point - hence neither is of much interest in a discussion of backtracking. When `I` marks t_i, and so is stored at t_i, we shall assume that its value is t_i.

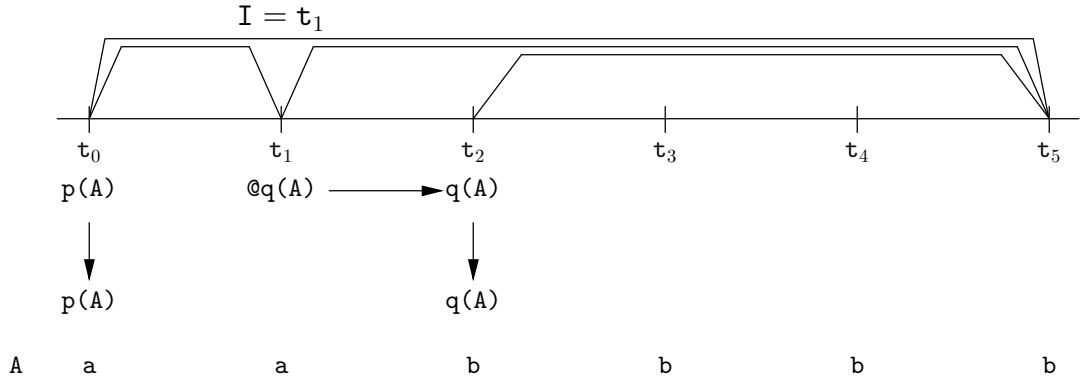
To take an example, suppose the database contains

```
p(a).
q(b).
r(A) :- p(A) && @q(A).
```

Suppose we then issue the query

```
tokio length(5), r(A), #write(A).
```

Then the chop point is at t_1, the first subinterval is $< t_0 \ t_1 >$, and the second subinterval is $< t_1 \ ... \ t_5 >$. The `p(A)` goal in the first subinterval matches the `p(a)` fact in the database (a case of unification across all time), so A is `a` during $< t_0 \ t_1 >$. The `@q(A)` goal in the second subinterval becomes the goal of `q(A)` for its suffix subinterval $< t_2 \ t_3 \ t_4 \ t_5 >$. This matches the `q(b)` fact in the database, so `A` is `b` during $< t_2 \ t_3 \ t_4 \ t_5 >$. It is obvious what Tokio will output in this case. We diagram this situation as follows:



We have written the interval variable for the first subinterval and its value over the point where it is stored. The $t-list for `A` will be

```
$t(a, $t(a, b))
```

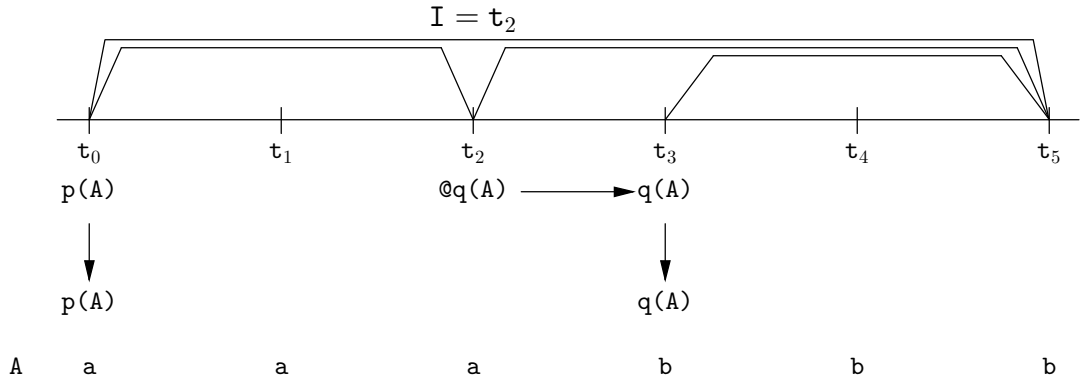
The first `a` is for t_0, the second for t_1, and the `b` is the value of `A` for t_2, t_3, t_4 and t_5.

Suppose that we now type ';' to force a failure. Since there are no goals or chop points at t_4 or t_3, backtracking returns to t_2 in search of a choice point. But there is no choice point at t_2 since there is only one clause for `q` in the database. Thus backtracking returns to the chop point t_1, where it finds `I` and advances it to t_2. After advancing `I` and while still at t_1, execution evaluates the `write` goal (not shown above) as it resumes its normal forward progress. The first subinterval, during which `A` is `a`, is now $< t_0 \ t_1 \ t_2 >$, and the second, during which `A`,

except for its initial point, is b, is $< t_2\ t_3\ t_4\ t_5 >$. The output produced by Tokio, from the time the failure is forced until the query succeeds with the second solution, is:

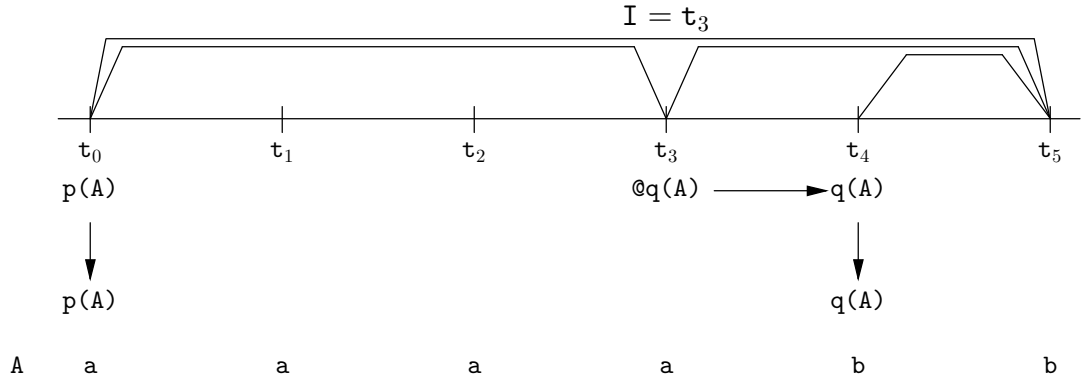
```
b4 :
b3 :
b2 :
b1 : a
t2 : a
t3 : b
t4 : b
t5 : b
```

The diagram for the second solution is

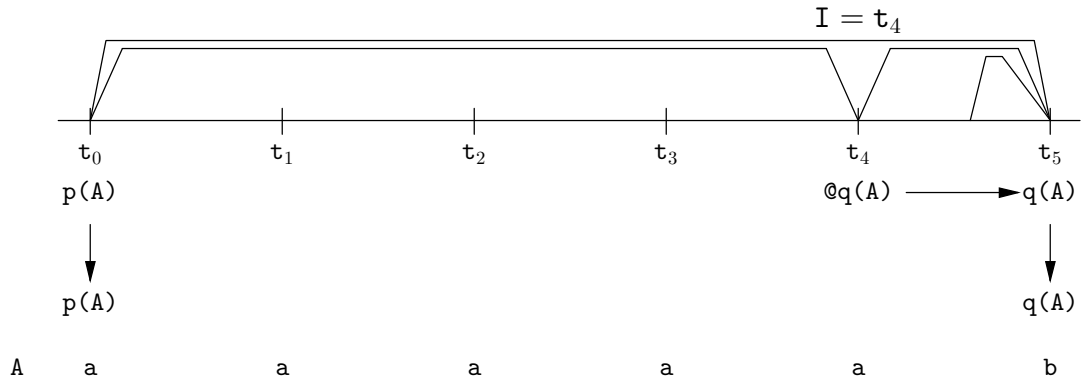


Note that it is not the case that the locations of the subgoals are unaffected by backtracking. This is because advancing the chop point forces the goals in the second subinterval to be generated one time unit later. Also (what is not illustrated here), the subgoals at t_0 may be the sources for subgoals at one additional later time unit when I is advanced.

If we force a failure at the top level a second time, I is advanced to t_3, and we get



A third forced failure advances `I` to t_4:



If we force a failure a fourth time, Tokio backtracks to t_4, where it finds `I` and advances it to t_5. It then evaluates the `write` at t_4, then the `write` at t_5, which now outputs an `a`. It then evaluates the `@q(A)` goal at t_5, which fails since there is no next time. So Tokio initiates backtracking again. `I`, now at t_5, cannot be advanced, so backtracking goes back through t_4, t_3, and so on, searching for a choice point. The only goal it encounters is `p(A)` at t_0, but the single `p` clause in the database has already been used, so the query now fails. The output produced by Tokio beginning with the last forced failure is:

```
b4 : a
t5 : a
b4 :
b3 :
b2 :
b1 :
-- fail --
```


The account just given must be modified slightly when @ operators occur in the first subinterval. Suppose a goal P contains no && operator. Then we define the "futurity" of P as follows:

1) If P contains no occurences of the @ operation, then the futurity of P is 0.

2) If P is of the form Q, R or Q; R, then the futurity of P is the maximum of the futurities of Q and R.

3) If P is of the form @Q, where Q is a (possibly conjunctive or disjunctive) goal with futurity n, then the futurity of P is n+1.

Thus, for example, both the following goals have futurity 2:

```
@@p(A)
@(q(B), @p(A))
```

Intuitively, the futurity of a goal is the number of future time units it demands in the current interval. (The notion of futurity can be extended to goals containing &&'s; each && requires at least one future time unit.) Now suppose the (possibly conjunctive or disjunctive) subgoal constituting the first subinterval has futurity $n \geq 1$. Then the length of the first subinterval is initially n, not 1 (unless $n = 1$), as we have thus far assumed. As before, however, backtracking increases, and never decreases, the length of the first interval.

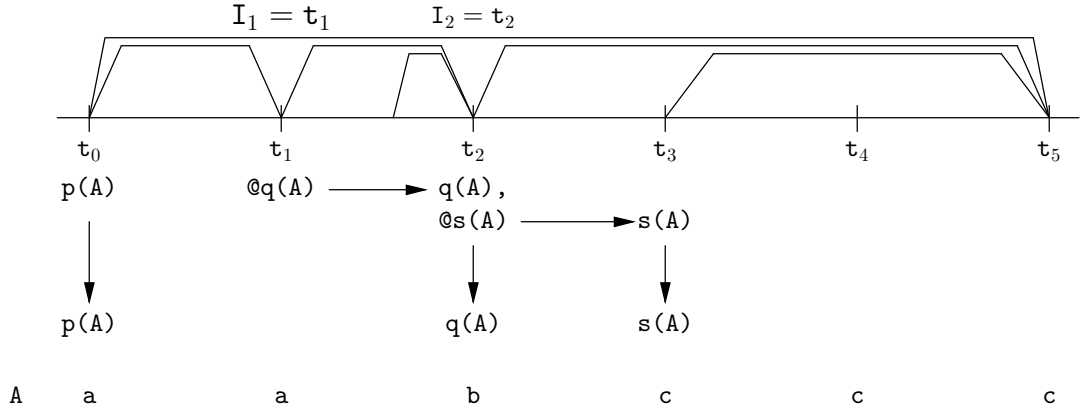
We may generalize this discussion of chop and backtracking by considering clauses that involve more than one &&. If there are n && operators, none of which are within parenthesized expressions, then the parent interval is divided into n+1 subintervals. As before, all subintervals but the last have minimum length of one, and the last may have length zero. We are now concerned with the interval variables of all subintervals but the last. So, to distinguish the interval variables of various subintervals, we let I_i denote the interval variable of the ith subinterval. We shall first present an example in which there are three chop points, and then we shall generalize to the n-chop-points case. So suppose the database contains

```
p(a).
q(b).
s(c).
r(A) :- p(A) && @q(A) && @s(A).
```

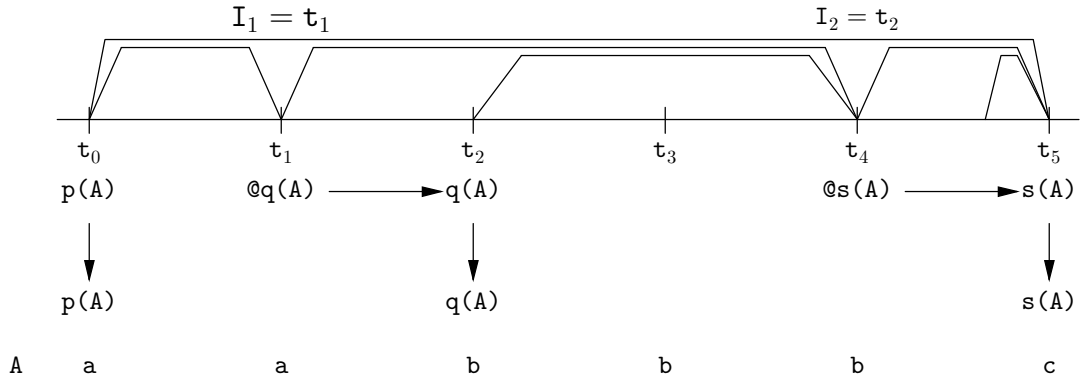
If we issue the query

```
tokio length(5), r(A), #write(A).
```

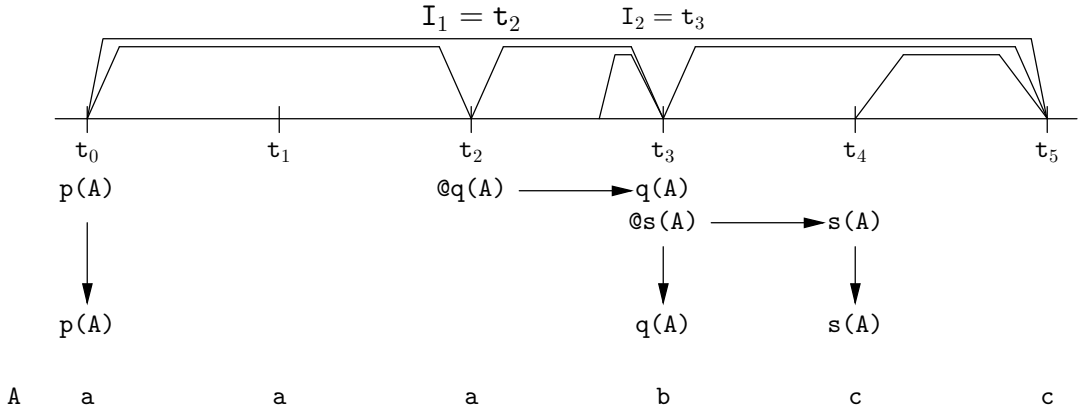
then we get the situation



Here again we write an interval variable above the time point it marks. If we now force a failure at the top level by typing a ';', backtracking will search into the past and find the first choice point at t_2, where it advances I_2 to t_3. After the next forced failure, backtracking will advance I_2 to t_4:



Another forced failure will cause Tokio to advance I_2 to t_5. Then, however, the `@s(A)` subgoal fails, so Tokio must backtrack into the past and find another choice point. It thus goes back to t_1 and advances I_1 to t_2. As execution proceeds forward, I_2 is set at the minimal distance from t_2, that is, at t_3:



Subsequent forced failures now advance $\mathtt{I}_2$ to $\mathtt{t}_4$, and then to $\mathtt{t}_5$, where `@s(A)` again fails, so $\mathtt{I}_1$ is advanced to $\mathtt{t}_3$ with $\mathtt{I}_2$ at $\mathtt{t}_4$. There are no more solutions after this. The following table summarizes the solutions.

Intervals			values of A					
I_1	I_2	I_3	t_0	t_1	t_2	t_3	t_4	t_5
$<t_0t_1>$	$<t_1t_2>$	$<t_2t_3t_4t_5>$	a	a	b	c	c	c
$<t_0t_1>$	$<t_1t_2t_3>$	$<t_3t_4t_5>$	a	a	b	b	c	c
$<t_0t_1>$	$<t_1t_2t_3>$	$<t_4t_5>$	a	a	b	b	b	c
$<t_0t_1t_2>$	$<t_2t_3>$	$<t_3t_4t_5>$	a	a	a	b	c	c
$<t_0t_1t_2>$	$<t_2t_3t_4>$	$<t_4t_5>$	a	a	a	b	b	c
$<t_0t_1t_2t_3>$	$<t_3t_4>$	$<t_4t_5>$	a	a	a	a	b	c

If backtracking returns to some chop point $\mathtt{t}_i$ marked by an interval variable $\mathtt{I}_j$ and there exists at $\mathtt{t}_i$ some choice other than advancing $\mathtt{I}_j$, Tokio will take the other choice. For example, suppose the database contains

```
q(b).

q(c).

r(A) :- (keep(A = a) && q(A)), keep(write(A)),
fin(write(A)).
```

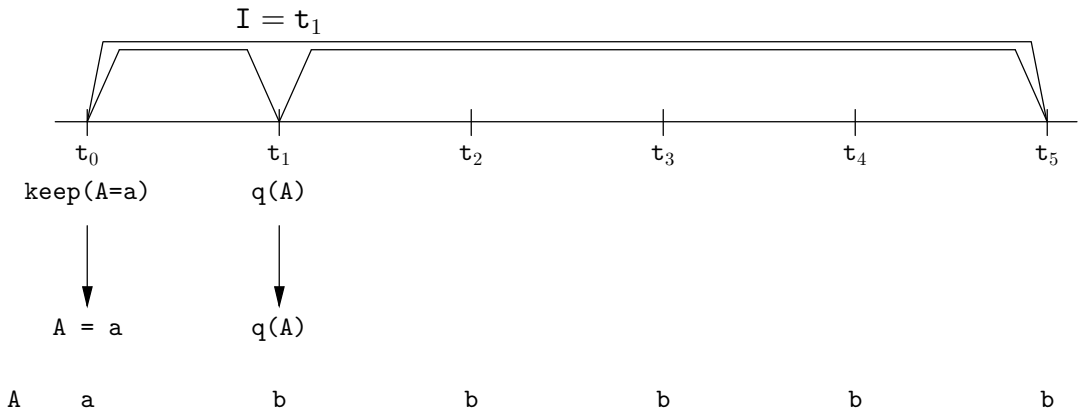
Note that there are two clauses for `q`, thus there are two solutions for the goal `q(A)`. We are interested in the behaviour of the goal

```
keep(A = a) && q(A)
```

in the clause defining `r`. The other two subgoals are included in this clause only so we may see the values for `A`. We must use `keep` and `fin` since the goal `keep(A = a)` would be evaluated `after` a simple `write(A)`, which would thus not show `A`'s value properly at each time. When we issue the query

```
tokio length(5), r(A).
```


the following situation ensues (we ignore in the diagram goals involving `write`):



After forcing a failure at the top level, backtracking finds the first choice point at $\mathtt{t}_1$. There are two choices here: resatisfy `q(A)` by matching the fact `q(b)` or advancing `I`. Since advancing the interval variable is always the last choice taken, `q(A)` is resatisfied at $\mathtt{t}_1$, and then the value of `A` from $\mathtt{t}_1$ to $\mathtt{t}_5$ become `c`, not `b`. If we force another failure, backtracking returns to $\mathtt{t}_1$. Since the alternatives for `q(A)` have been exhausted, advancing `I` to $\mathtt{t}_2$ is the only alternative. Proceeding forward, Tokio now satisfies `q(A)` anew, so `A` is `b` from $\mathtt{t}_2$ on, and is `a` before $\mathtt{t}_2$. Another backtrack would resatisfy `q(A)` at $\mathtt{t}_2$ (changing `b`'s to `c`'s); a subsequent backtrack would advance `I` to $\mathtt{t}_3$ and satisfy `q(A)` anew (giving `A` the value `b` from $\mathtt{t}_3$ on); etc. The following table summarizes the sequence of solutions.

Intervals		values of A					
first	second	t_0	t_1	t_2	t_3	t_4	t_5
$<t_0t_1>$	$<t_1t_2t_3t_4t_5>$	a	b	b	b	b	b
$<t_0t_1>$	$<t_1t_2t_3t_4t_5>$	a	c	c	c	c	c
$<t_0t_1t_2>$	$<t_2t_3t_4t_5>$	a	a	b	b	b	b
$<t_0t_1t_2>$	$<t_2t_3t_4t_5>$	a	a	c	c	c	c
$<t_0t_1t_2t_3>$	$<t_3t_4t_5>$	a	a	a	b	b	b
$<t_0t_1t_2t_3>$	$<t_3t_4t_5>$	a	a	a	c	c	c
$<t_0t_1t_2t_3t_4>$	$<t_4t_5>$	a	a	a	a	b	b
$<t_0t_1t_2t_3t_4>$	$<t_4t_5>$	a	a	a	a	c	c
$<t_0t_1t_2t_3t_4t_5>$	$<t_5>$	a	a	a	a	a	b
$<t_0t_1t_2t_3t_4t_5>$	$<t_5>$	a	a	a	a	a	c

It is possible simultaneously to chop up the same parent interval in two different ways. This may happen any time we have a goal of the form

`...,(`$\mathtt{P}_1$ `&&` $\mathtt{P}_2$`), ..., (`$\mathtt{Q}_1$ `&&` $\mathtt{Q}_2$`), ...`

and backtracking resets the interval variable for P_1 or Q_1. Hence the two &&'s chop the parent interval independently. Let us denote the interval variable for P_1 by I^1 and the interval variable for Q_1 by I^2. Initially both I^1 and I^2 mark t_1, so both &&'s chop up the parent interval in the same way. Suppose backtracking returns to t_1 and there are no alternatives at t_1 other than those presented by I^1 and I^2. (For simplicity we assume that the only choice points anywhere result from I^1 and I^2.) Since the composite goal (Q_1 && Q_2) appears after (in the clause - we are concerned at this point with backtracking along the current time axis) the composite goal (P_1 && P_2), I^2 will be advanced to t_2. Note that now the two &&'s chop up the parent interval in two different ways. Subsequent backtracks into the past will encounter I^2 before I^1, and I^2 will be advanced until it reaches the last point in the parent interval (assuming the parent interval is closed). Another failure will then cause Tokio to backtrack back to t_1. At this point, the only choice now is to advance I^1. As Tokio proceeds forward, I^2 is set anew to mark t_1. Subsequent backtracks into the past now encounter I^1 before I^2, so I^1 is now advanced, while I^2 sits at t_1. Eventually, after a sufficient number of failures, I^1 will reach the end of the parent interval. Another failure would cause the entire goal shown above to fail.

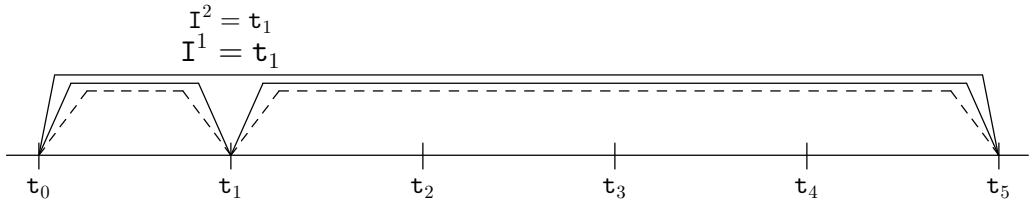
This sort of situation is complex enough to warrant an example. So suppose the database contains

```
test :- length(5),
     N = 3, (N gets N + 1 && stable(N)),
     M = 0, (M gets M + 1 && stable(M)),
     fin(M = N),
     #write((N,M)).
```

if we issue the query,

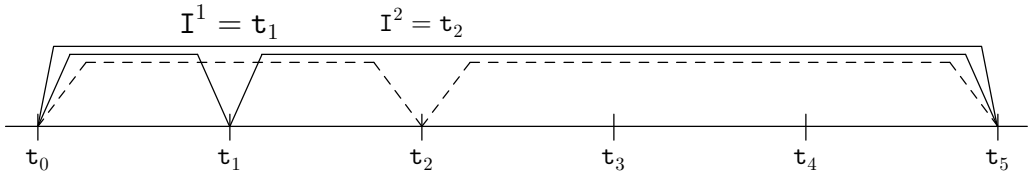
```
tokio test.
```

at first both I^1 and I^2 mark t_1. Thus N is incremented from 3 at t_0 to 4 at t_1 and then remain constant, and M is incremented from 0 at t_0 to 1 at t_1 and then remains constant. Consequently, at t_5 N is 4 and M is 1, so fin(M = N) fails:



N	3	4	4	4	4	4
M	0	1	1	1	1	1

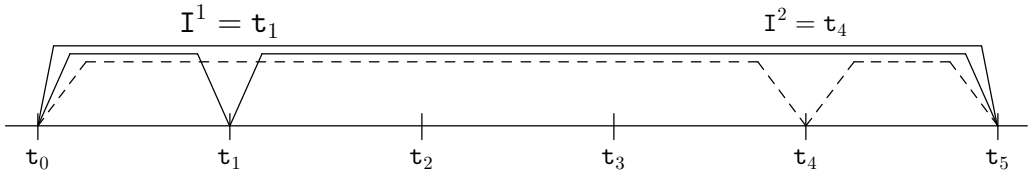
We have simplified the lists of goals shown in the diagram. Also the subintervals due to the first `&&` are drawn with a — line, while those due to the second `&&` are drawn with a --- line. The failure of `fin(M = N)` at t_5 initiates backtracking into the past, which finds no choice points until t_1. Neither `stable` goal can be resatisfied, so the interval variable due to the second `&&`, that is, I^2, is advanced to t_2. Thus the `M gets M + 1` subgoal must be evaluated at t_2, resulting in `M` having the value `2` at t_2. The `stable(M)` subgoal now does not appear until t_2, marked now by I^2. Thus,



Fail

	t_0	t_1	t_2	t_3	t_4	t_5	
N gets N+1		stable(N)				fin(M = N)	<fail>
M gets M+1			stable(M)				
N	3	4	4	4	4	4	
M	0	1	2	2	2	2	

The failure of `fin(M = N)` at t_5 again initiates another backtrack, which advances I^2 to t_3. Thus `M` is incremented to `3` before it is held stable. At t_5, `M` is now `3`, so `fin(M = N)` again fails. The resulting backtrack advances I^2 to t_4, and thus `fin(M = N)` finally succeeds:



Succeed

	t_0	t_1	t_2	t_3	t_4	t_5	
N gets N+1		stable(N)				fin(M = N)	<succeed>
M gets M+1					stable(M)		
N	3	4	4	4	4	4	
M	0	1	2	3	4	4	

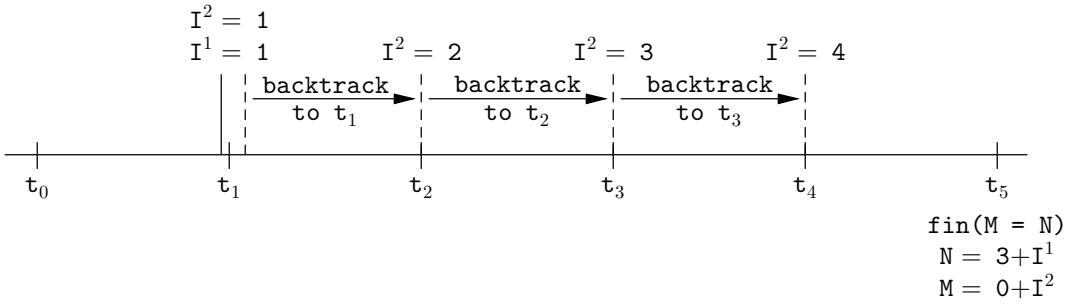
Notice that there are two invariants at t_5:

`N = 3 +` I^1

`M = 0 +` I^2

If backtracking never advances I^1 beyond t_1, then `N = 3+1 = 4` at t_5. Thus `M = N = 4` at t_5 when `M =` I^2 `= 4`, that is, when I^2 `= 4`. Thus we know in advance that `fin(M = N)` will fail three times before it succeeds. We may describe the

entire evaluation succintly by showing only the termination of intervals after subsequent backtracks, and by writing the invariants at t_5 beneath t_5:



In the example just presented, only I^2 (and not I^1) was advanced by backtracking. We now modify the example so that both I^2 and I^1 must be advanced before a solution is found. The database will now contain:

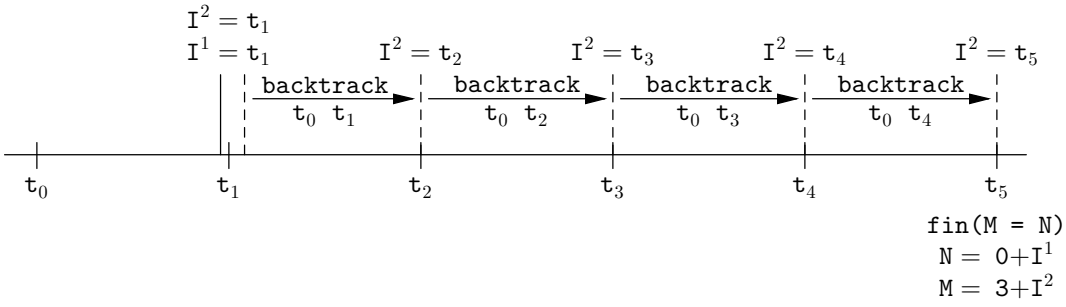
```
test :- length(5),
     N = 0, (N gets N + 1 && stable(N)),
     M = 3, (M gets M + 1 && stable(M)),
     fin(M = N),
     #write((N,M)).
```

Comparing this with the previous example, we see that, for this example, the invariant at t_5 is

$N = 0 + I^1$

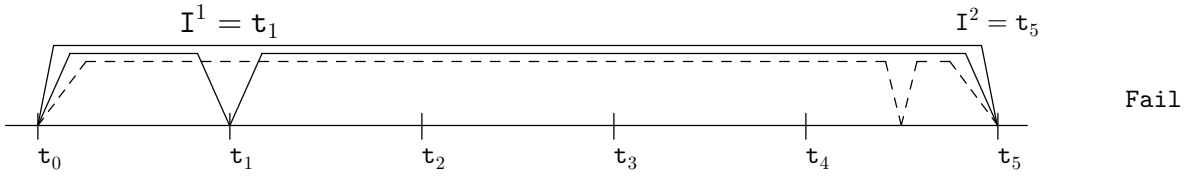
$M = 3 + I^2$

and the successive states of affairs that result when backtracking advances I^2 are, in our succint notation:



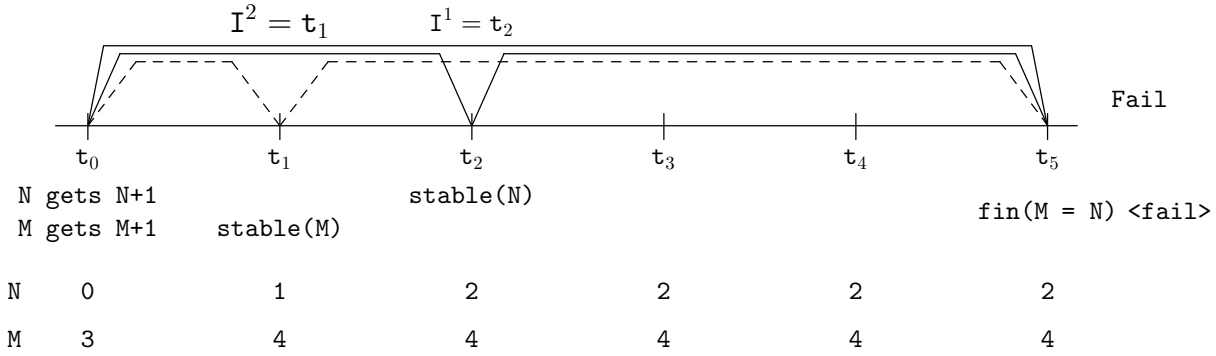
If backtracking never advances $\mathtt{I}^1$ beyond $\mathtt{t}_1$, then `N = 1`. With `1 ≤ I² ≤ 5` and `M = 3 + I²` at $\mathtt{t}_5$, there is no solution to `M = N` at $\mathtt{t}_5$. Thus to find a solution, $\mathtt{I}^1$ must be advanced.

Consider the state of affairs when $\mathtt{I}^1 = \mathtt{t}_1$, and $\mathtt{I}^2 = \mathtt{t}_5$. (This is the last state of affairs covered in the succint diagram above.) The diagram for this is:

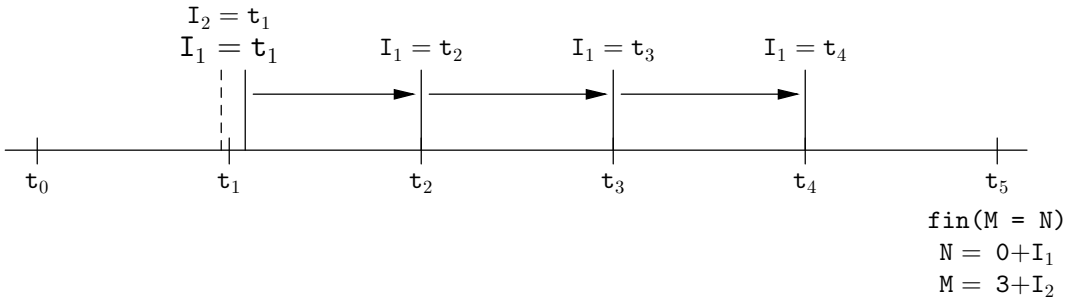



```
N gets N+1     stable(N)
M gets M+1                                          stable(M)
                                                    fin(M = N) <fail>
 N   0            1            1            1            1
 M   3            4            5            6            7            8
```

The failure at $\mathtt{t}_5$ causes evaluation to backtrack back through $\mathtt{t}_4$, $\mathtt{t}_3$, and $\mathtt{t}_2$, until it finds a choice point at $\mathtt{t}_1$. When backtracking into the past previously reached $\mathtt{t}_1$, backtracking along the current-time axis resatisfied the second `&&` subgoal, with the result that $\mathtt{I}^2$ was advanced to $\mathtt{t}_2$. When backtracking into the past reaches $\mathtt{t}_1$ this time, it again backtracks along the current-time axis. It now finds the alternatives for the second `&&` subgoal (governing $\mathtt{I}^2$) exhausted. Thus Tokio backtracks further along the current-time axis at $\mathtt{t}_1$. The next choice it encounters (and indeed, the only choice remaining at $\mathtt{t}_1$) is the alternative for the first `&&` subgoal (governing $\mathtt{I}^1$). Thus $\mathtt{I}^1$ is advanced to $\mathtt{t}_2$. Proceeding forward, Tokio evaluates the second `&&` goal anew, and so $\mathtt{I}^2$ is placed at $\mathtt{t}_1$.



Failure of `fin(M = N)` at t_5 initiates a backtrack that advances I^1 to t_3, and I^1 now advances on success or failure as I^2 did before. With I^2 fixed at 1, we see from our invariant that, at t_5, `M = 3 + 1 = 4` and $1 \le N \le 5$. There is thus a solution for `M = N` at t_5, namely, when $I_1 = 4$. Thus, `fin(M = N)` and the entire top-level goal succeed after two failures as evaluation proceeds from the state of affairs depicted above. The sequence of backtracks advancing I_2 are succintly portrayed as



The chops we have just considered may be referred to as parallel chops: they chop up the same interval, possibly in different ways. The chops we considered earlier were sequential chops: they chop up a single interval into a sequence of subintervals. As there may be two or more sequential chops in a single parent interval, so there may be two or more parallel chops in a parent interval. Any one of a set of parallel intervals may be treated like any other interval. In particular, there may be sequential chops within a parallel interval. Similarly, there may be parallel chops within a sequential interval.

4.3 Summary

CHAPTER 5

Conditionals and Iteration

5.1 Conditionals

Tokio defines its own conditional statement, which has the form:

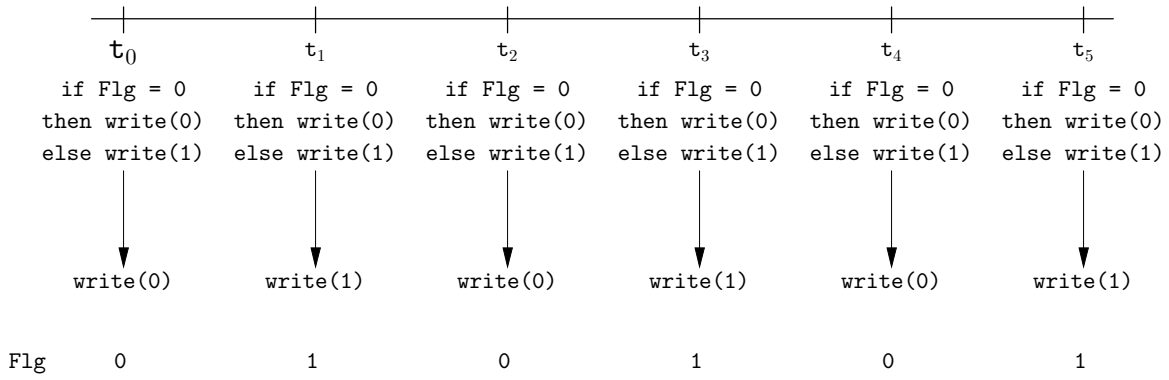
if G_1 then G_2 else G_3

The `else` part is optional. Here G_1, G_2, and G_3 are any Tokio goals. There is no need to enclose any of G_1, G_2, or G_3 within parentheses since they are evaluated before the conditional itself.

As an example, consider the clause

```
test :- length(5), Flg = 0,
     #( Flg gets 1 - Flg,
        if Flg = 0
            then write (0)
            else write (1)
      ).
```

The following diagram illustrates how this is evaluated:



Thus, the value of `Flg` is output at each time point. If the `else` part were omitted, the value of `Flg` would be output only when it is `1`. That is, given a conditional

if G_1 then G_2

if G_1 fails, then the entire conditional succeeds (although the `then` part is skipped).

Any of the constituent goals of a conditional may be composite (involve more than one predicate). For example, in the clause above, the conditional could be (if the predicates `foo1` and `foo2` are also defined):

```
(if Flg = 0
     then foo1(X), write(0)
     else foo2(X), write(1)),
write(X)
```


Notice that the parentheses around the conditional are required so that the `write(X)` does not become incorporated into the `else` part. Now, suppose `Flg` is 0 so the `then` part is selected. Then the goal becomes

```
foo1(X), write(0)
```

If the subgoal `foo1(X)` fails, then the entire conditional fails; if it succeeds, then `write(0)` is evaluated and succeeds, and so the entire conditional succeeds. The situation is similar if `Flg` is `1` and the `else` part is taken. Put succintly, once the goal following `if` has been evaluated, the success or failure of the conditional rides on the success or failure of either the `then` goal or the `else` goal, whichever is selected.

Although the parentheses are not required around the constituent goals in a conditional, their presence sometimes improves readability. Tokio also allows curly braces, `{ ... }`, to be used like parentheses.

For example, the last example could be written as

```
(if { Flg = 0 }
    then { foo1(X), write(0) }
    else { foo2(X), write(1) } ),
write(X)
```

Our convention shall be to enclose only composite goals within curly braces and not to use parentheses for this purpose, although they may be used to enclose an entire conditional. (Note that curly braces may be used anywhere to enclose a — usually composite — goal.)

5.1.1 Nested Conditionals Conditionals may be nested in any fashion. For example, the `then` goal may itself be a conditional:

```
if G1
    then if G2
         then G3
         else G4
    else G5
```

Note that parentheses or braces are not needed, although this example is clearer if written as

```
if G1
    then { if G2
         then G3
         else G4 }
    else G5
```

The dangling `else` problem is resolved by associating an `else` with the nearest `if`. Thus

```
if G1
   then
   if G2
   then G3
   else G4
```


is parsed as

```
if G1
    then { if G2
         then G3
         else G4 }
```

Also, in the following, braces (or, alternatively, parentheses) are required if the conditionalis to be parsed as shown:

```
if G1
    then { if G2
              then G3 }
    else G4
```

Probably the most useful way to nest conditionals is within the `else` goals. This produces a multiple-branch structure, as in the following:

```
if G1 then
    G2
else if G3 then
    G4
      •
      •
      •
else if Gn−1 then
     Gn
else
     Gn+1
```

This conditional has $\frac{n}{2} + 1$ branches (one of them being the `else` at the end).

5.1.2 Conditionals and Backtracking The behavior of conditionals in the context of backtracking is straightward. Recall that once it is determined whether the `then` or the `else` is taken, the success or failure of the conditional as a whole rides on the success or failure of the `then` goal or the `else` goal. If backtracking returns to the conditional, an attempt is made to resatisfy the goal selected. If this fails, then the entire conditional fails to be resatisfied. Note that no attempt is made to resatisfy the `if` goal.

To take an example, consider

```
q(3).
q(2).
q(1).
t(X) :- length(3), g(X), #write(X),
     if X = 1 then
         { write(a), write(b) }
     else if X = 2 then
         { keep(write(c)), fin(write(c)) && write(d) }
     else
         { write(e), @ @ write(f), #write(foo) }.
```


In the branch taken if `X = 2`, we have written

```
keep(write(c)), fin(write(c))
```

instead of simply

```
write(c)
```

so that, when backtracking advances the chop point, `write` is still evaluated at the point where backtracking encountered the chop point. If only `write(c)` were used, the chop point would be advanced beyond this point and no `write` would be evaluated at this point.

When the clause `t` is evaluated, subgoal `q(X)` matches the fact `q(3)` in the database, so the third branch is taken. This causes `write(e)` to be evaluated at t_0, `write(f)` to be evaluated at t_2, and `write(foo)` to be evaluated at all points in the interval, namely, at t_0, t_1, t_2, and t_3. In addition, in parallel with the conditional, `write(X)` is evaluated at all points in the interval. The output, then, is:

```
t0: 3 e foo
t1: 3   foo
t2: 3 f foo
t3: 3   foo

X = 3
```

If we then force a failure at the top level, backtracking is initiated and Tokio tries to resatisfy the branch taken, namely,

```
write(e), @ @ write(f), #write(foo)
```

Since the only predicate involved is `write`, there is no way this may be resatisfied, so the entire conditional fails, and backtracking returns to the subgoals before the conditional.

The only resatisfiable goal backtracking finds is `q(X)`. This is now matched with the fact `q(2)` in the database, and evaluation resumes in the usual forward direction. The conditional is encountered, and since `X` is `2`, Tokio now selects the second branch, namely,

```
keep(write(c)), fin(write(c)) && write(d)
```

As always, the first subinterval is given length one initially, so `c` is output at t_0 and t_1, and d is also output at t_1. The output to this point, from the time backtracking was initiated, is thus (recall that the value of `X` is output at all times):

```
b2:
b1: 2 c
t1: 2 c d
t2: 2
t3: 2

X = 2
```


Note that the time labeled `"b1:"` is actually t_0.

If backtracking is initiated again, tokio attempts to resatisfy the same branch. It does so by advancing the chop point from t_1 to t_2. To do this, it must go back to t_1, and then proceed forward anew. The new output is:

```
b2:
b1: c
t2: 2 c d
t3: 2

X = 2
```

(Note that only the `write(c)`, and not the `write(X)`, was evaluated at t_1.) Another backtrack will advance the chop point from t_2 to t_3:

```
b2: c
t3: 2 c d

X = 2
```

5.1.3 Composite `if` Goals As mentioned before, any of the goals in a conditional (the `if` goal, the `then` goal, or the `else` goal) may be composite; in fact, any of them may be arbitrarily complex. So far, we have seen examples of composite `then` and `else` goals. We turn now to cases in which the `if` goal is composite. We first consider composite `if` goals involving no temporal operators, and then we look at cases in which temporal operators are involved.

The first thing to note is that a variable, say `X`, appearing in the `if` goal is the same variable as an `X` appearing in the `then` or `else` goal. For example, suppose predicate `t` is defined by

```
t :- if q(X)
       then write(X)
       else write('q(X) failed').
```

Then, if `q(X)` succeeds with `X` bound to `1`, since the `X` in the `else` goal is the same `X` as the `X` in the `then` goal, the value `1` will be output.

When a composite `if` goal involves no temporal operator, it is evaluated like a composite Prolog goal with the exception of one feature to be mentioned shortly. In particular, backtracking along the current time axis within the `if` goal proceeds as in Prolog. As an example, suppose the database contains

```
q(3).
q(2).
q(1).
```



```
p(1).
p(2).

t :- if q(X), p(X)
      then write(yes)
      else write(no).
```

When the `t` goal is evaluated, the subgoal `q(X)` in the `if` goal will succeed with `X` bound to `3`. Then the `p(X)` subgoal fails, backtracking returns to `q(X)`, which is resatisfied with `X` bound to `2`, and then a new evaluation of `p(X)` succeeds. Thus, the `then` goal is selected, and `yes` to output at t_0.

Now suppose the `if` goal in the above definition of `t` is changed to

```
q(X), p(Y), X = Y
```

We would expect that first of all the `q(X)` subgoal would succeed with `X` bound to `3`, then the `p(Y)` subgoal would succeed with `Y` bound to `1`, and then `X = Y` subgoal would fail, thus initiating backtracking. What in fact happens is that, after `q(X)` succeeds, Tokio looks for a fact `p(3)` in the database (and fails). That is, Tokio incorporates the `X = Y` constraints directly into the variables of the other subgoals. This is the feature alluded to above in which a Tokio `if` goal involving no temporal operator differs from a Prolog goal.

When the `if` goal contains temporal operators, its success or failure at the first part of the interval (which we shall assume to be t_0) alone determines whether the `then` goal or the `else` goal is selected, but, if the `if` goal fails at some t_i, $i > 0$, then the entire conditional fails. There is no backtracking into the past within the `if` goal, although, as we have seen, there may be backtracking along the current time-axis. Thus, when the `if` goal involves temporal operators and succeeds at t_0 (so the `then` goal is selected), evaluation of the `if` goal at times t_i, $i > 0$, monitors the state of affairs as time progresses; if things do not transpire as stated in the `if` goal, the `then` goal is abandoned.

As an example, suppose the database contains

```
q(2).
q(1).

p(1).
p(3).
t1 :- if @q(X), #write(X)
      then #write(yes)
      else #write(no).
```

When `t1` is evaluated, the only subgoal derived from the `if` goal that is evaluated at t_0 is `write(X)`. This succeeds, and so the `then` goal is selected. Thus, at t_0, a value such as `_78` (indicating an unbound variable) is output for `X`, and this is followed by a `yes` output by the `then` goal. At t_1, the subgoal `q(X)`, derived from the `if` goal, is evaluated and succeeds, with `X` bound to `2`. Thus, `2` is output by the `write(X)` derived from the `if` goal, and the `then` goal is permitted to proceed, and so outputs another `yes`.

Now suppose there are no `q` facts in the database, so the `@q(X)` in the `if` goal fails. Since this has no effect until t_1, the situation remains unchanged at t_0;

in particular, the `then` goal is still selected. At t_1, `q(X)` fails, so the `write(X)` is not evaluated, and the `then` goal is abandoned, so `write(yes)` is not evaluated. Finally, the entire conditional fails with no attempt at resatisfying `q(X)`.

Next, suppose we have the same database, but with the definition of `t2` replaced with

```
t2 :- if length(1), (q(X) && p(X)), #write(X)
      then #write(yes)
      else #write(no).
```

When `t2` is evaluated, `q(X)` will succeed at t_0 with `X` bound to `2`. Thus `write(X)` outputs `2`, the `then` goal is selected, and `write(yes)` outputs `yes`. The chop point occurs at t_1. Here `X` is still `2`, but the attempt to satisfy `p(X)` at t_1 fails. Thus the entire conditional fails, and in particular, the `then` goal is abandoned. Note that no attempt is made to backtrack into the past to resatisfy `q(X)` at t_0. If, on the other hand, the `if` goal were

```
length(2), (g(X) && @p(X)), #write(X)
```

then the `if` goal would succeed at t_1, resulting in `2` (for `X`) and `yes` being output. At t_2, `X` would initially be unbound, so `p(X)` would succeed with `X` bound to `1`.

5.2 Iteration Across Time

5.2.1 Recursion with Temporal Operators Sometimes it is natural to combine recursion with temporal operators to drive evaluation of Tokio predicates through time. We content ourselves with a simple example. Let

```
loop(I) :- I < 4,
           @I = I + 1
        && loop(I).
loop(I).

t :- I = 0, loop(I), #write(I).
```

Note that the subgoals in the first `loop` clause are associated as

```
(I < 4, @I = I + 1) && loop(I)
```

When `t` is evaluated, `I` at t_0 gets the value `0`. The remaining subgoals, `loop(I)` and `#write(I)`, and evaluated in parallel, so, at each time t_i, `i` $\geq$ `0`, the value `loop` gives to `I` is output by `write`. When `loop` is evaluated at t_0, `I` is 0, so I $<$ 4 succeeds, and

```
@I = I + 1
```

causes `I` to have the value 1 at t_1. The `&&` causes the recursive `loop(I)` subgoal to be evaluated at t_1. At t_1, then, `I` is `1`, so `I < 4` succeeds, `I` at t_2 gets the value `2`, and `loop(I)` is evaluated at t_2. At t_2, `I` at t_3 gets `3`, and, at t_3, `I` at t_4 gets the value `4`. At t_4, `I < 4` fails, so the first `loop` clause fails; hence the recursive subgoal `loop(I)` in the `loop` clause as evaluated at t_3 (so the recursive subgoal is evaluated at t_4 because of the `&&`) is satisfied by the second `loop` clause in the database. Since this clause has no subgoals, nothing more is done, and the evaluation of `t` succeeds. The resulting output is :


```
t0 : 0
t1 : 1
t2 : 2
t3 : 3
t4 : 4
```

Now suppose we force the last subgoal in the evaluation just traced to fail by typing a ";". The last subgoal was satisfied by the second `loop` clause in the database. Since this clause has no subgoals, the `loop` subgoal after the `&&` in the last call to the first `loop` clause fails. Backtracking first encounters the `&&`, so we backtrack one time unit into the past, from t_5 to t_4, and advance the chop point from t_4 to t_5. The value of `I` at t_4 remains `4`. Tokio reports this backtracking into the past by outputting

```
b4 : 4
```

That is, we backtrack back to t_4 (to advance the chop point), and as execution proceeds forward, the `write` resulting from `#write(1)` is the last thing evaluated before we move from t_4 to t_5. At t_5, we must now satisfy the `loop(1)` subgoal after `&&`. This is first attempted by trying the first `loop` clause. The first subgoal in this is `I < 4`. But now I has no value at t_5. So when an attempt is made to evaluate *I < 4*, the system reports that an argument in an arithmetic expression is not a number, and bombs.

This problem is important because, when we define a predicate, we put no constraint on the contexts in which it may be used. In particular, any predicate we define must make a reasonable response when encountered by backtracking. One reasonable way for the `loop` predicate to behave in backtracking might be to extend the value of `I` at t_4 to t_5, and then carry on with the sequence as usual for subsequent times. However, given the way `loop` is defined, this is not natural and cannot be achieved in a straightforward fashion. A more reasonable way for `loop` to behave in backtracking is to throw away the last time and have the recursion terminate one time point earlier, at t_3 in our example. The intuition behind `loop` is that it gives `I` the value `i` at time t_i until `i` equals `4`. If a value is propagated back to a `loop` subgoal, `loop` in effect is asked to supply another solution, which we might take to mean the next less committal solution. We are claiming that the next less committal solution is that which is identical to the original solution except that it does not determine `I` at the last time point where the original solution determined it. What this alternative solution should not do, we claim, is move any chop points, The `&&` is included in `loop` so that `loop` may drive time forward one point per recursion cycle. If chop points could be moved forward in time, `loop` would no longer drive time in the same simple, uniform manner.

The way to fix the chop point so that they are always one time unit apart is to include a `length(1)` subgoal before the `&&` in the first `loop` clause. Recall that `skip` is the same as `length(1)`. So our modified definition of `loop` is

```
loop(I):- I < 4,
                @I = I + 1, skip
       && loop(I),
loop(I).
```

Suppose we again define `t` by

```
t :- length(5), I = 0, loop(I), #write(I).
```

Then evaluating `t` gives, as before, the output

```
t0: 0
t1: 1
t2: 2
t3: 3
t4: 4
t5: _10
```

where `_10` is some unbound variable. If we type ';' to force a backtrack, then the loop subgoal after the `&&` in the last call to `loop` fails. Since the chop point cannot be moved (because of the `skip`) and none of the previous subgoals can be resatisfied, this call to `loop` fails. This means that the `loop` subgoal after the `&&` in the previous call to `loop` fails. But now this subgoal may be satisfied by the second `loop` clause, which now terminates the recursion. Note that we have traversed backwards over one `&&`, moving from t_5 to t_4, and over a second `&&`, moving from t_4 to t_3. Resatisfying the last (in the backward direction) `loop` subgoal we have just considered does not require backtracking over a further `&&`. Thus we backtrack to t_4, then to t_3, where a `loop` subgoal is resatisfied by the nonrecursive clause and `write(I)` is executed again, outputting `3`, and time proceeds forward again, but now with no values for `I`, since the `loop` subgoal in `t` finished at t_3:

```
b4:
b3: 3
t4: _9
t5: _10
```

If we force another backtrack, we go back to t_2 and resatisfy with the nonrecursive `loop` clause:

```
b4:
b3:
b2: 2
t3: _8
t4: _9
t5: _10
```

Another forced backtrack take us back to `t1`:

```
b4:
b3:
b2:
```



```
b1: 1
t2: _7
t3: _8
t4: _9
t5: _10
```

Another takes us to t_0:

```
b4:
b3:
b2:
b1: 0
t1: _6
t2: _7
t3: _8
t4: _9
t5: _10
```

(The '0' should actually be next to a 'b_0:', not a 'b_1:'; recall that Tokio does not admit that it backtracks to t_0). At this point, we have backtracked back in time to the first call to `loop` and the recursive `loop` subgoal after the `&&` in this call is satisfied by the nonrecursive (the second) `loop` clause. If we force another failure, there is no other way for the recursive `loop` subgoal to succeed, and so the entire query fails.

```
b4:
b3:
b2:
b1:
-- fail --
```

5.2.2 `while` The general scheme illustrated by `loop`, where time is driven forward by recursion after a `&&` and is halted when a condition is nolonger satisfied, could be encoded as follows:

```
loop(Cond, Body):- if Cond
                   then Body && loop(Cond, Body).
```

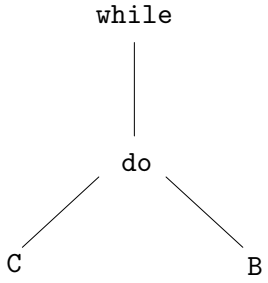
In fact, Tokio has a `while` operator that does much the same as the more general `loop`. A goal of the form

```
while C do B
```


is parsed as

```
while (C do B)
```

In fact, `do` itself is an infix operator, so the parse tree is



`While` is defined as

```
(while Cond do Body):- if Cond
                       then Body && while Cond do Body
                       else empty.
```

This differs from the definition we give of the general `loop` in that it combines an `else` clause: when Cond fails, this definition requires that `empty` succeed. `Empty` will succeed if the current time is at the end of an interval. If the interval is closed, `empty` is already true; if it is open-ended, `empty` may be made true. So `while` must finish at the end of an intervaland will close off the interval if it is open-ended.

5.2.3 `while` and Backtracking Unfortunately, `empty` is not properly implemented in the current version of Tokio. A subgoal `empty` always succeeds and, when backtracking encounters a subgoal `empty`, it goes into an infinite loop, advances from one time to the next. Consequently, backtracking goes into the same infinite loop when it encounters a `while`. Thus `while` should be avoided. Nevertheless, we shall include a few additional remarks, hoping that the bugs will soon be repaired.

5.2.4 Nested `while`'s and `if`'s `while` has a precedance similar to that of `if`, so parentheses are rarely needed within the condition or the body goals. Still, curly brackets (`{...}`) surrounding these goals may improve readability. As with the goals in a conditional, the condition and the body goals may be arbitrarily complex and may include temporal operators. `While` structures may be nested in the obvious way. Thus,

```
while C1 do while C2 do B
```

is parsed as

```
while C1 do (while C2 do B)
```


`Whiles` and conditionals may be nested together without much fear of violating syntax. Thus,

```
while C1 do if C2 then T else E
```

is parsed as

```
while C1
  (if C2
      then T
      else E
  )
```

and

```
if C1 then while C2 do B1 else while C3 do B2
```

is parsed as

```
if C1
   then (while C2
             do B1
         )
   else (while C3
             do B2
         )
```

5.3 Summary

CHAPTER 6

Static Variables

In Prolog, `assert` is used to record a result reached in evaluating a clause so that it is available when other clauses are later evaluated. On the other hand, `retract` is used so that an axiom (fact or rule) available when one clause is evaluated is no longer in force when other clauses are later evaluated. The combination of `retract` and `assert` may be used to modify the contents of the database, that is, the set of axioms, in midcourse. In general, `assert` and `retract` are used for their side effects. However, `assert` and `retract` do not have their usual utility in Tokio. This is because Tokio is compiled, that is, the contents of the database when a Tokio program is executed is the output of the Tokio compiler. If something was not in the file given as input to the Tokio compiler, then it is unknown to the compiled Tokio program. Likewise, retracting something at runtime does not do away with its image in the compiled code, and so has no effect

on run-time behavior.

Regarding the ability to make use of side effects, however, Tokio more than compensates for the loss of `assert` and `retract` by allowing global and static variables. These may often be thought of in analogy with literals asserted into the database and modified by combinations of `retract` and `assert`. However, it is not so much the global and static variables themselves that are of interest as it is their values. Because of this and because such variables serve as persistent stores for values, they are similar to variables in imperative languages such as Pascal and FORTRAN.

6.1 Assigning and Referencing Values

We describe static variables here. The present implementation does not distinguish global variables from static ones. A static variable retains its value until explicitly changed; it thus behaves like storage. The operator `*` creates a static variable. The `:=` operator is used to assign a value (to the right of `:=`) to a static variable (to the left); thus for example, `*s := 1` assigns the value `1` to the static variable `*s`. Efficiency is improved if static variables are declared. If s_1, s_2, and s_3 are static variables, then they are declared as such with the fact (usually put at the top of the program static([s_1, s_2, s_3])).

A static variable may not be used as an argument in a subgoal. More exactly, suppose, for example, we have a clause with head `foo(s)` and suppose the subgoal `foo(s)` occurs somewhere in the body of another clause. Then, when `foo(*s)` is evaluated, `s` is bound to `*s`, and not to the value of `*s`. Thus, in place of

```
write(*s)
```

one must use something like

```
S = *s, write(S)
```

Note that `=` is used to bind a logical (i.e. normal) variable to the value of a static variable. It has no effect on the value of the static variable. In particular, if `*s` has no value, evaluation of `*s = 1` leaves `*s` with no value.

Consider the following program:

```
t:- in && out.

in:- *s := 1.

out:- S = *s, write(S).
```

When `t` is evaluated, `in` is evaluated at t_0, and so `*s` is assigned the value `1` at t_0. Then `out` is evaluated at t_1, so `S` is bound to the value of `*s`, namely, `1`, and `1` is output at t_1. If `out` were defined rather as

```
out:- write(*s).
```

then, at t_1, `*s`, not `1`, would be output. Suppose the definition of `out` were replaced by

```
out:- S = *s, incr(S, S1), write(S1).
```

Suppose `incr` is defined by

```
incr(In, Out) :- Out = In + 1.
```


Then, at t_1, S, is bound to 1 (the value of *s), evaluation of incr(S, S1) results in S1 being bound to 2, and 2 is output. If out were defined rather as

```
out:- incr(*s, S1), write(S1).
```

then, at t_1, the system would attempt to evaluate *s + 1, resulting in an error because *s (as opposed to its value) is not numerical.

Even though a static variable may not be used as an argument in a predicate, it may be an argument in an arithmetic expression (as long, of course, as it has a numerical value). Consider, for example, the following program:

```
t :- *s := 0, loop(5), #{ S = (s= *s), write(S)}.

loop(Limit) :- *s < Limit,
               @(*s := *s + 1)
              && loop(Limit).
loop(_).
```

Tokio is able to evaluate the arithmetical expressions *s < Limit and *s + 1. Note in particular the assignment

```
*s := *s + 1
```

Thus, as in imperative language, a static variable may be updated to a value that is a function of its current value. Note how the value of *s is identified in the output produced by t. The goal S = (s= *s) binds S to two elements: the atom s= and the value of *s. Suppose the value of *s is 3. Then the subgoal write(S) will output s=3. The loop predicate here is like the loop predicates we considered earlier: the first clause consists of a condition (*s < Limit), a body (@(*s := *s + 1)), and a recursive call to itself after the chop. (The chop and recursive call combination drives the loop forward in time.) The second loop clause allows a call to loop to succeed when the condition in the first clause becomes false. Here the predicate t initializes the static variable *s to 0 at t_0, calls loop with Limit bound to 5, and outputs the value of *s for all times. The body of loop increments *s in the next time so that each recursive call of loop sees a value for *s one larger than that seen by the current call. The result is that *s has the value i for time t_i, $0 \leq i \leq 5 = \texttt{Limit}$.

6.2 Instantaneous vs. Temporal Assignment: := and <=

A static variable as updated by the instantaneous assignment operator, :=, acts as an individual latch that can be read immediately by any predicate. There is also a temporal assignment operator, <=. The effect of, say,

```
*s <= 1
```

appears at the end of the current interval. Thus a static variable as updated by the temporal assignment operator acts as a common bus memory; the memory unit is considered a remote device. The same static variable may be assigned values by using := at one time and using <= at another.

Before we present our simple example of the use of the temporal assignment operator, we must describe what happens when a static variable without a value is referenced. Suppose *s has no value and the subgoal S = *s is evaluated. Then Tokio will print the message


```
Reference not assigned value -- s1
```

For example, suppose `t` is defined as

```
t :- (true && *s1 := 1), #(S = *s1, write(S)).
```

Evaluation of `t` will result in the output

```
t0:
Reference not assigned value -- s1
_188
t1:1
```

The `_188` output at t_0 represents the unbound variable `S` in `write(S)`. The reference to `*s1` that produced the message at t_0 is the reference in `S = *s1`.

Now, as an example of the use of the temporal assignment operator, consider

```
t :- length(3), { length(2), *s1 <= 1 && true }, #{ S = *s1,
write(S)}.
```

When `t` is evaluated, the main interval of length three is chopped into an initial interval of length two followed by an interval of length one. The `*s1 <= 1` subgoal is evaluated in the first interval, at t_0, but the assignment does not take effect until the end of this interval, at the end of t_1. Thus the value 1 cannot be accessed as the value of `*s1` until t_2. Thus, when `t` is evaluated, the following output results:

```
t0:
Reference not assigned value -- s1

t1:
Reference not assigned value -- s1

t2: 1
```

Temporal assignment allows one to delay the change in the value of a static variable when the future value is already known. Thus the current value may be retrieved throughout the remainder of the present interval. In the next interval, the new value is retrieved when the static variable is accessed. Consider, for example, the following program:

```
t :- length(4), #{ S = (s1= *s1, s2= *s2, s3= *s3), write(S)},
     {             (length(1), *s1 := 1)
             && (length(2), *s1 <= 2, @(*s2 := *s1 + 1, @(*s3 := *s1 + 2)))
             && length(1)
     }.
```

When `t` is evaluated, the following subgoals are evaluated at the indicated time points:

```
t0: *s1 := 1
t1: *s1 <= 2
```



```
t2: *s2 := *s1 + 1
t3: *s3 := *s1 + 2
t4: -- nothing --
```

Note that the present interval <t_0 t_1 t_2 t_3 t_4> is divided into subintervals <t_0 t_1>, <t_1 t_2 t_3>, and <t_3 t_4>. Thus, because of `*s1 := 1`, `*s1` has the value `1` at the end of t_0. Because of `*s1 <= 1` and the fact that t_1 is a point in the interval ending at t_3, `*s1`'s value does not change to `2` until the end of t_3. Thus the subgoals evaluated at t_2 and t_3 access the value `1` for `*s1`. Thus `*s2` is assigned `2` at the end of t_2 and `*s3` is assigned `3` at t_3.

The instantaneous assignment operator `:=` causes assignement at the `end` of the time point at which it is evaluated, after evaluation of any `=` subgoals at that point. The output, therefore, lags one time point behind the effects of instantaneous assignments. Thus the output when the `t` predicate defined above is evaluated is (where unassigned-variable messages have been suppressed and `_1`, for example, indicates an unbound variable):

```
t0: s1 = _1, s2 = _2, s3 = _3
t1: s1 =  1, s2 = _4, s3 = _5
t2: s1 =  1, s2 = _6, s3 = _7
t3: s1 =  1, s2 =  2, s3 = _8
t4: s1 =  2, s2 =  2, s3 =  3
```

A static variable may be assigned more than one value at one time point. In that case, the last value assigned is the value the variable has after that point. For example, the sequence

```
*s := 1, *s := 2
```

will result in `*s` having the value `2` at the end of the current time point, and

```
*s <= 1, *s <= 2
```

will result in `*s` having the value `2` at the end of the current interval. If,

```
*s <= 1, *s := 2
```

is evaluated at the last point in an interval, then `*s` will have the value `1` at the end of the point and the interval. This is because a temporal assignment always takes effect after an instantaeous assignment evaluated at the same time.

6.3 Relation between Static and Logical Variables

If a normal variable is assigned to a static variable, and the normal variable is then bound to a value at the same time point, then the static variable gets this same value. For example, evaluating

```
*s := Z, Z = 1
```

causes `1` to be stored in `*s`. However, an attempt to store a $t-list in a static variable results in only the first element of the $t-list being stored. For example, evaluating


```
Z = 1, @ Z = 2, *s := Z
```

results in 1 being stored in `*s`, even though the value `Z` is `$t(1, $t(2, _))`. Thus, with respect to temporal variables, ':=' behaves like '=' in that the unification is at the current time only, not (as when a goal matches the head of a clause) over all time. If we later evaluate

```
S = *s, #write(S)
```

the value `1` will be output for the current and all subsequent times. The operator ':=' is unlike '=', however, in that static variables cannot be made to share. For example, if

```
*s := [X, 2, 3]
```

is evaluated, and then, in a different clause,

```
[X, 2, 3] = *s, X = 1
```

is evaluated, the first element in the list stored in `*s` remains uninstantiated.

6.4 Static-Variable Assignment and Backtracking

Unlike `assert` and `retract`, the effects of instantaeous and temporal assignments are undone by backtracking, whether the backtracking is done in current time or into the past. Consider, for example:

```
t:- *s := 1, fail.

t:- S = *s, write(S).
```

When `t` is evaluated, the first clause is chosen, so `*s` is assigned a `1`. The `fail` causes the first clause to fail, so the second clause is chosen, and the value of `*s` is bound to `S` and output. Since backtracking over the `*s = 1` subgoal (which cannot be resatisfied) undoes its effct, an unbound variable value is output. Next consider:

```
t :- length(2),
     {       *s <= 1,
       && skip
       && S = *s, write(S), fail
     }.
t :- length(2), #{S = *s, write(S)}.
```

When `t` as thus defined is evaluated, the first clause is chosen. At t_0, the `*s <= 1` subgoal is evaluated. This is done in the interval $< t_0\ t_1 >$. Thus at t_2, when

```
S = *s, write(S), fail
```

is evaluated, `*s` has the value `1`, which is output. But the `fail` is encountered, which forces the system to backtrack into the past. At t_2 and t_1, no resatisfiable goals are found. So the system returns to t_0. Since `*s <= 1` cannot be resatisfied, the second clause is chosen. Evaluation then proceeds forward in time through t_0, t_1 and t_2. At each point, the value of `*s` is output. Since backtracking undoes the effect of `<=`, at each point, including t_2, an unbound variable value is output.

6.5 Static Variables as Arrays

Static variables may have arguments, and hence act like arrays. For example, suppose `t` is defined by

```
t :- init && out.

init :- *g(1,1) := 1, *g(1,2) := 2,
          *g(2,1) := 3, *g(2,2) := 4.

out :- G11 = *g(1,1), G12 = *g(1,2),
          G21 = *g(2,1), G22 = *g(2,2),
          write((G11,G12,G21,G22)).
```

When `t` is evaluated, at t_1 Tokio will output

```
1,2,3,4
```

Here `*g` acts like a two-by-two array.

We may thus write programs that loop over the elements of arrays of several dimensions. For example, consider:

```
t(Rs, Cs) :-
     *cols := Cs, init(1, 1, Rs, Cs), nl,
     out(1, 1, Rs, Cs).

init(I, J1, In, Jn) :-
     I =< In, !,
     init1(I, J1, Jn), I1 = I + 1,
     init(I1, J1, In, Jn).

init(_,_,_,_).

init1(I, J1, Jn) :-
     J <= Jn, !,
     *g(I, J) := (I - 1) * *cols + J,
     J1 = J + 1,
     init1(I, J1, Jn).
init1(_,_,_).

out(I, J, In, Jn) :-
     I <= In, !,
     out1(I, J1, Jn), nl, I1 = I +1,
     out(I1, J1, In, Jn).

out(_,_,_,_).

out1(I, J, Jn) :-
     J <= Jn, !,
     K = *g(I,J), tab(3), write(k),
     J1 = J + 1,
     out1(I, J1, Jn).

out1(_,_,_).
```


The program produces the following output (which we have prettied slightly to align coulmns), when `t(4,6)` is evaluated:

```
t0:
  1     2     3     4     5     6
  7     8     9    10    11    12
  13    14    15    16    17    18
  19    20    21    22    23    24
t1:
```

`t(Rs, Cs)` computes in row-major form the integers from `1` to `Rs*Cs` and stores these values in the `Rs`-by-`Cs` array `*g(I, J)`. The first `Cs` positive integers are stored in row 1, the second `Cs` are stored in row 2, etc. It then outputs the contents of *`*g(I, J)`*, one row per output line. Predicate `t` first evaluates `*cols := Cs` so that the number of columns is later available as the value of `*cols`. It then evaluates `init(1, 1, Rs, Cs)`, which stores values in array `*g(I, J)`. Predicate `init` loops over rows and evaluates `init1(I, J1, Jn)` for each row `I`, `1 ≤ I ≤ Rs`. Predicate `init1`, for a given row `I`, loops over the columns `J`, `1 ≤ J ≤ Cs`. For each column `J`, it computes the value of `*g(I, J)` as

```
*g(I, J) := (I - 1) * *cols + J
```

The last thing `t` does is evaluate `out(1, 1, Rs, Cs)`, which outputs the contents of the array. Structually, `out` is like `init`, looping over rows and evaluating `out1` (and analogous to `init1`) to loop over columns. The sequence of subgoals in `out1` responsible for output is

```
K = *g(I, J), tab(3), write(K)
```

It is difficult to make `t` as defined above more general. The best we can do is write a predicate to compute array values, say

```
f1(I, J, Res) :- Res = (I - 1) * *cols + J.
```

Then, to compute array elements differently, one redefines `f1`. One is tempted to add another argument to `t`, `init`, and `init1`, say `Func`, to be bound to the functor of the predicate computing array values. The intent would be to use `univ` to construct the subgoal computing the array element:

```
Goal =.. [Func, I, J, K], call(Goal), *g(I, J) := K
```

The problem with this is that `Goal` is constructed at run-time and the predicate whose functor, say `f1`, is bound to `Func` is compiled, and so is in a form different from the `f1` predicate originally written. For the same reason, it is not possible to add an argument, say `Array`, and construct an array-assignment goal such as

```
Goal1 =.. [Array, I, J], Goal2 =.. [':=', Goal1, K],
call(Goal2)
```

Thus we must be satisfied with predetermined array name predicates for computing element values. However, this does allow us to keep several distinct global memories.

A static variable may be used with different numbers of indices. For example, we could have

```
*s(1) := 1, *s(1, 1) = 2
```

An alternative to using array notation is to store a list (possibly of lists) in a scalar static variable. Thus, instead of


```
*s(1, 1) := 1, *s(1, 2) := 2, *s(2, 1) := 3, *s(2, 2) := 4,
*s(3, 1) := 5, *s(3, 2) := 6
```

we could use

```
*s := [[1, 2], [3, 4], [5, 6]]
```

To output, for example, the element in the second row, first column, we would use

```
[_, [X, _], _] = *s, write(X)
```

The structure used to implement arrays need not be lists. For example, we could use

```
*s := ((1, 2), (3, 4), (5, 6))
```

and then

```
(_, (X, _), _) = *s, write(X)
```

Hybrids between array and structure notation are possible. Thus, for what is conceptually a three-dimensional array, we could have

```
*g(1, 2) := [a, b, c]
```

and output one element with

```
[_, X, _] = *g(1, 2), write(X)
```

The indices themselves may be structures, thus allowing a sort of associative memory. Thus we could store distances, both by air and by car, between cities with

```
*distance( air( detroit, miami)) := 1100,
*distance( car( detroit, miami)) := 1300,
*distance( air( detroit, boston)) := 700,
*distance( car( detroit, boston)) := 900
```

To output the distance by car between Detroit and Miami, we would use

```
D = *distance( car( detroit, miami)), write(D)
```

Lists may be used as indices. Suppose, for example, we are simulating a very simple computer with four-bit memory addresses and eight-bit memory words; we represent a binary number by a list of `0`'s and `1`'s. Then, to store the value `2` at address `2`, we would use

```
*mem([0,0,1,0]) := [0,0,0,0,0,0,1,0]
```

An index can even be another static variable. For example, to store the contents of the data buffer register (DBR) at the address in memory stored in the memory address register (MAR), we could use

```
*mem(*mar) := *dbr
```

When an assignment is made, an index may be a variable. This has the effect of assigning the same value to an entire family of array cells. For example, evaluation of

```
*mem(0,_,1) := foo
```

followed by evaluation of

```
M1 = *mem(0,1,1), M10 = *mem(0,10,1), M = *mem(0,_,1),
write((M1, M10, M))
```


results in the output

```
foo, foo, foo
```

The reverse technique, however, is not useful. That is, suppose values are assigned to array elements whose indices are fully specified, for example,

```
*s(1) := 11, *s(2) := 12
```

Then an array reference `*s(X)` will retrieve the last value assigned to an array element of the form `*s(_)`. For example,

```
W = *s(X)
```

binds `W` to `12`, and

```
11 = *s(X)
```

fails since it attempts to unify `11` and `12`.

However, a related technique is useful. Suppose a partially instantiated structure is assigned to a family of array cells by virtue of the fact that a variable index is used, for example

```
*s(1,_) := (a,_)
```

The structure may be retrieved, uninstantiated parts may be instantiated in various ways, and the results may be stored in members of the family of array cells that originally contained the partially instantiated structure. Given the above assignment, we may fill out the structure `(a, _)` differently for elements `*s(1,1)` and `*s(1,2)`:

```
(X,_) = *s(1,1), *s(1,1) := (X, b)
```

and

```
(X,_) = *s(1,2), *s(1,2) := (X, c)
```

This technique allows for progressive differentiation of array elements and their contents.

6.6 Summary

CHAPTER 7

Macros

Tokio has a macro facility that allows functions and relations to be defined so that uses of these functions and relations are expanded in line in the clauses in which they appear. We here first discuss functions, and then turn to relations defined as macros.

7.1 Defining and Applying Functions: `$function`

A function definition is introduced by the keyword `$function` and is terminated with a full stop. It has the form

```
$function <expression in X1, ..., Xn> = <return value, Z>
:-
                              <goal sequence in X1, ..., Xn
determining Z>.
```

Here the variables X_1, ..., X_n are variables that may be thought of as the formal parameters of the function. The variable `Z` gets bound to the function value. Normally, the expression to the left of the '=' is the function name with the formal parameters as arguments, so the head of the defining clause has the form

```
<function name> (X1, ..., Xn) = Z
```

7.1.1 Function Applications that Return Values One way in which a function may be applied within a clause is as one term in an equality expression:

```
.... :- ..., <expression in a1, ..., an> = Z,...
```

Here a_1, ..., a_2 may be variables or constants. The most common form for the expression to take is

```
<function name> (a1, ..., an)
```

For example, the following function increments its argument by one:

```
$function incr(I) = I1  :-  I1 = I + 1.
```

Suppose `incr` is applied in the definition of `t`:

```
t(X) :- incr(X) = W, write(W).
```

(Note that the order of `incr(X)` and `W` is immaterial.) Then the compiler replaces `incr(X) = W` with its definition to give what would have resulted if `t` had been defined by

```
t(X) :- W = X + 1, write(W).
```

As a second example, suppose function `foo` is defined by

```
$function foo(I,J) = I1 :- g(I,X), h(J,Y), I1 = X + Y.
```

We assume `g` and `h` are defined by their own clauses, say

```
g(1,1).
g(2,2).

h(1,2).
h(2,1).
```

Now suppose `foo` is used in

```
t :- X = foo(1,2), write(X).
```

Then the compiler uses the definition of `foo` to expand `X = foo(1,3)`, in effect giving


```
t :- g(1,X1), h(2,Y), X = X1 + Y, write(X).
```

Functions may also appear in arithmetical expressions, for example,

```
3 + foo(1,1) * foo(2,2) > incr(2)
```

Given the above definitions of `incr` and `foo`, this is expanded to

```
g(1,X1), h(1,Y1), I1 = X1 + Y1,
g(2,X2), h(2,Y2), I2 = X2 + Y2,
I3 = 2 + 1,
I4 = 3 + I1 * I2,
I4 > I3
```

Again, a function expression may occur as the argument of a predicate or even as the argument of another function. For example,

```
write(incr(foo(2,1)))
```

is expanded to

```
g(2,X), h(1,Y), I1 = X + Y, I2 = I1 + 1, write(I2)
```

It is sometimes beneficial to declare an operator that is then given a meaning in one or more function definitions. Consider, for example, the following

```
:- op(100, xfy, '::').

$function  X::first  = U :-   X = (U,V,W).
$function  X::second = V :-   X = (U,V,W).
$function  X::third  = W :-   X = (U,V,W).
```

Here three zero-ary functions, `first`, `second`, and `third`, are defined. They occur as the right operands of the infix operator `::` ; these function definitions may thus be viewed as defining the meaning of `::` . Note that the left operand of `::` is expected to be a structure of the form `(U,V,W)`. If the left operand is not of this form, then an attempt to evaluate one of the functions `first`, `second`, or `third` results in failure. Function `first` simply returns the first element in a structure of the form `(U,V,W)`; `second` and `third` return the second and third elements, respectively.

Now consider the clause

```
t :- A = (2,4,6), W = 2 * A::first + 3 * A::second -
A::third, write(W).
```

The Tokio compiler is smart enough to rewrite this (in light of the definitions of `first`, `second`, and `third`) as

```
t :- W = 2 * 2 + 3 * 4 - 6, write(W).
```

Predefined operators can also be exploited. Consider, for example,

```
$function (if Cond then Yes else No) = Reply :-
                    if  Cond  then  Yes  =  Reply  else  No  =
Reply.
```


Suppose this functional expression is used in

```
t :- g(1,X), g(2,Y), write(if Y>X then 1 else 0).
```

(We assume that `g` is defined as above.) The compiler expands this to:

```
t :- g(1,X), g(2,Y),
     {if Y > X then 1 = Reply else 0 = Reply},
     write(Reply).
```

7.1.2 Function Definitions as Conditional Equalities We have thus far thought of functions largely in terms of expressions that are evaluated to produce return values. It should be kept in mind, however, that the function facility in Tokio is a macro facility that expands expressions in accordance with what may be viewed as conditional equations. For a function definition

```
$function LHS = RHS :- condition
```

states that the left hand side (`LHS`) of the equation may be replaced by the right hand side (`RHS`) if the goal `condition` succeeds. Bindings made to `LHS` are in force when `condition` is evaluated, and bindings established by this evaluation are in force when `RHS` replaces `LHS`. Alternatively, if the function is applied as part of an equation, it is possible for `RHS` and `LHS` both to have bindings, since the head of the function definition (i.e. the entire equality) then matches the equality constituting the function application. Further bindings may then be imposed on either `LHS` or `RHS` as a result of evaluating `condition`.

As an example, suppose we have a relation `nth(N,List,Element)`, whose logical reading is : `Element` is the *N*th element in list `List` (where the head of `List` is considered the zeroth element). This may be defined by

```
nth(0, [H|T], H) :- !.

nth(N, [H|T], H1) :- N1 = N - 1, nth(N1,T,H1).
```

Now, in general, if we have a relation `f` in n variables and, for each tuple of values for the first *n-1* variables, there is at most one value for the nth variable, then we may regard `f` as a function in the first n-1 variables whose function value is the value of the nth variable. The relation *nth* meets these requirements, so it makes sense to define

```
$function nth(N,L) = E :- nth(N,L,E).
```

Now notice that, in the definition of the relation `nth`, arithmetic is performed on the first argument, so this argument must be an input argument. Also, the second argument is most naturally used as an input argument, leaving the third as the only natural output argument. Thus a typical use of the `function nth` would be, for example,

```
nth(2,L) = E
```

where `L` is bound to a list and `E` is unbound. This would be translated into

```
nth(2,L,E1), E1 = E
```

and, when evaluated, would bind `E` to the value of the second element in list `L`. But suppose the second element in `L` is an unbound variable and `E` is bound to some value. Then the above would bind the second element of `L` for this value. In

this case,

```
nth(2,L) = E
```

would not fit the paradigm of a function expression that evaluates to a value, to which `E` is then bound. Rather, it would conform to the more general paradigm of unifying the two sides of an equation.

7.1.3 Function Applications and Temporal Operators Finally, functions may be used together with temporal operators. As an example, consider again the function `nth` discussed in the last paragraph, and suppose we have

```
@nth(2,L) = E
```

This is expanded to

```
nth(2,L,E1), @E1 = E
```

Suppose again that the second element of list `L` is an unbound variable and that `E` is bound to a value. Then the effect of the above is to make the value of the second element in `L` at the next time point equal to the current value of `E`.

7.2 Defining and Invoking Macro Relations: `$define` and `$clause`

Definitions of relation macros are introduced by the keyword `$define`. Relation macros may be used to save the expense of extra calls (as macros are used in LISP). They are also convenient in that relation macros, unlike normal relations or predicates, may take static variables as arguments. We illustrate this latter point after discussing how relation macros are defined and used in general.

7.2.1 Simple Form: `$define` without `$clause` In its simplest form, a relation macro definition consists of a single clause introduced by `$define`, for example,

```
$define sum(I1, I2, Out) :- Out = I1 + I2.
```

Suppose this is used in the clause

```
t1(X, Y) :-  sum(X, Y, Res), write(Res).
```

Then the goal `sum(X, Y, Res)` is expanded inline according to the definition of `sum` to give

```
t1(X,Y) :-  Res = X + Y, write(Res).
```

Again, if `sum` is used in

```
t2 :-  sum(1, 2, R), write(R).
```

then the goal `sum(1, 2, R)` is expanded inline to give

```
t2 :- R = 1 + 2, write(R).
```

Note that the Tokio compiler recognizes constants and does *not* introduce intermediate variables to accommodate them.

7.2.2 Special Relations Subsidiary to Macro Relations: `$clause` Only single-clause macro definitions are allowed with `$define`, as one would expect since the definition supplies the code for inline expansions. But suppose, for example, we wish `sum` to be written so that it signals whether or not `I1 > I2`; suppose it writes a `1` if `I1 > I2`, and writes a `0` otherwise. As a normal relation, then `sum` would be defined as

```
sum(I1, I2, Out) :- I1 > I2, !, write(1), nl, Out = I1 +
I2.

sum(I1, I2, Out) :- write(0), Out = I1 + I2.
```

We may write this as a macro by first of all factoring out what is common to these two clauses; this gives the `sum` clause following `$define`:

```
$define sum(I1, I2, Out) :- H, Out = I1 + I2.
```

The `H` occupies the position where the code for the two clauses differ. We define this code using the `$clause` keyword:

```
$clause (H :- I1 > I2, !, write(1))

$clause (H :- write(0)).
```

In fact, the two `$clause` clauses are part of the `$define` definition. When `$clause` is used, the clause following `$define` must be enclosed in parentheses. Note that each clause introduced by a `$clause` is also enclosed in parentheses. Any *variable* (beginning with a capital letter or a '_') name would do in place of `H`. Finally, only one '.' should appear in the macro definition, and this at the very end, after the last *£clause* clause. Thus the new definition of the macro `sum` should be:

```
$define (sum(I1, I2, Out) :- H, Out = I1 + I2)

$clause (H :- I1 > I2, !, write(1))

$clause (H :- write(0)).
```

Suppose this newly defined `sum` is used in

```
t1(X, Y) :- sum(X, Y, Res), write(Res).
```

Then `sum` is expanded inline in `t1` to include a call to a relation corresponding to `H`, but with a (peculiar) name, say `$0t`, selected by the macro-expansion facility:

```
t1(X, Y) :- $0t(X, Y), Res = X + Y, write(Res).
```

The procedure `$0t` is defined to cover the `H` clauses for this use of `sum`:

```
$0t(U, V) :-  U > V, !, write(1).

$0t(U, V) :- write(0).
```

Suppose, on the other hand, `sum` is used with constant inputs in:

```
t2 :-  sum(1, 2, R), write(R).
```

Then the macro-expansion facility defines a different procedure, say `$1t`, to cover the `H` clauses for this use of `sum`, and produces:


```
t2 :-  $1t, R = 1 + 2, write(R).
$1t :-  1 > 2, !, write(1).
$1t :- write(0).
```

Note that the constants 1 and 2 are hard-coded into the definition of $1t. Also note that two $1t clauses, corresponding to the two H clauses, are produced, even though the two together are equivalent (because '1 > 2' is always false) to the last clause alone. In general, for every occurence of sum in a normal Tokio program clause, the macro-expansion facility defines a new procedure with a unique name to cover the clauses introduced by $clause.

7.2.3 Scope of Variables in Macro Relation Definitions Notice that, in the example above, the compiler was able to identify the variables I1 and I2 in the first H clause with the variables I1 and I2 in the definition of sum. When sum was used in t1 and t2, the compiler was also able to distinguish between the case in which the actual arguments of I1 and I2 were variables (i.e. in t1) and the case in which they were constants (i.e. in t2). In the former case, to cover the H clauses, it defined a procedure $0t with two arguments corresponding to the two actual arguments. In the latter case, it defined a zero-ary procedure $1t that had the constant actual arguments written into it. In either case, the compiler had to identify I1 and I2 in the first H clause with the I1 and I2 in the definition of sum. In general, the compiler remembers the variable names in the head of a $define clause. The same variable names occurring in a $clause clause of the same definition are required to be bound to the same values. In the present case, the variables in the head of the $define clause are I1, I2, and Out. Of these, only I1 and I2 appear in a $clause clause (specifically, the first). When the program clause t1 is compiled, I1 and I2 are associated with the variables X and Y, respectively. Since X and Y are variables, the procedure covering the H clauses in this case, that is, $0t, must have two variable arguments if it is to have access to the values associated with I1(X) and I2(Y). On the other hand, when the program clause t2 is compiled, I1 and I2 are associated with the constants 1 and 2, respectively. Then the procedure covering the H clauses in this case, that is, $1t, need have no arguments since 1 and 2 can be written into it at the appropriate places.

It is important to keep in mind that only the variables in the head of the clause following $define are recognized in the clauses introduced by $clause. For example, suppose we define

```
$define (sum(I1, I2, Out) :- I3 = I1 + 1, H, Out = I1 + I2)
$clause (H :- I1 > I2, !, write(1), write(I3))
$clause (H :- write(0)).
```

Given the program clause

```
t :- sum(2, 1, R), write(R).
```

macro expansion produces

```
t :- I3 = 1 + 2, $0t, R = 2 + 1, write(R).
```



```
$0t :- 2 > 1, write(1), write(X).
$0t :- write(0).
```

The first `$0t` clause contains a variable (called `X` here - any other variable name would do) that does not occur as an argument and is not bound in the clause. This variable is the image of the `I3` in the first `H` clause and was not recognized as anything special because `I3` does not occur in the head of `sum`.

7.2.4 Constants in the Heads of Macro Clauses The head of the clause introduced by the `$define` may contain one or more constants, for example

```
$define sum(1, I1, I2, Out) :- Out = I1 + I2.
```

Given the program clause

```
t :-  sum(X, 1, 2, R), write(S), write(R).
```

Macro expansion produces

```
t :-  R = 1 + 2, write(1), write(R).
```

A single macro relation may contain any number of special relations defined using `$clause`. The `$clause` clauses all follow the `$define` clause, and a '.' appears only once, after the last `$clause` clause. For example, we could define

```
$define (sum(I1, I2, Out) :- H, Out = I1 + I2, G)

$clause (H :- I1 > I2, !, write(1))
$clause (H :- write(0))

$clause (G :- I1 > 0, !, write(a))
$clause (G :- write(b)).
```

Given the program clause

```
t :- sum(1, 2, R), write(R).
```

macro expansion produces

```
t :- $0t0, R = 1 + 2, $0t1.

$0t0 :- 1 > 2, !, write(1).
$0t0 :- write(0).

$0t1 :- 1 > 0, !, write(a).
$0t1 :- write(b).
```

As is evident from this example, no special provision is needed to handle multiple special relations defined by `$clause`.

7.2.5 Recursive Special Relations A special relation defined with `$clause`

may be recursive, for example

```
$define (foo :- write(start), nl, H)
$clause (H :- write('enter number'), read(N),
              if N < 5 then (true && H)
        ).
```

Given the program clause

```
t :- foo, fin(nl, write(end)).
```

macro expansion produces

```
t :- write(start), nl, $0t, fin(nl, write(end)).

$0t :- write('enter number'), read(N),
         if N < 5 then (true && $0t).
```

As long as the members entered are less than `5`, `$0t` will chop the current interval and call itself at the next time point.

7.2.6 Variable vs. Constant Names as Special Relation Names Notice that, when the name of a special relation defined with `$clause` begins with an uppercase letter or underscore (and so is a valid variable name), and as long as the special relation needs access to the values of only the variables in the head of the macro relation, then there is no need for such a special relation to have arguments. In fact, an attempt to define such a special relation with arguments is considered a syntax error. However, there are times when we would like values of non-head variables to be communicated to special relations defined with `$clause`. In that case, the name of the special relation must begin with a lowercase letter (like a normal program relation or predicate name) and, as long as the names of the variables in the head of the macro relation do not occur in the special relation, all communication is through the arguments (just as with a normal program relation). For example, consider

```
$define (sum(I1, I2, Out) :- I3 = I1 + 1, h(I1, I2, I3),
Out = I1 + I2)

$clause (h(X, Y, Z) :- X > Y, !, write(1), write(Z))
$clause (h(_, _, _) :- write(0)).
```

Then the program clause

```
t1 :- sum(2, 3, R), write(R).
```

is expanded to

```
t1 :- I3 = 2 + 1, h(2, 3, I3), R = 2 + 3, write(R).

h(X, Y, Z) :- X > Y, !, write(1), write(Z).
h(_, _, _) :- write(0).
```


Notice that the definition of `h` is unchanged by the macro expansion and that the constants in the definition of `t1` have been written into the `h(2,3,R)` goal. In fact, there is no difference between defining `h` as a special relation in the definition of the macro relation `sum` and defining `h` as a normal program clause. In either case, an `h` goal is included in the expansion of `sum(2, 3, R)`, and (perhaps surprisingly) the `h` relation is available to any clause in the program.

However, if the a variable occurring in the head of the macro relation also occurs in the body of a lowercase special relation, then the appropriate value is written in place of the occurrence of the variable when the macro is expanded. For example, consider

```
$define (sum(I1, I2, Out) :- I3 = I1 + 1, h(I3), Out = I1 +
I2)

$clause (h(Z) :- I1 > I2, !, write(1), write(Z))
$clause (h(_) :- write(0)).
```

Then the program clause

```
t1 :- sum(2, 3, R), write(R).
```

is expanded to

```
t1 :- I3 = 2 + 1, h(I3), R = 2 + 3, write(R).

h(Z) :- 2 > 3, !, write(1), write(Z).
h(_) :- write(0).
```

Here we may think of the scope of `I3` (or any variable in the head of the macro relation) as the entire macro definition, including the `$clause` clauses.

The extended scope of a variable in a macro relation's head causes problems if the same variable name is used for a variable in the head of a special relation clause. For the occurrences of this variable in the body of the special relation clause are treated as occurrences of the variable in the head, not of the special relation clause, but of the macro relation. Suppose, for example, we intend to write a macro relation to compute the factorial of a natural number, and we define:

```
$define (fact(N) :- fact(N, F), write(F))

$clause (fact(N, F) :- N =< 0, !, F = 1)
$clause (fact(N, F) :- N1 = M - 1, fact(N1, F1), F = N *
F1).
```

Then the program clause

```
t :- fact(3).
```

is expanded to

```
t :- fact(3, F), write(F).
fact(N, F) :- 3 =< 0, !, F = 1.
```



```
fact(N, F) :- N1 = 3 - 1, fact(N1, F1), F = 3 * F1.
```

Notice that all occurrences of `N` in the bodies of the special relation clauses have been replaced by `3`. If `t` is evaluated, a non-terminating recursion will result since the first special clause always fails. This problem is avoided if `N` in the special relation clauses is renamed, say to `M`. Note, however, that none of the variables in the clause defining the 2-ary `fact` relation is intended to have a scope extending beyond the clause in which it occurs. Thus the safest and best thing to do in this case and all similar cases is define `fact` as a normal program relation.

Another problem with a lowercase special relation is that its definition is written into the database everytime the associated macro relation is expanded. This results in redundant definitions. Although correct behavior is not jeopardized, this result is obviously undesirable.

A final problem with a lowercase special relation is that, once the associated macro relation is expanded, its definition is indistinguishable from, and may clash with, the definition of a normal program relation. Consider, for example,

```
$define (foo :- write(1), foo1)
$clause (foo1 :- write(2)).

foo1 :- write(3).

t :- foo, foo1.
```

When the `foo` macro is expanded, this program becomes

```
t :- write(1), foo1.

foo1 :- write(3).
foo1 :- write(2).
```

Obviously the `foo1` clause derived from the special relation can never be executed. In such cases, where the special relation needs no argument, it is best to use an uppercase special relation. Then, when the macro relation is expanded, a weird name (such as `$0t`) is used for the special relation, thus nearly (as long as the programmer does not use the same weird names) eliminating the possibility of name clashes.

There are thus three choices for special relations: use an uppercase special relation, use a lowercase special relation, or use a normal program relation instead of a special relation. An uppercase special relation should be used when variables whose scope is the entire macro definition — that is, variables appearing in the macro relation's head — are used and the values of variables in the body of the macro relation are not needed — so no arguments are needed for the special relation. A normal, not special, relation should be used when no variable has a scope beyond the clause in which it occurs. And a lowercase special relation should be used when some variables in the relation clauses have normal and some have extended scope.

7.2.7 Macro Relation Goals with Static Variable Arguments The most

important use of macro relations is to allow for goals in which static variables occur. Recall that a static variable may not appear as an argument in a goal whose functor is a system-defined relation or a normal user-defined relation. For example, evaluating `write(*u)` would cause `*u` itself, not the value of `*u`, to be output. To produce concise, readable code, however, we could define

```
$define Output(X) :- Y = X, write(Y).
```

Suppose our program also includes

```
t :- *u := 2, output(*u).
```

Then, after macro expansion, our program is

```
t :- *u := 2, Y = *u, write(Y).
```

It is generally good practice to define macro relations to manipulate the values of static variables.

7.3 Summary

Table of Contents

