(a) Prove by induction on *xs* that this specification specifies `filter` uniquely.

(b) Show that `filter` is implemented by

```
filter p xs = fold (++) []  (map f xs)
                  where f x= [x], if p x
                           = [],  otherwise
```

8. For each of the functions given below:

   (a) Write down equations to show their values in the cases when

   $ys = []$
   $ys = us{+}{+}vs$
   $ys = [y]$

   (b) Show (by list induction) that there is at most one function that satisfies your answers to (a).

   (c) Write a similar equation for the case when $ys = y{:}\,zs$, and show how it is implied by the equations given in (a).

   (d) Use (c) to write down a recursive Miranda definition of the function.

   (e) Prove by induction that your definition satisfies the properties in (a).

   (f) Use `map` and `fold` to write a non-recursive definition of the function.

   (g) Use standard properties of `map` and `fold` to show that your definition in (f) satisfies the properties given in (a).

   Here are the functions:

   - `length`: `[*] -> num`, `length` *ys* is the length of *ys*.
   - `prod`: `[num] -> num`, `prod` *ys* is the product of the elements of *ys*. (NOTE: consider carefully what `prod []` should be.)
   - `count`: `* -> [*] -> num`, `count` *x ys* is the number of occurrences of *x* in *ys*.
   - `split`: `(* -> bool) -> [*] -> ([*], [*])`. If `split` *p ys*=(ys_1,ys_2), then `merge`(ys_1, ys_2, *ys*), and for every *y*, if *y* is an element of ys_1 then (*p y*), while if *y* is an element of ys_2 then ¬ (*p y*).
   - `all`: `(* -> bool) -> [*] -> bool`, (`all` *p ys*) iff for every element *y* of *ys* we have *(p y)*.
   - `some`: `(* -> bool) -> [*] -> bool`, (`some` *p ys*) iff for some element *y* of *ys* we have *(p y)*.
   - `sum`: `[*] -> num`, `sum` *ys* is the sum of the elements of *ys*.

9. Consider `fold (&) True :: [bool] -> bool`. (`&` is associative, and `True` is its identity.) Remember that in Miranda it is possible to have infinite lists, for instance `trues` where

   ```
   trues = True:trues
   ```

   (all its element are True).

   Show that if *bs* is an infinite list of type [bool], then

`foldr (&) True (False:`*bs*`) = False`	but
`foldl (&) True (False:`*bs*`)`	goes into an infinite loop.

10. (a) Define the polymorphic function `reverse` using `foldr`.

    (b) Define the polymorphic function `reverse` using `foldl`.

    (c) Which is more efficient and why?

11. Define the higher-order function `map` without explicit recursion by using the higher-order function `foldr` (with a non-associative argument).

12. In a version of the game Mastermind, one player thinks of a four-digit number, while the other player repeatedly tries to guess it. After each guess, player 1 scores the guess by stating the number of bulls and cows. A bull is a correct digit in the correct place and a cow is a correct digit in an incorrect place. No digit is scored more than once. For example, if the secret code is 2113, then:

    1234 scores 03
    1111 scores 20
    1212 scores 12

    Construct a function `score` which takes a code and a guess and returns the number of bulls and cows. (Your function `score` should be written using higher-order functions.)
    You may find it helpful to use the `--` construct. `--` is a list subtraction operator. The value of *xs*`--`*ys* is the list which results when, for each element *y* in *ys*, the first occurrence of *y* is removed from *xs*. For example,

    ```
    [1,2,1,3,1,3]--[1,3] = [2,1,1,3]
    "angle"--"l" ++"l" = angel
    || "xyz" is short for ['x','y','z']
    ```

13. (Advanced) This is an exercise in using both polymorphism and higher-order functions. The question investigates *predicates* on Miranda types: a predicate on type `*` is understood as a function from `*` to `bool`:

    ```
    pred * == (* -> bool)
    ```

Suppose `f :: bool -> bool -> bool`. Then `f` can be extended to a function on predicates by applying it *pointwise*:

```
ptwise :: (bool->bool->bool)->(pred *)->(pred *)->(pred *)
ptwise f p q x = f(p x)(q x)
```

(Experiment: define this in Miranda, and try `ptwise ::`. Miranda realizes that this definition can be used much more widely than just when `f :: bool -> bool -> bool`. Also, why does the type of `ptwise` seem to give it three arguments, whereas the definition gives it four?)
If p, q :: `pred *`, let us write

$$p \Rightarrow q \text{ iff } \forall x :: *((p\ x) = \texttt{True} \rightarrow (q\ x) = \texttt{True})$$

(a) Translate the following specifications into English, and write Miranda definitions for functions to implement them:

```
all :: (pred *) -> (pred [*])
||pre:  none
||post: (all p t)=True <->
||      (A)x. ((E)n. In-At(x,t,n)->(p x)=True)

some :: (pred *) -> (pred [*])
||pre:  none
||post: (some p t)=True <->
||      (E)x. ((E)n. In-At(x,t,n) & (p x)=True)
```

(b) Prove that for all p, q : `pred` $*$,

`all`(`ptwise (\/) `p q) $\Rightarrow$ `ptwise (\/) (all `p`) (some `q`)`
`ptwise (&) (all `p`) (some `q`)` $\Rightarrow$ `some (ptwise (&) `p q`)`

Describe in English what these results mean.

Chapter 9

Specification for Modula-2 programs

We now move on to imperative programming, using the Modula-2 language. (This material also applies to Pascal and Ada programs.) We will not describe the features of Modula-2 here because there are already many books about it.

9.1 Writing specifications for Modula-2 procedures

The general idea is the same as for Miranda: a specification has some typing information, a pre-condition and a post-condition. These can be conveniently placed at the header of the procedure as follows:

```
PROCEDURE CardMin(x,y: CARDINAL):CARDINAL;
(*pre: none
 *post: (result = x \/ result = y) & (result <=x & result <=y)
 *)
BEGIN
  IF x<=y THEN RETURN x ELSE RETURN y END
END CardMin;
```

The principles here are exactly the same as in Miranda, with three minor points of difference. First, the typing information, that is,

`PROCEDURE CardMin(x,y: CARDINAL):CARDINAL;`

is compulsory in Modula-2. Second, comments look different: they are between `(*` and `*)`, instead of being after `||`. Third, we are using the word `result` in post-conditions to mean the value returned by the procedure. This means that it would be inadvisable to have a *variable* called `result` because of the confusion that would arise.

> `result` has a special meaning in post-conditions of functions: it means the value returned.

Variables changing

What is not apparent from this example is that there is a big difference between Miranda and Modula-2: Modula-2 has variables that *change their values*. Therefore, our reasoning must be able to cope with symbols that take different values at different times. In general, because a variable may change its value many times during the computation, there may be lots of different times at which we may wish to put our finger on the value and talk about it. There is a general technique for doing this. But in a procedure specification, there are really only two values to talk about, before (on entry to) and after (on return from) the procedure, and we use a special-purpose notation to distinguish these.

A pre-condition must only talk about the values before the procedure is executed, so when a variable is used in a pre-condition it means the value *before*. A post-condition will usually want to compare the values before and after, and this is where the special notation comes in. A variable with a *zero* (for example, x_0 or `x_0`) means the value *before*; an unadorned variable (for example, x or `x`) means the value *after*. We shall be consistent in using unadorned variables to denote the value *now* (in the pre-condition, 'now' is the time of entry; in the post-condition it is the time of return), and in using various adornments such as the zero to show the value at some *other* time.

The following are two examples:

```
PROCEDURE Swap (VAR x,y: INTEGER);
(*pre: none
 *post: x=y_0 & y=x_0
 *)
PROCEDURE Sqrt (VAR x: REAL);
(* Replaces x by an approximation to its square root.
 * epsilon is a global variable.
 * pre:  x>=0 & epsilon>0
 * post:  x>=0 & | x^2-x_0|< epsilon & epsilon = epsilon_0
 *)
```

Some variables are not expected to change

To specify that a variable does not change, you say so in the post-condition: for example, `epsilon = epsilon_0` says that *epsilon* does not change (value on return = value on entry). But this could get out of hand, so let us adopt the following two conventions.

First, if a global variable is not mentioned at all in the specification, then we assume an implicit specification that it should not change.

Second, if a parameter is called by value, then, again, we assume an implicit specification that it should not change. (That is why in `CardMin` we did not bother to write `x_0` or `y_0`.) (If you think about it, this assumption will seem pointless. Apparently, all the changes made to the parameter are local to the procedure and the caller can never notice them.)

9.2 Mid-conditions

When we implement the specifications, there is a very simple technique for reasoning. It generalizes the idea of pre- and post-conditions by using logical assertions that are supposed to hold at points in the middle of the computation, not just at the beginning or end. We call them *mid-conditions*. They are written as comments in the middle of the code.

The following is an implementation of `Swap`, with a complete set of mid-conditions:

```
PROCEDURE Swap (VAR x,y: INTEGER);
(* pre: none
 *post: x=y_0 & y=x_0
 *)
VAR z: INTEGER;
BEGIN        (*x=x_0 & y=y_0*)
  z:=x;      (*z=x_0 & y=y_0*)
  x:=y;      (*z=x_0 & x=y_0*)
  y:=z;      (*y=x_0 & x=y_0*)
END Swap;
```

You would not normally put in so many mid-conditions. There are just certain key positions where they are important — you have already seen two, namely entry and return (corresponding to pre- and post-conditions). With most simple straight-line sections of code such as this it is easy to omit the intermediate mid-conditions and fill them in mentally. But we can use the example to illustrate the reasoning involved.

Each mid-condition is supposed to hold whenever program control passes through that point — at least, provided that the procedure was called correctly, with the pre-condition holding. (Note that unadorned variables still denote the value 'now', that is, at the time when control passes through that point; zeroed variables denote the value on entry.) Does this work here?

The first mid-condition, $x = x_0 \wedge y = y_0$, holds by definition: we have only just entered the procedure, so the value of x has to be its value on entry, which is x_0 by definition.

Now look at the next mid-condition, $z = x_0 \wedge y = y_0$. To have arrived here, we must have started at the point where we had $x = x_0 \wedge y = y_0$, and then done the assignment $z := x$. It is not difficult to see that this is bound to

set up the mid-condition we are looking at (though there are formal systems in which this can be proved — in effect they define the meaning of the assignment statement).

The next mid-condition is similar, and finally we reach the final mid-condition, which is the post-condition. By this stage we know that by the time the program returns it must have set up the post-condition.

Note the 'stepping stone' nature of the reasoning. To justify a mid-condition we do not look at *all* the computation that has gone before, but, rather, at the preceding program statement and the mid-condition just before that.

Conditionals

Here is an example with an `IF` statement.

```
PROCEDURE IntMax (x,y: INTEGER):INTEGER;
(*pre: none
 *post:  (result = x_0 \/ result = y_0) &
 *       (result >=x_0 &  result >=y_0)
 *)
BEGIN
  IF x>=y
  THEN (*x>=y*) RETURN x (* result = x_0 &  result >=y_0*)
  ELSE (*x<y*) RETURN y  (* result = y_0 &  result >x_0*)
  END
END IntMax;
```

There are two branches of the code, the `THEN` and `ELSE` parts, and in each we can write a mid-condition based on the condition '`IF` $x \geq y$'. For instance, when we enter the `THEN` part, that can only be because the condition has evaluated as `TRUE`: so we know at that point that $x \geq y$. (This is relying on the fact that there are no side-effects when the condition $x \geq y$ is evaluated.) After `RETURN x`, we know that the result is x and also, because we knew $x \geq y$, that `result` $\geq y$. The other branch, the `ELSE` part, is similar. On entering it, we know that the condition evaluated as `FALSE`, so $x < y$.

Finally, we must show that the post-condition is set up. There are *two* return points, each with a different mid-condition. But it is a matter of logic (and properties of $\geq$) to show that

$$\begin{aligned} \texttt{result} = \;& x \wedge \texttt{result} \geq y \\ & \rightarrow (\texttt{result} = x \vee \texttt{result} = y) \wedge \texttt{result} \geq x \wedge \texttt{result} \geq y \\ \texttt{result} = \;& y \wedge \texttt{result} > x \\ & \rightarrow (\texttt{result} = x \vee \texttt{result} = y) \wedge \texttt{result} \geq x \wedge \texttt{result} \geq y \end{aligned}$$

9.3 Calling procedures

When you specify a procedure, the zero convention is very convenient; and throughout that procedure you use the zeroed variables for the values on entry. But when you *call* the procedure, you must be careful about the zeroes in its specification: because you now have two contexts, the called procedure and the calling context, in which zero has different meanings.

The following is an example of a rather simple sorting algorithm. The first procedure, `Order2`, sorts two variables, and the second, `Order3`, uses `Order2` to sort three variables.

```
PROCEDURE Order2 (VAR x,y: INTEGER);
(*pre: none
 *post: ((x=x_0 & y=y_0) \/ (x=y_0 & y=x_0)) & x<=y
 *)
 BEGIN              (*x=x_0 & y=y_0*)
   IF x>y
   THEN (*x_0>y_0*) Swap(x,y);     (*x=y_0 & y=x_0 & x<y*)
(* ELSE x_0<=y_0*)                (*x=x_0 & y=y_0 & x<=y*)
   END              (*either way, x<=y*)
 END Order2;
```

Before giving the definition of `Order3`, let us outline the idea. We are ordering x, y and z. *If* we can arrange for z to be the greatest, that is, $x \leq z \wedge y \leq z$, then the rest is easy: just order x and y. So this condition becomes a key objective in our computation strategy, dividing the task into two. It appears as the second mid-condition. You can probably believe that this objective is achievable using `Order2`(y,z) and `Order2`(x,z), and we shall show this more carefully.

On this analysis, the first two mid-conditions are slightly different in character. The second is a computational objective, used to specify the task of the first part of the code. As a condition it does not express everything known at that point, but, rather, just something achievable that gives us what we need to be able to finish off the problem. The first mid-condition, on the other hand, is more to help us reason that our code, once written, really does work:

```
PROCEDURE Order3 (VAR x,y,z: INTEGER);
(*pre: none
 *post: x,y,z  are a permutation of x_0,y_0,z_0 & x<=y<=z
 *)
BEGIN
  Order2(y,z);      (*y<=z*)
  Order2(x,z);      (*y<=z & x<=z*)
  Order2(x,y)       (*x<=y<=z*)
END Order3;
```

How do we know that `Order3` works? (Are you actually convinced at this stage?) Let us dispose straight away of the specification that x, y and z are a permutation of x_0, y_0 and z_0 (that is, the same values, possibly rearranged). Although it is actually quite difficult to express this in pure logic, it is quite clear that each call of `Order2` just permutes the variables, so that is all the three consecutive calls can do. The real problem is knowing that the order is correct in the end.

The first call certainly sets up the first mid-condition, $y \leq z$, but how do we know that the second call does not spoil this? We must look at the specification of `Order2`, which says (after we have substituted the actual parameter z for the formal parameter y)

$$((x = x_0 \wedge z = z_0) \vee (x = z_0 \wedge z = x_0)) \wedge x \leq z$$

The zero here denotes the value on entry to the (second) call of `Order2`, but we are reasoning about `Order3`, trying to prove it correct: so for us the zero could also denote the value on entry to `Order3`. To avoid the conflict, you have to invent some new names: say x_1, y_1 and z_1 for the values of x, y and z between the first two `Order2`s: At that point we have, by the mid-condition, $y_1 \leq z_1$, and this is *eternally true* — because y_1 and z_1 are unchanging values not computer variables.

Now what `Order2` sees as x_0 and z_0 are — in our `Order3` context — x_1 and z_1. Hence on return from the second `Order2`, we can use its post-condition to write

$$((x = x_1 \wedge z = z_1) \vee (x = z_1 \wedge z = x_1)) \wedge x \leq z \wedge y = y_1$$

So far, although we have said a lot by way of explanation, all that has happened has been some notational manipulation and with practice you should be able to do it automatically. What comes next is real logic; Figure 9.1 contains a box proof that shows $y \leq z \wedge x \leq z$.

More compactly, we want to show at this point that $y \leq z$, that is, $y_1 \leq z$. Since $y_1 \leq z_1$, it is sufficient to show that $z_1 \leq z$ (that is, `Order2`$(x,\ z)$ cannot decrease the value of z; you would expect this intuitively, but we can also prove it). There are two cases. If $(x = x_1 \wedge z = z_1)$, that is, `Order2` did not do a swap, then $z_1 = z$. In the other case, we have $(x = z_1 \wedge z = x_1 \wedge x \leq z)$, so $z_1 = x \leq z$.

We have now proved that the second mid-condition, $y \leq z \wedge x \leq z$, is set up correctly. For the third mid-condition, the fact that z is greater than both x and y is unaffected whatever `Order2`(x, y) does, while it also ensures $x \leq y$. Hence, finally, $x \leq y \leq z$.

> You may invent new *logical* constants as names for intermediate computed values. This is like Miranda '`where` ...' notation.

1	$y_1 \leq z_1$				
2	$(x = x_1 \wedge z = z_1) \vee (x = z_1 \wedge z = x_1)$				
3	$x \leq z$				
4	$y = y_1$				
5	$y \leq z_1$				eqsub in 1
6	$x = x_1 \wedge z = z_1$			$x = z_1 \wedge z = x_1$	
7	$z = z_1$	$\wedge\mathcal{E}$		$x = z_1$	$\wedge\mathcal{E}$
8	$y \leq z$	eqsub in 5		$z_1 \leq z$	eqsub in 3
9				$y \leq z$	trans $\leq$
10	$y \leq z$				$\vee\mathcal{E}(2)$
11	$y \leq z \wedge x \leq z$				$\wedge\mathcal{I}(3, 10)$

Figure 9.1 $y \leq z \wedge x \leq z$

9.4 Recursion

To deal with recursion, you use recursion variants (or induction) just as in Miranda. Recursively defined functions in Miranda translate readily into recursively defined function procedures in Modula, and the reasoning is the same in both cases. Actually, it is often more convenient to reason with the Miranda definitions, because the notation is much more economical. Consider, for instance, the Euclidean algorithm implemented recursively in Modula-2:

```
PROCEDURE gcd(x,y: CARDINAL):CARDINAL;
(*pre: none
 *post: result | x & result | y &
 *      (A)z:Cardinal. (z | x & z | y-> z | result)
 *recursion variant = y
 *)
BEGIN
  IF y=0 THEN RETURN x ELSE RETURN gcd(y,x MOD y) END
END gcd;
```

Proposition 9.1 The definition of `gcd` satisfies the specification.

Proof Both specification and definition are direct translations of those for the Miranda function `gcd` given in Chapter 5. (Note that the Miranda pre-condition $\mathtt{nat}(x) \wedge \mathtt{nat}(y)$ has been translated into typing information in Modula-2. Unlike Miranda, Modula-2 has special types `CARDINAL` and `INTEGER`, with `CARDINAL`s corresponding to `nat`.) We have already proved that the Miranda definition satisfied the Miranda specification. □

It is somewhat difficult at this stage to give sensible examples of recursion that genuinely use the new imperative features. The following is a rather artificial example:

```
PROCEDURE gcd1(VAR x,y: CARDINAL);
(*Replaces x by the gcd of x and y.
 *pre: none
 *post: x | x_0 & x | y_0 & (A)z:Cardinal. (z | x_0& z | y_0-> z | x)
 *recursion variant=y
 *)
VAR z: CARDINAL;
BEGIN
  IF y # 0 THEN
    z:=x MOD y;
    x:=y;
    y:=z;        (*x=y_0 & y=x_0 MOD y_0*)
    gcd1(x, y)
  END
END gcd1;
```

Proposition 9.2 The definition of `gcd1` satisfies the specification.

Proof If $y = 0$ then $x = \texttt{gcd}(x, 0)$ and so nothing has to be done. If $y \neq 0$, then by the usual reasoning with recursion variants we can assume that the recursive call `gcd1`(x, y) replaces x by the `gcd` of y_0 and x_0 `MOD` y_0, which, by the same argument as given in Chapter 5, is the `gcd` of x and y. □

9.5 Examples

The following procedure swaps the values of two variables without using any extra variables as storage space. Mid-conditions show very clearly how the sequence of assignments works:

```
PROCEDURE Swap (VAR x,y: INTEGER);
(* pre: none
 *post: x=y_0 & y=x_0
 *)
BEGIN          (*x=x_0 & y=y_0*)
  x:=x-y;      (*x=x_0-y_0 & y=y_0*)
  y:=x+y;      (*x=x_0-y_0 & y=x_0*)
  x:=y-x       (*x=y_0 & y=x_0*)
END Swap;
```

Walkies Square

Imagine a Walkies package with position coordinates X and Y, and procedures `Up` and `Right` for updating these:

```
VAR X,Y: INTEGER;
PROCEDURE Up(n: INTEGER);
(*pre: none
 *post: X=X_0 & Y=Y_0+n
 *)
 PROCEDURE Right(n: INTEGER);
(*pre: none
 *post: X=X_0+n & Y=Y_0
 *)
```

We can use mid-conditions to show that the following procedure returns with X and Y unchanged:

```
PROCEDURE Square(n: INTEGER);
(*pre: none
 *post:  ... & X=X_0 & Y=Y_0
 *)
BEGIN             (*X=X_0 & Y=Y_0*)
  Right(n);       (*X=X_0+n & Y=Y_0*)
  Up(n);          (*X=X_0+n & Y=Y_0+n*)
  Right(-n);      (*X=X_0 & Y=Y_0+n*)
  Up(-n)          (*X=X_0 & Y=Y_0*)
END Square;
```

It is reasonably clear that these mid-conditions are correct. But to justify this more formally you need to use the specifications of `Right` and `Up`. For instance, consider the call '`Right`$(-n)$'. In the specification for `Right`, X_0 and Y_0 mean the values of X and Y *on entry to* `Right`, and not, as we should like to use them in the mid-conditions, on entry to `Square`. But we do know (from the preceding mid-condition) that on entry to this call of `Right` X and Y have the values $X_0 + n$ and $Y_0 + n$(where X_0 and Y_0 are values on entry to `Square`), so we can substitute these into the post-condition for `Right`. Also, `Right` is called with actual parameter $-n$, so we must substitute this for the formal parameter n in the post-condition. All in all, in $X = X_0 + n \ \& \ Y = Y_0$ substitute

- $-n$ for n,
- $X_0 + n$ for X_0,
- $Y_0 + n$ for Y_0,

giving $X = X_0 \ \& \ Y = Y_0 + n$. This is the next mid-condition.

9.6 Calling procedures in general

A typical step of reasoning round a procedure call looks as follows:

$$\ldots\, mid1(x, y, z, \ldots)\ \ P(a, b, c, \ldots)\ \ \ mid2(x, y, z, \ldots)$$

Here $x, y, z, \ldots$ represent the relevant variables, and $a, b, c, \ldots$, expressions involving the variables, are the actual parameters in the call of P. We assume for simplicity that evaluating these actual parameters does not call functions that cause any side-effects. We have reasoned that *mid1* holds just before entry to P (imagine freezing the computer and inspecting the variables: they should satisfy the logical condition *mid1*, and we now want to reason that *mid2* will hold on return). We must do this by using the specification of P; however, that is written using the *formal* parameters of P, and the first step is to replace these by the actual parameters $a, b, c, \ldots$ to obtain the properties of $x, y, z, \ldots$

```
pre: preP(x,y,z,... )
post: postP(x,x_0,y,y_0,z,z_0,... )
```

(But the zeros in *postP* show values of x, y, and z *on entry to* P, and we shall have to allow for this.) Next we must show that *mid1* entails *preP*, in other words that *mid1* is sufficient to ensure that P works correctly. This is pure logic.

Next, we must work out what exactly we know on return from P, at the same time coping with possible notational clashes due to x_0, and so on, having different meanings in different places. Suppose x_1, y_1, z_1, ...are convenient names for the values of $x, y, z, \ldots$ before the call of P. Then the post-condition tells us that on return we have $postP(x, x_1, y, y_1, z, z_1, \ldots)$. But we also know, because x_1, and so on, are just names of values, that we have $mid1(x_1, y_1, z_1, \ldots)$. Hence, on return from P we know the following (and no more):

$$postP(x, x_1, y, y_1, z, z_1, \ldots) \wedge mid1(x_1, y_1, z_1, \ldots)$$

Our final task is to prove that this entails $mid2(x, y, z, \ldots)$. Again, this is pure logic.

To summarize, after manipulating the specification of P a little, we have two tasks in pure logic: prove —

$$mid1(x, y, z, \ldots) \rightarrow preP(x, y, z, \ldots)$$
$$postP(x, x_1, y, y_1, z, z_1, \ldots) \wedge mid1(x_1, y_1, z_1, \ldots) \rightarrow mid2(x, y, z, \ldots)$$

Thus the step between $mid1$ and $mid2$ (via P) really has the two logical steps, above, and a computational step (P) in the middle. The specification of P gets us from $preP(x, y, z, \ldots)$ to

$$postP(x, x_1, y, y_1, z, z_1, \ldots) \wedge mid1(x_1, y_1, z_1, \ldots).$$

9.7 Keeping the reasoning simple

When all the features of imperative programming are taken together, some of them can be quite complicated to reason about. There is a general useful principle:

> Keep the programming simple to keep the reasoning simple.

We have already seen some examples:

- It is simpler if you do not assign to 'call by value' parameters, even though Modula-2 allows you to (hence our default assumption that they do not change their values).
- It is simpler if functions, and hence expressions containing them, do not have side-effects. We assumed this when we were discussing `IF` statements — it is tricky if the condition has side-effects for the actual parameters.

When we say that these features make the reasoning more difficult, this applies even to the most superficial of reasoning. The effects they have are easy to overlook when you glance over the program. A classic source of error is careless use of global variables, because they tend to be updated in a hidden way, as a side-effect of a procedure.

9.8 Summary

- For Modula-2 the essential ideas of pre- and post-conditions (also recursion variants) are the same as for Miranda; *result* in a post-condition means the result of the procedure.
- Variables change their values, so a logical condition must always carry an idea of 'now', a particular moment in the computation. For pre- and post-conditions, 'now' is, respectively, entry to and return from a procedure.
- An unadorned variable always denotes its value 'now'.
- A zero on a variable indicates its value 'originally', that is, on entry to the procedure it appears in.
- Introduce new constant symbols (for example, variables adorned with 1s) as necessary to indicate values at other times.
- There are implicit post-conditions: variables not mentioned, and local variables, are not changed.
- Mid-conditions can be used as computational objectives ('post-conditions for parts of a procedure body') and to help reason correctness.
- In an `IF` statement, the test gives pre-conditions for the `THEN` and `ELSE` parts.

- When reasoning about procedure calls, there are three parts:
  1. notational manipulation to see what the pre- and post-conditions say in the calling context;
  2. logical deduction to prove the pre-condition;
  3. logical deduction to prove the next mid-condition (what you wanted to achieve by the procedure call).

9.9 Exercises

1. You have already seen the following problems for solution in Miranda:
   - `round`: round a real number to the nearest integer.
   - `solve`: solve the quadratic equation $ax^2 + bx + c = 0$.
   - `middle`: find the middle one of three numbers.
   - `newtonsqrt`: calculate a square root by Newton's method.

   Translate the Miranda solutions (specifications and definitions) directly into Modula-2.
2. The following standard procedures are defined in Niklaus Wirth's *Programming in Modula-2*: `ABS`, `CAP`, `CHR`, `FLOAT`, `ODD`, `TRUNC`, `DEC`, `INC`. Try to translate the explanations in the report into formal, logical specifications.
3. Implement the `middle` function (see Exercise 1) in Modula-2 using the `SWAP` procedure instead of recursion. Show that it works correctly.
4. Specify and define Modula-2 procedures `Order4` and `Order5` analogous to `Order3`, and using the same method, a straight-line sequence of calls of `Order2`. Prove that they work correctly. Can you show that you use the minimum number of calls of `Order2`? Is there a general argument that shows that this method works for ordering any given number of variables?

Chapter 10

Loops

An important difference between functional and imperative programming is the loop constructs (`WHILE`, `UNTIL` and `FOR`). They are essentially imperative (that is what `DO` means), and to perform analogous computations in Miranda you must use recursion. The techniques you need to reason about `WHILE` loops are really just a use of mid-conditions; but the mid-conditions involved are so important that they are given a special name of their own — they are *loop invariants*. Even in relatively unreasoned programming, experience shows that there is a particularly crucial point at the top of the loop where it is useful to put comments, and the method of loop invariants is a logical formalization of this idea.

10.1 The coffee tin game

This game illustrates reasoning with loop invariants. It uses a tin full of two kinds of coffee bean, Blue Mountain and Green Valley (Figure 10.1).

Rules:
WHILE at least two beans in tin DO
  Take out any two beans;
  IF they are the same colour
  THEN
    throw them both away;
    put a Blue Mountain bean back in (*you may need spare blue beans*)
  ELSE
    throw away the blue one;
    put the green one back
  END
END;

Figure 10.1 Coffee beans

QUESTION: if you knew the original numbers of blue and green beans, can you tell the colour of the final bean?

The contents of the tin at any given moment are described by the numbers of blue and green beans. Let us write the state as mB + nG for m blue beans, n green.

A transition (move) is determined by the colours of the two beans taken out: BB, BG or GG (Figure 10.2).

More generally, we have

BB: $m\text{B} + n\text{G} \rightarrow (m-1)B + n\text{G}$
BG: $m\text{B} + n\text{G} \rightarrow (m-1)B + n\text{G}$
GG: $m\text{B} + n\text{G} \rightarrow (m+1)B + (n-2)G$

The important thing to notice is the way the number of *green* beans can change. If it changes at all, it is decreased by 2, and this means that the *parity* of the number of greens — whether it is odd or even — does not change. The parity is *invariant*.

Suppose, then, there is originally an *odd* number of green beans. Then, however the game progresses (and there are lots of different possibilities), there will always be an odd number of greens. This holds true right up to the end, when there is only one bean left. So what colour is that? It must be green. Similarly, if there is originally an even number of green beans, then the final bean must be blue. So we have answered our question.

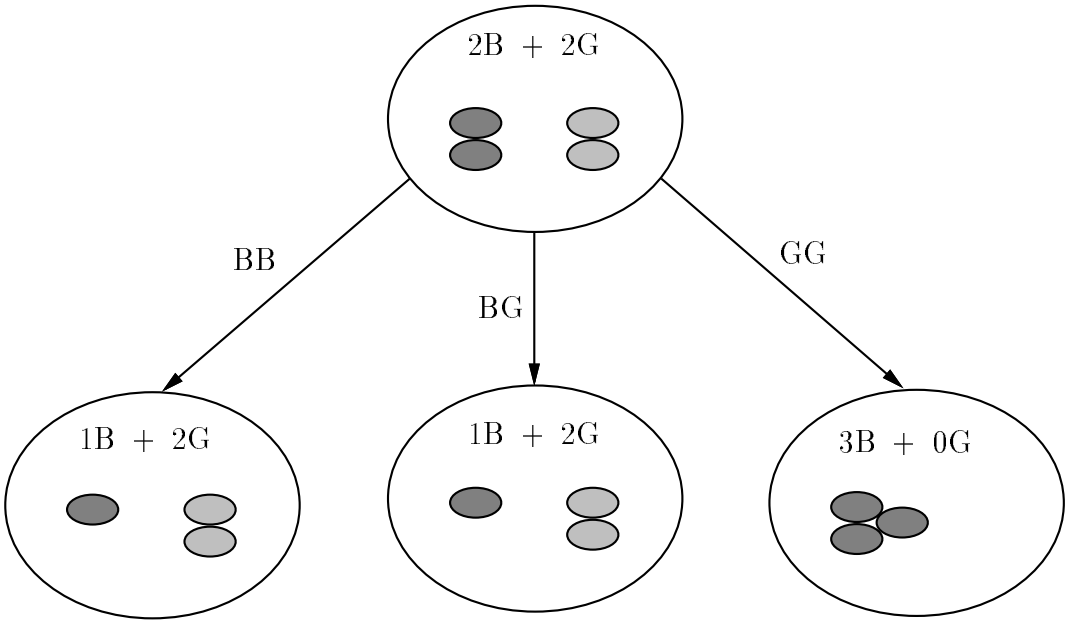

Figure 10.2 Transition

Notice how the invariant, the parity, does not in itself tell us much about the numbers of beans. It is only when we reach the end that the parity combines with that fact to give very precise information about the numbers.

Another small point. How do we know that we ever reach a state with only one bean? This is obvious, because the *total* number of beans always decreases by one at each move. This total number is called a *variant* because it varies and it works very like recursion variants.

Coffee tin game with comments

Here is a version with 'mid-conditions' written as comments. We talked before about an invariant *quantity*, the green parity (odd or even). However, what appears here is an invariant *assertion*, a logical formula, namely that the current parity is the same as the original one. Our reasoning said that if this assertion was true before the move, then it will be true afterwards as well; hence if it was true at the beginning of the game (which it was, by definition) then it will be true at the end as well.

This conversion of invariant quantity into invariant assertion might look cumbersome in this case, but it gives a very general way of formulating invariants. Henceforth, an invariant will always be a logical assertion.

The *variant*, on the other hand (the total number of beans, which we used to prove that the game would end), is always a number:

```
(*pre: green parity p is p_0 & no. of beans > 0
 *post:  ( p_0 = Even & one blue left)
 *       \/ ( p_0 = Odd & one green left)
 *loop invariant: green parity p_0 & no. of beans > 0
 *loop variant = total number of beans
 *)
 WHILE at least two beans in tin DO
      (* number of greens = n, say *)
   Take out any two beans;
   CASE two colours OF
     BB: replace by B      (* greens = n*)
   | GG: replace by B      (* greens = n-2*)
   | BG,GB: replace by G   (* greens = n*)
   END
   (* green parity = p_0 again, variant decreased *)
 END;
 (* green parity is still p_0 & just one bean left *)
```

10.2 Mid-conditions in loops

Now think of a real `WHILE` loop, `WHILE` *test* `DO` *body* `END`, and imagine putting mid-conditions in. There is one point in the program execution that is crucial, namely (each time round) immediately before the loop test is evaluated. What makes it special is that there are two ways of reaching this point — when control comes to the loop from higher up in the code, and when it loops back from the end of the body — so it ties different execution paths together.

A mid-condition here is called the *loop invariant*. You should write it explicitly in a comment before the loop:

```
(*loop invariant:  ...  *)
WHILE test DO
body
END
```

Because there are two ways of reaching the invariant's point, two things need to be proved to show that the invariant behaves:

1. that it holds the first time the loop is reached, in other words that the invariant is *established* initially;
2. that *if* it holds at the start of an iteration, and *if* the loop test succeeds (so that we continue looping and we know that invariant $\wedge$ test), then the execution of the body will ensure that the invariant still holds next time round, in other words that the body *reestablishes* the invariant.

Because the loop invariant point can be reached by two routes, it is — apart from the overall specification — far and away the most important place for mid-conditions. We suggest you take every opportunity to practise the method in your programming.

For the Coffee Tin, the invariant (green parity = p_0 and beans > 0) is established trivially — by definition of p_0. It is the reestablishment that is important, showing that whatever move is made (and whatever happens while the move is being made), the green parity is restored to p_0 and beans remain > 0.

At the end, the payoff is that we still know that the invariant holds, but we also know that the loop test fails (that is why we have finished looping). If the invariant is a good one, this combination will allow us to deduce the post-condition (maybe with some final computation). At the end of the Coffee Tin game, we have both that the green parity is still p_0 and that there is only one bean left. This combination is strong enough to tell us exactly what colour the bean is.

10.3 Termination

If we finish looping, then we know the combination 'invariant $\wedge\neg$loop test' holds. But not all loops do terminate. Some loop for ever, and we want to rule out this possibility. The Coffee Tin Game must terminate, because each move decreases by one the total number of beans left, but this can never go negative. Therefore after finitely many moves, the game must stop.

In general, to reason with `WHILE` loops we use not only the invariant, a logical condition as above, but also a *loop variant*. This works the same way as does a recursion variant. It is a natural number related to the computer variables such that the loop body must strictly decrease it, but it can never go negative. Then only finitely many iterations are possible, so the `WHILE` loop must eventually terminate.

For the Coffee Tin, the variant is the total number of beans left.

10.4 An example

Apparently, the method of invariants and variants as presented so far is a *reasoning tool:* given a `WHILE` loop, you might be able to find a loop invariant to prove that it works. But actually, the invariant can appear much earlier than that, even before you have written any code, as a clarification of how you think the implementation will work. Let us explore this in a simple problem to sum the elements of an array of reals: