# Chapter 20

# Procedure Declarations and Calls

The procedure call is an effective tool for managing the complexity of programming problems. A large problem can usually be subdivided into a number of smaller problems, which are then solved individually. If a team of programmers is working on a project, this provides a natural way of dividing the work among the individual programmers. If some of the subproblems are still somewhat large, further subdivision may be called for. This approach is called *top-down design*. Program derivation is especially useful in this method, as the specification of the original problem is used to guide the subdivision in such a way that correct solutions to the smaller problems provide a correct solution to the larger one.

The proof rules for procedure declarations and calls are most easily introduced in the context of a simple example. In the procedure $inc$ below, $x$ is an input (value) parameter, and $y$ is an output (variable) parameter. What is the result of a call $inc(s,t)$ to the procedure below if $s$ has a value of 7 before the call?

**procedure** $inc($ $x$ : **int** ; **var** $y$ : **int**);
$|[\ y := x + 1]|$

If procedure declarations and calls are viewed as a way of expanding the functionality of a programming language (in particular, GCL), rules are needed for annotating a procedure call with pre and post conditions. What would be a valid Hoare triple for the call $inc(s,t)$ to procedure $inc$ above?

A good first guess would be

$$\{\ true\ \}\ inc(s,t)\ \{\ t = s + 1\ \}$$

However, what would this suggest about the call $inc(s,s)$ ? Also, what would be the result of such a call if $s$ initially has value 4?

The result, of course, would be that parameter $s$ is increased to 5 by the procedure, but following the pattern of the first guess at an appropriate

Hoare triple would give

$$\{\ true\ \}\ inc(s, s)\ \{\ s = s + 1\ \}$$

This postcondition is equivalent to $FALSE$, since no state satisfies it.

This somewhat surprising result is caused by a phenomenon called *aliasing* because the two formal parameters $x$ and $y$ both correspond to the same actual parameter $s$. Aliasing may also occur in other ways, some of which greatly complicate the procedure proof rules. As an example, consider the procedure declaration below.

**procedure** $bump(\mathbf{var}\ x, y : \mathbf{int})$;
$|[\ x := x + 1\ ;\ y := y + 1\ ;\ z := z + 1]|$

Hand trace the sequence of commands below to observe and understand other types of aliasing.

$a, b, z := 5, 10, 15\ ;\ bump(a, b)$;
$a, z := 5, 10\ ;\ bump(a, a)$;
$a, z := 5, 10\ ;\ bump(a, z)$;
$z := 10\ ;\ bump(z, z)$;

In the first call there is no aliasing. In the second call, the aliasing arises from one actual parameter $a$ corresponding to the two formal variable parameters $x$ and $y$. In the third call, the aliasing arises from the actual parameter $z$ corresponding to formal variable parameter $y$ while also appearing as a nonlocal variable in the procedure body. How does aliasing occur in the fourth call?

In order to keep the proof rule simple, these kinds of aliasing will not be allowed. In particular, passing the same actual parameter to more than one formal *variable* parameters will not be permitted. Also, for the time being, nonlocal variables will not be permitted.

However, the type of aliasing exhibited in $inc(s, s)$ is *benign* in the sense that the output parameter gets the value that is expected. For example, if $s = 8$ is true before the call $inc(s, s)$, then $s = 9$ is true after the call.

The first part of the proof rule is a specification (pre and post conditions) for the procedure body in terms of the formal parameters. The second part consists of rules for substituting the names of actual parameters for their corresponding formal parameters in that specification.

For example, the first guess above corresponded to the following specification of the procedure:

**procedure** $inc(\ x : \mathbf{int}\ ;\ \mathbf{var}\ \ y : \mathbf{int})$;
$|[\ \{\ pre : true\ \}\ y := x + 1\ \{\ post : y = x + 1\ \}\ ]|$

A straightforward substitution of actuals for formals then leads to the erroneous triple:

$$\{\ true\ \}\ inc(s, s)\ \{\ s = s + 1\ \}$$

How may the specification be changed to accommodate the benign type of aliasing, that is, the type where an actual parameter $s$ corresponds to

both a value and a variable formal parameter? How would you convince someone that, after the procedure call $inc(s, s)$, variable $s$ has the value that it should have? After some thought, it may be seen that what is needed is a way to refer to the value of $s$ (as the input parameter) *before* the call. This may be accomplished by introducing a specification constant $C$ to record the value of $s$ before the call. The appropriate triple would then be

$$\{\ s = C\ \}\ inc(s, s)\ \{\ s = C + 1\ \} \quad . \tag{20.1}$$

This suggests a specification of the form

**procedure** $inc($ $x$ : **int** ; **var** $y$ : **int**);
$|[\ \{\ pre : x = C\ \}\ y := x + 1\ \{\ post : y = C + 1\ \}\ ]|$

The idea is to forbid references to the value parameters in the postcondition, as their pre-call values may be changed by benign aliasing. This is accomplished by introducing a specification constant for each value parameter, and asserting that each value parameter is equal to its respective constant in the precondition. Then the specification constant may be used in the postcondition.

With this provision, the call rule is based upon substituting the actual parameters for the formal parameters. Note that a substitution may also be necessary to rename the specification constants, as required by the context of the call. For example, the substitution $C := B$ gives the valid triple

$$\{\ s = B\ \}\ inc(s, t)\ \{\ t = B + 1\ \} \quad . \tag{20.2}$$

The substitution may contain specification constants, program variables, and literals, as in $C := B + z + 5$, which yields the triple

$$\{\ s = B + z + 5\ \}\ inc(s, t)\ \{\ t = B + z + 6\ \} \quad . \tag{20.3}$$

Such a substitution will be valid as long as the substituted expression (such as $B + z + 5$) does not refer to any actual parameter.

## 20.1 The proof rule for procedures

Consider declarations of the form

**procedure** $f(x$ ; **var** $y)$; $|[\ S]|$

where $S$ is the procedure body and $x, y$ are formal parameters. Parameters $x$ and $y$ should each be thought of as a list of parameters, but the discussion here will be simplified by treating them as single variables. Types will be omitted throughout this discussion.

In practice a procedure $f$ is verified with respect to a specification. The specification consists of a procedure heading (with the list of parameters) together with a pair $pre$, $post$ of predicates expressed in terms of the formal parameters:[1]

[1] For readability, pre and post conditions should both appear before lengthy procedure bodies.

**procedure** $f(x\,;\,\textbf{var}\ y)$;
$|[\ \{\,pre\,\}\ S\ \{\,post\,\}\ ]|$

For the correctness of the procedure, the proof obligation is

$$\{\,pre\,\}\ S\ \{\,post\,\} \quad . \tag{20.4}$$

Then for procedure calls $f(a,b)$ the basic idea is that we can replace the formal parameters of the specification with the actual parameters of the procedure call. Recall that actual parameters corresponding to value formal parameters may be expressions, but those corresponding to variable parameters must be assignable (able to appear on the left side of an assignment).

For the procedure declaration rule we will consider only *proper* procedure declarations, which are subject to the previously discussed restriction that no value parameter is mentioned in the postcondition.

*Proper Specification*
A procedure specification

$$\textbf{procedure}\ f(x\,;\,\textbf{var}\ y);|[\ \{\,pre\,\}\ S\ \{\,post\,\}\ ]|$$

is *proper* provided that $post$ is independent of $x$.

The procedure declaration rule is then simply

*Valid Declaration*
A procedure declaration

$$\textbf{procedure}\ f(x\,;\,\textbf{var}\ y);|[\ \{\,pre\,\}\ S\ \{\,post\,\}\ ]|$$

is *valid* provided that the specification is proper and a proof is supplied for $\{\,pre\,\}\ S\ \{\,post\,\}$ .

The call rule is recorded in the box below.

*Procedure Call Rule*

If $f$ has a valid declaration then

$$\{\,pre(x,y:=a,b)\sigma\,\}\ f(a,b)\ \{\,post(y:=b)\sigma\,\} \tag{20.5}$$

for all $a,b$ and all substitutions $\sigma$ for constants in $pre, post$, provided that $b$ contains no variable more than once (recall that $a$ and $b$ are lists of parameters), and substitution $\sigma$ does not refer to any parameter.

**Exercise 20.0**
Show that the following specification of $inc$ is not valid.

**procedure** $inc(\, x : \textbf{int} \,;\, \textbf{var}\ y : \textbf{int})$;
$|[\ \{\, pre : true \,\}\ y := x + 1\ \{\, post : y = x + 1 \,\}\ ]|$

**Exercise 20.1**
Show that the following declaration of $inc$ is valid.

**procedure** $inc(\, x : \textbf{int} \,;\, \textbf{var}\ y : \textbf{int})$;
$|[\ \{\, pre : x = C \,\}\ y := x + 1\ \{\, post : y = C + 1 \,\}\ ]|$

As another example, the valid specification

**procedure** $swap(\textbf{var}\ x, y)$;
$|[\ \{\, x = X \ \wedge\ y = Y \,\}\ x, y := y, x\ \{\, x = Y \ \wedge\ y = X \,\}\ ]|$

gives (using $(X, Y := A, B)$ for $\sigma$)

$$\{\, a = A \ \wedge\ Y = B \,\}\ swap(a, b)\ \{\, a = B \ \wedge\ b = A \,\} \quad . \tag{20.6}$$

Also (using $(X, Y := 7 * A, T + 1)$ for $\sigma$)

$$\{\, a = 7 * A \ \wedge\ b = T + 1 \,\}\ swap(a, b)\ \{\, a = T + 1 \ \wedge\ b = 7 * A \,\} \quad . \tag{20.7}$$

The rule of independence discussed in Chapter 7 is frequently used in the context of a procedure call. What should it mean for a procedure call $f(a, b)$ to be independent of a predicate $R$? In Chapter 7, a command $S$ is independent of a predicate $R$ if $S$ does not assign to any of the variables of $R$. For a procedure call, the variable parameters correspond to variables receiving assignments, so $f(a, b)$ is independent of $R$ if $R$ does not refer to any variable actual parameter of the call $f(a, b)$. This rule is recorded in the box below.

*Rule of independence*
If $R$ is independent of $b$ (recall that $b$ is the list of variable parameters) and

$$\{\, P \,\}\ f(a, b)\ \{\, Q \,\}$$

then

$$\{\, P \ \wedge\ R \,\}\ f(a, b)\ \{\, Q \ \wedge\ R \,\} \tag{20.8}$$

For example, from (**??**) the rule of independence (with $R$ being $s = B$) gives

$$\{\, s = B \,\}\ inc(s, t)\ \{\, t = B + 1 \ \wedge\ s = B \,\} \quad .$$

but not the erroneous

$$\{\, s = B \,\}\ inc(s, s)\ \{\, s = B + 1 \ \wedge\ s = B \,\} \quad .$$

since $s = B$ is not independent of the actual variable parameter $s$

Now, any of the programs derived earlier in the text may be converted into a procedure just by determining the parameters and local variables, and introducing specification constants for the value parameters. For example, the following program for selection sort was derived in Chapter 16. All intermediate assertions have been removed.

```
|[ con N : int ; var h, k, m : int ; a : array[0..N) of int;
    {pre : N ≥ 0 ∧ a = A}
  h := N;
  do h ≠ 0 →
    k, m := 0, 1;
    do m ≠ h →
      if a.m > a.k → k := m
      [] a.m ≤ a.k → skip
      fi
      m := m + 1;
    od
    h, a := h − 1, a(h − 1, k : a.k, a.(h − 1))
  od
    {post : S.a.0.N ∧ (bag.a.0.N = bag.A.0.N)}
]|
```

The goal of this algorithm is to sort array $a$, so $a$ will be a variable parameter. The other variables $h$, $k$, and $m$ will be local variables. Constant $N$ will be global. The procedure is given below; note that there is no need for specification constants since there are no value parameters.

```
procedure SelSort(var a : array[0..N) of int);
|[ var h, k, m : int;
    {pre : N ≥ 0 ∧ a = A}
  h := N;
  do h ≠ 0 →
    k, m := 0, 1;
    do m ≠ h →
      if a.m > a.k → k := m
      [] a.m ≤ a.k → skip
      fi
      m := m + 1;
    od
    h, a := h − 1, a(h − 1, k : a.k, a.(h − 1))
  od
    {post : S.a.0.N ∧ (bag.a.0.N = bag.A.0.N)}
]|
```

In order to illustrate the use of specification constants, suppose that some subrange $a[q..r)$ of the entire array is to be sorted. This would require $q$ and $r$ to be passed as value parameters. Then the specification would appear as below, with essentially the same proof of the procedure body. The specification constants $Q$ and $R$ are now necessary to represent the precondition values of value parameters $q$ and $r$, respectively.

```
procedure SelSort(var a : array[0..N) of int ; q, r : int);
|[ var h, k, m : int;
   {pre : 0 ≤ q ≤ r ≤ N ∧ a = A ∧ q = Q ∧ r = R}
 h := r;
 do h ≠ q →
   k, m := q, q + 1;
   do m ≠ h →
     if a.m > a.k → k := m
     [] a.m ≤ a.k → skip
     fi
     m := m + 1;
   od
   h, a := h − 1, a(h − 1, k : a.k, a.(h − 1))
 od
   {post : Sort.a.Q.R ∧ bag.a.Q.R = bag.A.Q.R}
]|
```

## 20.2 Exercises

**Exercise 20.2**
Give a proper declaration of a procedure *max* , which calculates the maximum of two input integers, and puts the result in an output integer.

**Exercise 20.3**
Verify the triple in (**??**), using the second definition of *inc* .

**Exercise 20.4**
Verify the triple in (**??**), using the second definition of *inc* .

**Exercise 20.5**
Verify the triple in (**??**), using the second definition of *inc* .

**Exercise 20.6**
Prove that the specification of procedure *swap* is valid.

**Exercise 20.7**
Verify the triple in (**??**), using the definition of *swap* given in the text.

**Exercise 20.8**
Verify the triple in (**??**) above, using the definition of *swap* given in the text.

**Exercise 20.9**
Write the Linear Search algorithm of Chapter 16 as a procedure and provide a valid procedure declaration.

**Exercise 20.10**
Write the Bounded Linear Search algorithm of Chapter 16 as a procedure and provide a valid procedure declaration.

**Exercise 20.11**

Write the Binary Search algorithm of Chapter 16 as a procedure and provide a valid procedure declaration.

# Chapter 23

# Recursive Procedures

So far all the procedure derivations derived in this text have been nonrecursive ones. In fact, the proof rule given in Chapter 20 for a procedure declaration is not adequate for a recursive procedure. The difficulty arises in giving a proof of the body of such a procedure. In particular, at the point of a recursive call to the procedure, it seems that one must know the procedure is correct in order to prove that it is correct. As before, a simple example can help the reader to discover an appropriate proof rule. Recall from Chapter 17 the integer factorial function, which has a natural recursive definition.

$$X! = 1 \quad \text{if } X = 0$$
$$X! = X * (X-1)! \quad \text{if } X > 0$$

This definition leads to a recursive procedure definition for factorial.

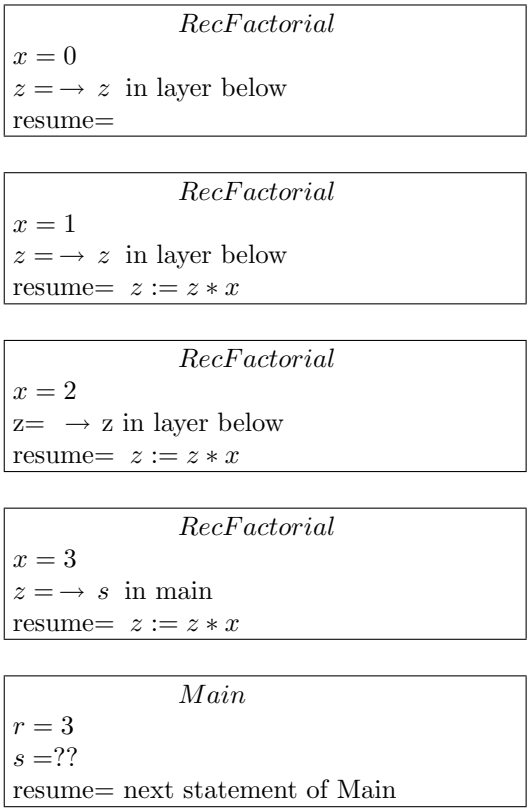
**procedure** $RecFactorial(x : \textbf{int}\,;\, \textbf{var}\ z : \textbf{int})$;
|[ $\{\, pre : \ 0 \leq x = X \,\}$
**if** $x = 0 \ \rightarrow \ z := 1$
[] $x > 0 \ \rightarrow$
$RecFactorial(x-1, z)$;
$z := z * x$
**fi**
$\{\, post : z = X! \,\}$
]|

However, it is not obvious how to prove the correctness of this procedure. Using the runtime stack model of Chapter 17, hand run a procedure call of the form below.

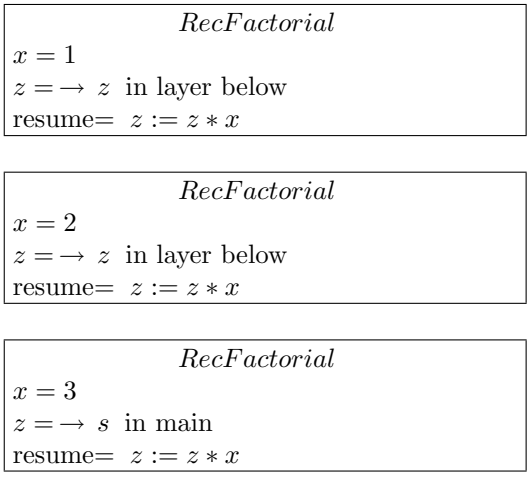
$$r := 3\,;\, RecFactorial(r, s)$$

Why does $s$ receive a correct value of 6 for this call? Why does the procedure call terminate? That is, why does the procedure not continue to call itself indefinitely? What happens the last time the procedure calls itself?

Right after this last call the runtime stack would look something like this:




Now as the procedure body is run at the top layer, since the value of $x$ is 0, $z$ is assigned 1, and the topmost activation terminates. The $z$ at the top layer refers indirectly to the $z$ at the layer below, and that one refers to the one below it, and so on until the $z$ at layer two refers indirectly to $s$ in the main. Therefore, the assignment $z := 1$ at the top layer indirectly assigns 1 to $s$ in the main. This leaves the stack looking as below.



| *Main* |
|---|
| $r = 3$ |
| $s = 1$ |
| resume= next statement of Main |

Resuming the run of this activation at $z := z * x$ indirectly assigns a value of 1*1 to $s$ in the main, and then terminates, leaving the stack looking as below.

| *RecFactorial* |
|---|
| $x = 2$ |
| $z = \rightarrow z$ in layer below |
| resume= $z := z * x$ |

| *RecFactorial* |
|---|
| $x = 3$ |
| $z = \rightarrow s$ in main |
| resume= $z := z * x$ |

| *RecFactorial* |
|---|
| $r = 3$ |
| $s = 1, 1$ |
| resume= next statement of Main |

Continuing in this fashion results in $s$ getting a value of 2 for the next layer, and finally a value of 6, as required.

The notable feature of this process is that each time a recursive call is made, the problem to be solved is simpler than the initial problem. In this example, the measure of difficulty, or "size", of the problem is the value of parameter $x$. The idea is that in proving the procedure correct on a call of arbitrary size $K$, it may be assumed that every procedure call of size smaller than $K$ is correct. So in proving the *RecFactorial* procedure correct, add conjunct $x = K$ to the precondition. Then $x - 1 < K$ is a precondition to the recursive call $RecFactorial(x - 1, z)$, so the call may be assumed to establish $z = (X - 1)!$. The annotated procedure would then appear as follows.

**procedure** $RecFactorial(x :$ **int** ; **var** $z :$ **int**);
|[ $\{ pre : 0 \leq x = X \}$
**if** $x = 0 \rightarrow \{ pre \wedge X = 0 \}\ z := 1\ \{ z = X! \}$
[] $x > 0 \rightarrow$
$\{ 0 \leq x - 1 = X - 1 \wedge x - 1 < K \}$
$RecFactorial(x - 1, z)$;
$\{ X > 0 \wedge z = (X - 1)! \}$
$z := x * z$
$\{ X > 0 \wedge z = X * (X - 1)! \}$
**fi**
$\{ post : z = X! \}$
]|

## 23.1 The proof rule for recursive procedures

The factorial example illustrates that the proof rule for procedures is inadequate for proving correctness of recursive procedures. What is needed to prove correctness of a recursive procedure body is an assumption that "smaller" recursive calls within the body meet the specification. That is, a measure of the "size" of a call will need to be defined, and any recursive calls in the body of the procedure having a reduced "size" may be assumed to meet the specification. This motivates the following rule for recursive procedures.

*Recursion rule*
If $f$ has a proper declaration then to show its correctness, i.e., $\{\ pre\ \}\ S\ \{\ post\ \}$ , it is sufficient to give a variant function $v$ (of the *value* and *var* parameters and the constants) and to prove

$$\{\ v = K\ \wedge\ pre\ \}\ S\ \{\ post\ \}$$

(where $K$ is a fresh constant). In that proof, one can assume this "*recursive call rule*":

$$\{\ (v < K\ \wedge\ pre)(x, y := a, b)\sigma\ \}\ f(a, b)\ \{\ post(y := b)\sigma\ \} \qquad (23.1)$$

for any $a, b$ and substitutions $\sigma$ for constants in $pre, post$ , provided that no variable appears more than once in $b$ , and $b$ is disjoint from the domain and range of $\sigma$ .

As an example, consider the fibonacci function $fib$ defined for natural numbers $X$ by

$$fib.X = X \quad \text{if } X = 0\ \vee\ X = 1$$
$$fib.X = fib.(X-1) + fib.(X-2) \quad \text{if } X > 1$$

We want a procedure $fib$ satisfying the following specification.

**procedure** $fib(x :$ **int** ; **var** $z :$ **int**);
|[ **var** $w :$ **int**;
$\{\ pre :\ 0 \le x = X\ \}$
$\{\ variant : x\ \}$
$\{\ post :\ z = fib.X\ \}$
]|

The annotated procedure body below satisfies the specification

$\{\, pre \ \wedge\ x = K \,\}$
**if** $x = 0 \ \vee\ x = 1 \ \rightarrow\ \{\, X = 0 \ \vee\ X = 1 \,\}\ z := 1\ \{\, z = fib.X \,\}$
$[\!]\ \ x > 1 \ \rightarrow$
$\quad \{\, 0 \leq X - 1 < K \ \wedge\ 0 \leq X - 2 < K \,\}$
$\quad fib(x - 1, w);$
$\quad \{\, 0 \leq X - 2 < K \ \wedge\ w = fib.(X - 1) \,\}$
$\quad fib(x - 2, z);$
$\quad \{\, 0 \leq X - 2 \ \wedge\ w = fib.(X - 1) \ \wedge\ z = fib.(X - 2) \,\}$
$\quad z := w + z;$
$\quad \{\, post:\ z = fib.X \,\}$
**fi**

Here the recursive call rule is used to establish the two triples

$$\{\, X - 1 < K \ \wedge\ 0 \leq x - 1 = X - 1 \,\}\ fib(x - 1, w)\ \{\, w = fib.(X - 1) \,\}$$
$$\{\, X - 2 < K \ \wedge\ 0 \leq x - 1 = X - 1 \,\}\ fib(x - 2, z)\ \{\, z = fib.(X - 2) \,\}$$

and the independence rule is used to carry along established conjuncts.

## 23.2 A recursive procedure for binary search

It should come as no surprise that the proof rule for recursive procedures may be used to guide the derivation of recursive procedures. The key idea is that in deriving the body of a procedure, a specification may arise for some part of the body which closely resembles the specification for the procedure itself. If parameter substitutions can be found which make the inner specification agree with the overall specification, then a recursive call with these substitutions is appropriate. The only difference between this and the derivation of an arbitrary procedure call is that a variant function must be defined for the recursive procedure, and the derived recursive call must have a smaller variant than that of the procedure. As usual, an example will help to clarify these ideas.

In the binary search algorithm, a divide-and-conquer strategy is used to efficiently search an ordered array for a given element. The specification is

$|[\ \mathbf{con}\ N :\ \mathbf{int}\{N \geq 0\}\ ;\ a : array[0..N) of\ \mathbf{int};$
$\quad \{\, Q : sorted.a[0..N) \ \wedge\ (a.0 \leq x < a.(N - 1)) \,\}$
$\quad BinSer$
$\quad \{\, R : (0 \leq y < N - 1) \ \wedge\ (a.y \leq x < a.(y + 1)) \,\}$
$]|$

A divide and conquer strategy suggests splitting the range into two (probably about equal) parts. To that end, let $m$ be any integer such that $0 < m < N - 1$. Note that if there is no such integer then the postcondition can be satisfied by the assignment $y := 0$. If there is such an integer $m$, the postcondition may be transformed as follows:

$$
\begin{array}{ll}
 & (0 \le y < N-1) \ \wedge \ (a.y \le x < a.(y+1)) \\
\equiv & \quad \{\ 0 < m < N-1\ \} \\
 & ((0 \le y < m) \ \vee \ (m \le y < N-1)) \ \wedge \ (a.y \le x < a.(y+1)) \\
\equiv & \quad \{\ \text{distributivity}\ \} \\
 & ((0 \le y < m) \ \wedge \ (a.y \le x < a.(y+1))) \vee \\
 & \quad ((m \le y < N-1) \ \wedge \ (a.y \le x < a.(y+1))) \\
\equiv & \quad \{\ (y < m \ \Rightarrow \ (y+1) \le m) \wedge \ \text{sorted.a}\ \} \\
 & ((0 \le y < m) \ \wedge \ (a.y \le x < a.(y+1)) \ \wedge \ (x < a.m)) \vee \\
 & \quad ((m \le y < N-1) \ \wedge \ (a.y \le x < a.(y+1)) \ \wedge \ (a.m \le x))
\end{array}
$$

This disjunction suggests the investigation of a selection

$$
\begin{array}{l}
\quad \{\ Q : sorted.a[o..N) \ \wedge \ (a.0 \le x < a.(N-1))\ \} \\
\textbf{if}\ x < a.m \ \rightarrow \\
\qquad \{\ sorted.a[o..m] \ \wedge \ (a.0 \le x < a.m)\ \} \\
\quad S0 \\
\qquad \{\ (0 \le y < m) \ \wedge \ (a.y \le x < a.(y+1))\ \} \\
[\!] \ a.m \le x \ \rightarrow \\
\qquad \{\ sorted.a[m..N-1] \ \wedge \ (a.m \le x < a.(N-1))\ \} \\
\quad S1 \\
\qquad \{\ (m \le y < N-1) \ \wedge \ (a.y \le x < a.(y+1))\ \} \\
\textbf{fi} \\
\quad \{\ R : (0 \le y < N) \ \wedge \ (a.y \le x < a.(y+1))\ \}
\end{array}
$$

For both $S0$ and $S1$, the specification has the same form as the original specification, which suggests the definition of a recursive procedure with the following specification. In order to make the specification of the procedure proper, we introduce spec constants $A$, $B$, $T$, $X$, and for convenience of notation we define two predicates:

$$
\begin{array}{l}
Q.a.b.t.x : sorted.a[b..t] \ \wedge \ (a.b \le x < a.t) \\
R.a.b.t.x.y : (b \le y < t) \ \wedge \ (a.y \le x < a.(y+1))
\end{array}
$$

$$
\begin{array}{l}
\textbf{procedure}\ BinSer(\textbf{in a}, \textbf{b}, \textbf{t}, \textbf{x}; \textbf{out y}); \\
\quad |[\ a : array[0..N) of\ \textbf{int}\ ; b, t, x : \ \textbf{int}; \\
\qquad \{\ Q.a.b.t.x \ \wedge \ a = A \ \wedge \ b = B \ \wedge \ t = T \ \wedge \ x = X\ \} \\
\qquad S \\
\qquad \{\ R.A.B.T.X.y\ \} \\
\quad ]|
\end{array}
$$

The procedure body is developed as above. If there is no integer $m$ such that $b < m < t$, then the postcondition is satisfied by the assignment $y := b$. Otherwise, the calculations above lead us to a selection, giving the recursive procedure below.

**procedure** $BinSer(a : \textbf{array}[0..N)\ \textbf{of}\ \textbf{int}\ ; b, t, x : \textbf{int}\ ; y : \textbf{int})$;
|[ $\{\ Q.a.b.t.x \ \wedge\ a = A \ \wedge\ b = B \ \wedge\ t = T \ \wedge\ x = X\ \}$
**if** $b + 1 = t \ \rightarrow\ y := b$;
[] $b + 1 \ \neq t \ \rightarrow$
$m := (b + t) \div 2$;
**if** $x < a.m \ \rightarrow$
$\{\ Q.a.b.m.x \ \wedge\ a = A \ \wedge\ b = B \ \wedge\ m = M \ \wedge\ x = X\ \}$
$BinSer(a, b, m, x, y)$
$\{\ R.A.B.M.X.y\ \}$
[] $a.m \leq x \ \rightarrow$
$\{\ Q.a.m.t.x \ \wedge\ a = A \ \wedge\ m = M \ \wedge\ t = T \ \wedge\ x = X\ \}$
$BinSer(a, m, t, x, y)$
$\{\ R.A.M.T.X.y\ \}$
**fi**
**fi**
$\{\ R.A.B.T.X.y\ \}$
]|

The variant function for this procedure is $v = t - b$. The annotations below will guide the reader to a straightforward proof of correctness for the recursive procedure, using the recursive procedure rule.

**procedure** $BinSer(a : \textbf{array}[0..N)\ \textbf{of}\ \textbf{int}\ ; b, t, x : \textbf{int}\ ; y : \textbf{int})$;
|[ $\{\ Q.a.b.t.x \ \wedge\ a = A \ \wedge\ b = B \ \wedge\ t = T \ \wedge\ x = X \ \wedge\ t - b = K\ \}$
**if** $b + 1 = t \ \rightarrow\ y := b$;
[] $b + 1 \ \neq t \ \rightarrow$
$m := (b + t) \div 2$;
**if** $x < a.m \ \rightarrow$
$\{\ (m - b < K) \ \wedge\ Q.a.b.m.x \ \wedge\ a = A \ \wedge\ b = B \ \wedge\ m = M \ \wedge\ x = X\ \}$
$BinSer(a, b, m, x, y)$
$\{\ R.A.B.M.X.y\ \}$
[] $a.m \leq x \ \rightarrow$
$\{\ t - m < K)\ \wedge\ Q.a.m.t.x \ \wedge\ a = A \ \wedge\ m = M \ \wedge\ t = T \ \wedge\ x = X\ \}$
$BinSer(a, m, t, x, y)$
$\{\ R.A.M.T.X.y\ \}$
**fi**
**fi**
$\{\ R.A.B.T.X.y\ \}$
]|

**Exercise 23.0**

For the program in the text above, write a corresponding Oberon program, and test it on a real machine. You will need to add some code which initializes the variables, and some code to output the results.

## 23.3 A recursive procedure for sift

Procedure $Sift$ from Chapter 22 provides another example of deriving a recursive procedure. The pertinent abbreviations for the algorithm are repeated below.

$$
\begin{array}{l}
j \ \rightarrow \ k \equiv (k = j) \ \vee \ (2 * j \ \rightarrow \ k) \ \vee \ (2 * j + 1 \ \rightarrow \ k) \\
f.a.j.n \equiv (\forall k : (j \ \rightarrow \ k) \ \wedge \ (k \leq n) : a.j \geq a.k) \\
H.a.m.n \equiv (\forall j : m < j : f.a.j.n)
\end{array}
$$

$$
\begin{array}{l}
P5 : p = 2 * m \\
P6 : q = (2 * m + 1) \min n \\
P8 : p > n \ \vee \ (z = p \ \wedge \ a.p \geq a.q) \ \vee \ (z = q \ \wedge \ a.p \leq a.q)
\end{array}
$$

Using these abbreviations, the specification of $Sift$ is:

**procedure** $Sift$(**var** $a$ : **array**$[0..N]$ **of** **int** ; $m, n$ : **int**);
|[ **var** $p, q, z$ : **int**;
$\{pre : H.a.m.n \ \wedge \ a = A\}$
$\{post : H.a.(m-1).n\}$
]|

Splitting off a term gives some insight into the recursive approach.

$$
\begin{array}{ll}
 & H.a.(m-1).n \\
\equiv & \quad \{ \text{ splitting off a term } \} \\
 & H.a.m.n \ \wedge \ f.a.m.n \\
\equiv & \quad \{ \text{ and-ident, distributivity } \} \\
 & H.a.m.n \ \wedge \ f.a.m.n \ \wedge \ \neg f.A.m.n \ \vee \\
 & H.a.m.n \ \wedge \ f.a.m.n \ \wedge \ f.A.m.n
\end{array}
$$

This disjunction together with the initializations of the variables $p$ , $q$ , and $z$ leads to a partial procedure body of the form

**procedure** $Sift($**var** $a :$ **array**$[0..N)$ **of int** $; m, n :$ **int**$);$
$|[$ **var** $p, q, z :$ **int**;
$\{pre : H.a.m.n \wedge a = A\}$
$p := 2 * m; \{P5\}$
$q := (2 * m + 1) \min n; \{P6\}$
**if** $p > n \rightarrow$ **skip**$\{H.a.m.n \wedge f.a.m.n\}$
$[]$ $p \leq n \rightarrow$
**if** $a.p \geq a.q \rightarrow z := p$
$[]$ $a.q \geq a.p \rightarrow z := q$
**fi**
$\{P8\}$
**if** $a.m \geq a.z \rightarrow$ **skip**
$\{H.a.m.n \wedge f.a.m.n \wedge f.A.m.n\}$
$[]$ $a.m < a.z \rightarrow$
$\{H.A.m.n \wedge \neg f.A.m.n \wedge a.m < a.z\}$
$a := b$
$\{H.a.m.n \wedge f.a.m.n \wedge \neg f.A.m.n\}$
**fi**
**fi**
$]|$

TBD: exercise for case $a.m \geq a.z$

$$
\begin{array}{ll}
 & f.a.m.n \wedge H.a.m.n \\
\equiv & \quad \{ \text{ definition of } H.a.m.n \ \} \\
 & f.a.m.n \wedge (\forall j : m < j : f.a.j.n) \\
\equiv & \quad \{ \text{ split the range } \} \\
 & f.a.m.n \wedge (\forall j : m < j < z : f.a.j.n) \wedge \\
 & (\forall j : z - 1 < j : f.a.j.n) \\
\equiv & \quad \{ \text{ definition of } H.a.(z-1).n \ \} \\
 & f.a.m.n \wedge (\forall j : m < j < z : f.a.j.n) \wedge H.a.(z-1).n
\end{array}
$$

The similarity of the third conjunct of this to the postcondition of $Sift$ suggests that recursive call $Sift(a, z, n)$ will establish the third conjunct. This gives a composition of the form below.

$$
\begin{array}{l}
\quad \{H.A.m.n \wedge \neg f.A.m.n \wedge a.m < a.z\} \\
a := b; \\
\quad \{f.a.m.n \wedge (\forall j : m < j < z : f.a.j.n) \wedge H.a.z.n\} \\
Sift(a, z, n) \\
\quad \{f.a.m.n \wedge (\forall j : m < j < z : f.a.j.n) \wedge H.a.(z-1).n\}
\end{array}
$$

$$
\begin{array}{ll}
 & \{ \text{ assume } A.m < A.z \ \} \\
 & wp.(a := b).(f.a.m.n \ \wedge \ (\forall\, j : m < j < z : f.a.j.n) \ \wedge \ H.a.z.n) \\
\equiv & \quad \{ \text{ definition of } H \ \} \\
 & wp.(a := b).(f.a.m.n \ \wedge \ (\forall\, j : m < j \ \wedge \ j \neq z : f.a.j.n)) \\
\equiv & \quad \{ \text{ assignment axiom } \} \\
 & f.b.m.n \ \wedge \ (\forall\, j : m < j \ \wedge \ j \neq z : f.b.j.n) \\
\equiv & \quad \{ \ P8,\ A.m < A.z \ \} \\
 & f.b.m.n \ \wedge \ A.m < A.z \ \wedge \ (\forall\, j : m < j \ \wedge \ j \neq z : f.b.j.n) \\
\Leftarrow & \quad \{ \ P6 - P8, \text{ definition of } f,\ [f.b.m.n \ \Rightarrow \ b.z \leq b.m] \ \} \\
 & b = a(m, z : a.z, a.m)
\end{array}
$$

TBD: clarify follows from above.

This calculation completes the derivation, resulting in the procedure body below.

$\{pre : H.a.m.n \ \wedge \ a = A \ \wedge \ n - m = K\}$
$p := 2 * m; \{P5\}$
$q := (2 * m + 1) \min n; \{P6\}$
**if** $p > n \ \rightarrow$ **skip**$\{H.a.m.n \ \wedge \ f.a.m.n\}$
$[\!]$ $p \leq n \ \rightarrow$
  **if** $a.p \geq a.q \ \rightarrow \ z := p$
  $[\!]$ $a.q \geq a.p \ \rightarrow \ z := q$
  **fi**
    $\{P8\}$
  **if** $a.m \geq a.z \ \rightarrow$ **skip**
    $\{H.a.m.n \ \wedge \ f.a.m.n \ \wedge \ f.A.m.n\}$
  $[\!]$ $a.m < a.z \ \rightarrow$
      $\{H.A.m.n \ \wedge \ \neg f.A.m.n \ \wedge \ a.m < a.z\}$
    $a := b;$
      $\{f.a.m.n \ \wedge \ (\forall\, j : m < j < z : f.a.j.n) \ \wedge \ H.a.z.n \ \wedge \ n - z < K\}$
    $Sift(a, z, n)$
      $\{f.a.m.n \ \wedge \ (\forall\, j : m < j < z : f.a.j.n) \ \wedge \ H.a.(z-1).n\}$
      $\{H.a.m.n \ \wedge \ f.a.m.n \ \wedge \ \neg f.A.m.n\}$
  **fi**
**fi**
  $\{post : H.a.(m-1).n\}$
$]\!]$

Note that $Sift(a, z, n)$ does not alter condition $f.a.m.n$ by the way that $f$ is defined, and similarly, does not alter condition $f.a.j.n$ for any index $j$ with $m < j < z$. The reason $f.a.m.n$ is not altered is that the call $Sift(a, z, n)$ swaps a parent only with one of its children, therefore it preserves the bag of descendants of $a.m$ (the parent of $a.z$). For index $j$ with $m < j < z$, no descendant of $a.j$ is moved.

## 23.4 Exercises