

Since \mathbf{ys} is reduced by one element at each call, we eventually arrive at the final state $S_{\text{final}} = (i, \text{nil})$, and the function returns i . Since $\text{length}(\text{nil}) = 0$, from $P(S_{\text{final}})$ it follows that $i = \text{length}(\mathbf{xs})$.

The difficult step in this proof is to choose the property P . It has to satisfy two constraints. First, it has to combine the arguments of the iterative computation such that the result does not change as the computation progresses. Second, it has to be strong enough that the correctness follows from $P(S_{\text{final}})$. A rule of thumb for finding a good P is to execute the program by hand in a few small cases, and from them to picture what the general intermediate case is.

Constructing programs by following the type

The above examples of list functions all have a curious property. They all have a list argument, $\langle \text{List } T \rangle$, which is defined as:

$$\begin{aligned} \langle \text{List } T \rangle &::= \text{nil} \\ &\quad | \quad T \text{ ' } | \text{ ' } \langle \text{List } T \rangle \end{aligned}$$

and they all use a **case** statement which has the form:

```
case Xs
of nil then <expr>   % Base case
[] X|Xr then <expr> % Recursive call
end
```

What is going on here? The recursive structure of the list functions exactly follows the recursive structure of the type definition. We find that this property is almost always true of list functions.

We can use this property to help us write list functions. This can be a tremendous help when type definitions become complicated. For example, let us write a function that counts the elements of a nested list. A *nested list* is a list in which each element can itself be a list, e.g., `[[1 2] 4 nil [[5] 10]]`. We define the type $\langle \text{NestedList } T \rangle$ as follows:

$$\begin{aligned} \langle \text{NestedList } T \rangle &::= \text{nil} \\ &\quad | \quad \langle \text{NestedList } T \rangle \text{ ' } | \text{ ' } \langle \text{NestedList } T \rangle \\ &\quad | \quad T \text{ ' } | \text{ ' } \langle \text{NestedList } T \rangle \end{aligned}$$

To avoid ambiguity, we have to add a condition on T , namely that T is neither `nil` nor a cons. Now let us write the function $\{\text{LengthL } \langle \text{NestedList } T \rangle\} : \langle \text{Int} \rangle$ which counts the number of elements in a nested list. Following the type definition gives this skeleton:

```
fun {LengthL Xs}
  case Xs
  of nil then <expr>
[] X|Xr andthen {IsList X} then
  <expr> % Recursive calls for X and Xr
[] X|Xr then
```

```

    <expr> % Recursive call for Xr
  end
end

```

(The third case does not have to mention {Not {IsList X}} since it follows from the negation of the second case.) Here {IsList X} is a function that checks whether X is nil or a cons:

```

fun {IsCons X} case X of _|_ then true else false end end
fun {IsList X} X==nil orelse {IsCons X} end

```

Fleshing out the skeleton gives the following function:

```

fun {LengthL Xs}
  case Xs
  of nil then 0
  [] X|Xr andthen {IsList X} then
    {LengthL X}+{LengthL Xr}
  [] X|Xr then
    1+{LengthL Xr}
  end
end

```

Here are two example calls:

```

X=[[1 2] 4 nil [[5] 10]]
{Browse {LengthL X}}
{Browse {LengthL [X X]}}

```

What do these calls display?

Using a different type definition for nested lists gives a different length function. For example, let us define the type $\langle \text{NestedList2 } T \rangle$ as follows:

$$\begin{array}{lcl}
 \langle \text{NestedList2 } T \rangle & ::= & \text{nil} \\
 & | & \langle \text{NestedList2 } T \rangle \text{ ' } | \text{ ' } \langle \text{NestedList2 } T \rangle \\
 & | & T
 \end{array}$$

Again, we have to add the condition that T is neither nil nor a cons. Note the subtle difference between $\langle \text{NestedList } T \rangle$ and $\langle \text{NestedList2 } T \rangle$! Following the definition of $\langle \text{NestedList2 } T \rangle$ gives a different and simpler function LengthL2:

```

fun {LengthL2 Xs}
  case Xs
  of nil then 0
  [] X|Xr then
    {LengthL2 X}+{LengthL2 Xr}
  else 1 end
end

```

What is the difference between LengthL and LengthL2? We can deduce it by comparing the types $\langle \text{NestedList } T \rangle$ and $\langle \text{NestedList2 } T \rangle$. A $\langle \text{NestedList } T \rangle$ *always* has to be a list, whereas a $\langle \text{NestedList2 } T \rangle$ can also be of type T. Therefore the

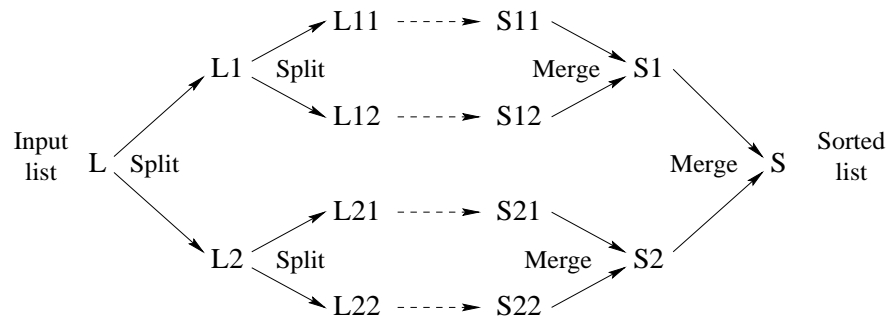


Figure 3.9: Sorting with mergesort

call `{LengthL2 foo}` is legal (it returns 1), whereas `{LengthL foo}` is illegal (it raises an exception). It is reasonable to consider this as an error in `LengthL2`.

There is an important lesson to be learned here. It is important to define a recursive type before writing the recursive function that uses it. Otherwise it is easy to be misled by an apparently simple function that is incorrect. This is true even in functional languages that do type inference, such as Standard ML and Haskell. Type inference can verify that a recursive type is *used* correctly, but the *design* of a recursive type remains the programmer's responsibility.

Sorting with mergesort

We define a function that takes a list of numbers or atoms and returns a new list sorted in ascending order. It uses the comparison operator `<`, so all elements have to be of the same type (all integers, all floats, or all atoms). We use the mergesort algorithm, which is efficient and can be programmed easily in a declarative model. The mergesort algorithm is based on a simple strategy called *divide-and-conquer*:

- Split the list into two smaller lists of approximately equal length.
- Use mergesort recursively to sort the two smaller lists.
- Merge the two sorted lists together to get the final result.

Figure 3.9 shows the recursive structure. Mergesort is efficient because the split and merge operations are both linear-time iterative computations. We first define the merge and split operations and then mergesort itself:

```

fun {Merge Xs Ys}
  case Xs # Ys
  of nil # Ys then Ys
  [] Xs # nil then Xs
  [] (X|Xr) # (Y|Yr) then
    if X<Y then X|{Merge Xr Ys}
    else Y|{Merge Xs Yr}
  end

```

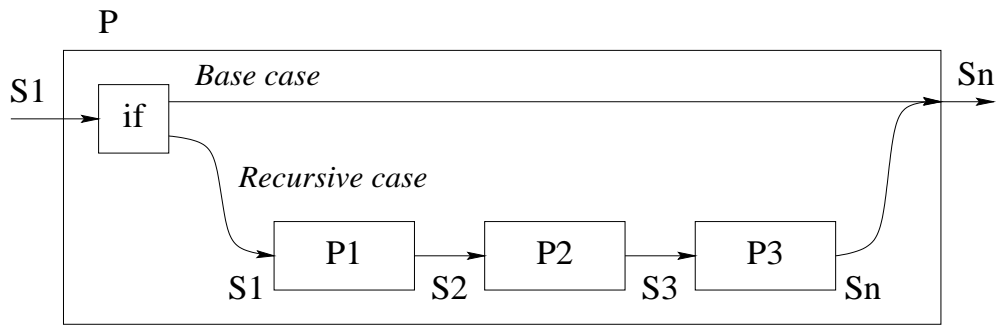


Figure 3.10: Control flow with threaded state

end
end

The type is $\langle \text{fun } \{ \$ \langle \text{List } T \rangle \langle \text{List } T \rangle \} : \langle \text{List } T \rangle \rangle$, where T is either $\langle \text{Int} \rangle$, $\langle \text{Float} \rangle$, or $\langle \text{Atom} \rangle$. We define `split` as a procedure because it has two outputs. It could also be defined as a function returning a *pair* as a single output.

```
proc {Split Xs ?Ys ?Zs}
  case Xs
  of nil then Ys=nil Zs=nil
  [] [X] then Ys=[X] Zs=nil
  [] X1|X2|Xr then Yr Zr in
    Ys=X1|Yr
    Zs=X2|Zr
    {Split Xr Yr Zr}
  end
end
```

The type is $\langle \text{proc } \{ \$ \langle \text{List } T \rangle \langle \text{List } T \rangle \langle \text{List } T \rangle \} \rangle$. Here is the definition of `merge-sort` itself:

```
fun {MergeSort Xs}
  case Xs
  of nil then nil
  [] [X] then [X]
  else Ys Zs in
    {Split Xs Ys Zs}
    {Merge {MergeSort Ys} {MergeSort Zs}}
  end
end
```

Its type is $\langle \text{fun } \{ \$ \langle \text{List } T \rangle \} : \langle \text{List } T \rangle \rangle$ with the same restriction on T as in `Merge`. The splitting up of the input list bottoms out at lists of length zero and one, which can be sorted immediately.

3.4.3 Accumulators

We have seen how to program simple list functions and how to make them iterative. Realistic declarative programming is usually done in a different way, namely by writing functions that are iterative from the start. The idea is to carry state forward at all times and never do a return calculation. A state S is represented by adding a pair of arguments, $S1$ and Sn , to each procedure. This pair is called an *accumulator*. $S1$ represents the *input state* and Sn represents the *output state*. Each procedure definition is then written in a style that looks like this:

```

proc {P X S1 ?Sn}
  if {BaseCase X} then Sn=S1
  else
    {P1 S1 S2}
    {P2 S2 S3}
    {P3 S3 Sn}
  end
end

```

The base case does no calculation, so the output state is the same as the input state ($Sn=S1$). The recursive case threads the state through each recursive call ($P1$, $P2$, and $P3$) and eventually returns it to P . Figure 3.10 gives an illustration. Each arrow represents one state variable. The state value is given at the arrow's tail and passed to the arrow's head. By *state threading* we mean that each procedure's output is the next procedure's input. The technique of threading a state through nested procedure calls is called *accumulator programming*.

Accumulator programming is used in the `IterLength` and `IterReverse` functions we saw before. In these functions the accumulator structure is not so clear, because they are functions. What is happening is that the input state is passed to the function and the output state is what the function returns.

Multiple accumulators

Consider the following procedure, which takes an expression containing identifiers, integers, and addition operations (using label `plus`). It calculates two results: it translates the expression into machine code for a simple stack machine and it calculates the number of instructions in the resulting code.

```

proc {ExprCode E C1 ?Cn S1 ?Sn}
  case E
  of plus(A B) then C2 C3 S2 S3 in
    C2=plus|C1
    S2=S1+1
    {ExprCode B C2 C3 S2 S3}
    {ExprCode A C3 Cn S3 Sn}
  [ ] I then
    Cn=push(I)|C1
    Sn=S1+1

```

```

    end
end

```

This procedure has two accumulators: one to build the list of machine instructions and another to hold the number of instructions. Here is a sample execution:

```

declare Code Size in
  {ExprCode plus(plus(a 3) b) nil Code 0 Size}
  {Browse Size#Code}

```

This displays:

```

5#[push(a) push(3) plus push(b) plus]

```

More complicated programs usually need more accumulators. When writing large declarative programs, we have typically used around half a dozen accumulators simultaneously. The Aquarius Prolog compiler was written in this style [198, 194]. Some of its procedures have as many as 12 accumulators. This means 24 additional arguments! This is difficult to do without mechanical aid. We used an extended DCG preprocessor⁶ that takes declarations of accumulators and adds the arguments automatically [96].

We no longer program in this style; we find that programming with explicit state is simpler and more efficient (see Chapter 6). It is reasonable to use a few accumulators in a declarative program; it is actually quite rare that a declarative program does not need a few. On the other hand, using many is a sign that some of them would probably be better written with explicit state.

Mergesort with an accumulator

In the previous definition of mergesort, we first called the function `Split` to divide the input list into two halves. There is a simpler way to do the mergesort, by using an accumulator. The parameter represents “*the part of the list still to be sorted*”. The specification of `MergeSortAcc` is:

- `S#L2={MergeSortAcc L1 N}` takes an input list `L1` and an integer `N`. It returns two results: `S`, the sorted list of the first `N` elements of `L1`, and `L2`, the remaining elements of `L1`. The two results are paired together with the `#` tupling constructor.

The accumulator is defined by `L1` and `L2`. This gives the following definition:

```

fun {MergeSort Xs}
  fun {MergeSortAcc L1 N}
    if N==0 then
      nil # L1
    elseif N==1 then
      [L1.1] # L1.2
    elseif N>1 then

```

⁶DCG (Definite Clause Grammar) is a grammar notation that is used to hide the explicit threading of accumulators.

```

        NL=N div 2
        NR=N-NL
        Ys # L2 = {MergeSortAcc L1 NL}
        Zs # L3 = {MergeSortAcc L2 NR}
    in
        {Merge Ys Zs} # L3
    end
end
in
    {MergeSortAcc Xs {Length Xs}}.1
end

```

The Merge function is unchanged. Remark that this mergesort does a different split than the previous one. In this version, the split separates the first half of the input list from the second half. In the previous version, split separates the odd-numbered list elements from the even-numbered elements.

This version has the same time complexity as the previous version. It uses less memory because it does not create the two split lists. They are defined implicitly by the combination of the accumulating parameter and the number of elements.

3.4.4 Difference lists

A *difference list* is a pair of two lists, each of which might have an unbound tail. The two lists have a special relationship: it must be possible to get the second list from the first by removing zero or more elements from the front. Here are some examples:

```

X#X                % Represents the empty list
nil#nil            % idem
[a]#[a]            % idem
(a|b|c|X)#X        % Represents [a b c]
(a|b|c|d|X)#(d|X) % idem
[a b c d]#[d]      % idem

```

A difference list is a representation of a standard list. We will talk of the difference list sometimes as a data structure by itself, and sometimes as representing a standard list. Be careful not to confuse these two viewpoints. The difference list `[a b c d]#[d]` might contain the lists `[a b c d]` and `[d]`, but it represents neither of these. It represents the list `[a b c]`.

Difference lists are a special case of *difference structures*. A difference structure is a pair of two partial values where the second value is embedded in the first. The difference structure represents a value that is the first structure minus the second structure. Using difference structures makes it easy to construct iterative computations on many recursive datatypes, e.g., lists or trees. Difference lists and difference structures are special cases of accumulators in which one of the accumulator arguments can be an unbound variable.

The advantage of using difference lists is that when the second list is an unbound variable, another difference list can be appended to it in constant time. To append $(a|b|c|X)\#X$ and $(d|e|f|Y)\#Y$, just bind X to $(d|e|f|Y)$. This creates the difference list $(a|b|c|d|e|f|Y)\#Y$. We have just appended the lists $[a\ b\ c]$ and $[d\ e\ f]$ with a single binding. Here is a function that appends any two difference lists:

```
fun {AppendD D1 D2}
  S1#E1=D1
  S2#E2=D2
in
  E1=S2
  S1#E2
end
```

It can be used like a list append:

```
local X Y in {Browse {AppendD (1|2|3|X)#X (4|5|Y)#Y}} end
```

This displays $(1|2|3|4|5|Y)\#Y$. The standard list append function, defined as follows:

```
fun {Append L1 L2}
  case L1
  of X|T then X|{Append T L2}
  [] nil then L2
end
end
```

iterates on its first argument, and therefore takes time proportional to the length of the first argument. The difference list append is much more efficient: it takes constant time.

The limitation of using difference lists is that they can be appended only *once*. This property means that difference lists can only be used in special circumstances. For example, they are a natural way to write programs that construct big lists in terms of lots of little lists that must be appended together.

Difference lists as defined here originated from Prolog and logic programming [182]. They are the basis of many advanced Prolog programming techniques. As a concept, a difference list lives somewhere between the concept of value and the concept of state. It has the good properties of a value (programs using them are declarative), but it also has some of the power of state because it can be appended once in constant time.

Flattening a nested list

Consider the problem of *flattening* a nested list, i.e., calculating a list that has all the elements of the nested list but is no longer nested. We first give a solution using lists and then we show that a much better solution is possible with difference lists. For the list solution, let us reason with mathematical induction based on the

type `<NestedList>` we defined earlier, in the same way we did with the `LengthL` function:

- Flatten of `nil` is `nil`.
- Flatten of `x|xr` where `x` is a nested list, is `z` where
 flatten of `x` is `y`,
 flatten of `xr` is `yr`, and
 append `y` and `yr` to get `z`.
- Flatten of `x|xr` where `x` is not a list, is `z` where
 flatten of `xr` is `yr`, and
 `z` is `x|yr`.

Following this reasoning, we get the following definition:

```
fun {Flatten Xs}
  case Xs
  of nil then nil
  [] X|Xr andthen {IsList X} then
    {Append {Flatten X} {Flatten Xr}}
  [] X|Xr then
    X|{Flatten Xr}
  end
end
```

Calling:

```
{Browse {Flatten [[a b] [[c] [d]] nil [e [f]]]}
```

displays `[a b c d e f]`. This program is very inefficient because it needs to do many append operations (see Exercises). Now let us reason again in the same way, but with difference lists instead of standard lists:

- Flatten of `nil` is `x#x` (empty difference list).
- Flatten of `x|xr` where `x` is a nested list, is `y1#y4` where
 flatten of `x` is `y1#y2`,
 flatten of `xr` is `y3#y4`, and
 equate `y2` and `y3` to append the difference lists.
- Flatten of `x|xr` where `x` is not a list, is `(x|y1)#y2` where
 flatten of `xr` is `y1#y2`.

We can write the second case as follows:

- Flatten of `x|xr` where `x` is a nested list, is `y1#y4` where
 flatten of `x` is `y1#y2` and
 flatten of `xr` is `y2#y4`.

This gives the following program:

```

fun {Flatten Xs}
  proc {FlattenD Xs ?Ds}
    case Xs
    of nil then Y in Ds=Y#Y
    [] X|Xr andthen {IsList X} then Y1 Y2 Y4 in
      Ds=Y1#Y4
      {FlattenD X Y1#Y2}
      {FlattenD Xr Y2#Y4}
    [] X|Xr then Y1 Y2 in
      Ds=(X|Y1)#Y2
      {FlattenD Xr Y1#Y2}
    end
  end Ys
in
  {FlattenD Xs Ys#nil} Ys
end

```

This program is efficient: it does a single cons operation for each non-list in the input. We convert the difference list returned by `FlattenD` into a regular list by binding its second argument to `nil`. We write `FlattenD` as a procedure because its output is *part* of its last argument, not the whole argument (see Section 2.5.2). It is common style to write a difference list in *two* arguments:

```

fun {Flatten Xs}
  proc {FlattenD Xs ?S E}
    case Xs
    of nil then S=E
    [] X|Xr andthen {IsList X} then Y2 in
      {FlattenD X S Y2}
      {FlattenD Xr Y2 E}
    [] X|Xr then Y1 in
      S=X|Y1
      {FlattenD Xr Y1 E}
    end
  end Ys
in
  {FlattenD Xs Ys nil} Ys
end

```

As a further simplification, we can write `FlattenD` as a function. To do this, we use the fact that `S` is the output:

```

fun {Flatten Xs}
  fun {FlattenD Xs E}
    case Xs
    of nil then E
    [] X|Xr andthen {IsList X} then
      {FlattenD X {FlattenD Xr E}}
    [] X|Xr then

```

```

        X|{FlattenD Xr E}
      end
    end
  in
    {FlattenD Xs nil}
  end

```

What is the role of E ? It gives the “rest” of the output, i.e., when the `FlattenD` call exhausts its own contribution to the output.

Reversing a list

Let us look again at the naive list reverse of the last section. The problem with naive reverse is that it uses a costly append function. Perhaps it will be more efficient with the constant-time append of difference lists? Let us do the naive reverse with difference lists:

- Reverse of `nil` is `X#X` (empty difference list).
- Reverse of `X|Xs` is `Z`, where
 reverse of `Xs` is `Y1#Y2` and
 append `Y1#Y2` and `(X|Y)#Y` together to get `Z`.

Rewrite the last case as follows, by doing the append:

- Reverse of `X|Xs` is `Y1#Y`, where
 reverse of `Xs` is `Y1#Y2` and
 equate `Y2` and `X|Y`.

It is perfectly allowable to move the equate before the reverse (why?). This gives:

- Reverse of `X|Xs` is `Y1#Y`, where
 reverse of `Xs` is `Y1#(X|Y)`.

Here is the final definition:

```

fun {Reverse Xs}
  proc {ReverseD Xs ?Y1 Y}
    case Xs
    of nil then Y1=Y
    [] X|Xr then
      {ReverseD Xr Y1 X|Y}
    end
  end Y1
in
  {ReverseD Xs Y1 nil} Y1
end

```

Look carefully and you will see that this is almost exactly the same iterative solution as in the last section. The only difference between `IterReverse` and `ReverseD` is the argument order: the output of `IterReverse` is the second argument of `ReverseD`. So what's the advantage of using difference lists? With them, we derived `ReverseD` without thinking, whereas to derive `IterReverse` we had to guess an intermediate state that could be updated.

3.4.5 Queues

An important basic data structure is the *queue*. A *queue* is a sequence of elements with an *insert* and a *delete* operation. The insert operation adds an element to one end of the queue and the delete operation removes an element from the other end. We say the queue has FIFO (First-In-First-Out) behavior. Let us investigate how to program queues in the declarative model.

A naive queue

An obvious way to implement queues is by using lists. If `L` represents the queue content, then inserting `X` gives the new queue `X|L` and deleting `X` is done by calling `{ButLast L X L1}`, which binds `X` to the deleted element and returns the new queue in `L1`. `ButLast` returns the last element of `L` in `X` and all elements but the last in `L1`. It can be defined as:

```
proc {ButLast L ?X ?L1}
  case L
  of [Y] then X=Y L1=nil
  [] Y|L2 then L3 in
    L1=Y|L3
    {ButLast L2 X L3}
  end
end
```

The problem with this implementation is that `ButLast` is slow: it takes time proportional to the number of elements in the queue. On the contrary, we would like both the insert and delete operations to be *constant-time*. That is, doing an operation on a given implementation and machine always takes time less than some constant number of seconds. The value of the constant depends on the implementation and machine. Whether or not we can achieve the constant-time goal depends on the expressiveness of the computation model:

- In a strict functional programming language, i.e., the declarative model without dataflow variables (see Section 2.7.1), we cannot achieve it. The best we can do is to get *amortized* constant-time operations [138]. That is, any sequence of n insert and delete operations takes a total time that is proportional to some constant times n . Any individual operation might not be constant-time, however.

- In the declarative model, which extends the strict functional model with dataflow variables, we can achieve the constant-time goal.

We will show how to define both solutions. In both definitions, each operation takes a queue as input and returns a new queue as output. As soon as a queue is used by the program as input to an operation, then it can no longer be used as input to another operation. In other words, there can be only one version of the queue in use at any time. We say that the queue is *ephemeral*.⁷ Each version exists from the moment it is created to the moment it can no longer be used.

Amortized constant-time ephemeral queue

Here is the definition of a queue whose insert and delete operations have constant amortized time bounds. The definition is taken from Okasaki [138]:

```

fun {NewQueue} q(nil nil) end

fun {Check Q}
  case Q of q(nil R) then q({Reverse R} nil) else Q end
end

fun {Insert Q X}
  case Q of q(F R) then {Check q(F X|R)} end
end

fun {Delete Q X}
  case Q of q(F R) then F1 in F=X|F1 {Check q(F1 R)} end
end

fun {IsEmpty Q}
  case Q of q(F R) then F==nil end
end

```

This uses the pair $q(F\ R)$ to represent the queue. F and R are lists. F represents the front of the queue and R represents the back of the queue in reversed form. At any instant, the queue content is given by $\{\text{Append } F\ \{\text{Reverse } R\}\}$. An element can be inserted by adding it to the front of R and deleted by removing it from the front of F . For example, say that $F=[a\ b]$ and $R=[d\ c]$. Deleting the first element returns a and makes $F=[b]$. Inserting the element e makes $R=[e\ d\ c]$. Both operations are constant-time.

To make this representation work, each element in R has to be moved to F sooner or later. When should the move be done? Doing it element by element is inefficient, since it means replacing F by $\{\text{Append } F\ \{\text{Reverse } R\}\}$ each time, which takes time at least proportional to the length of F . The trick is to do it only occasionally. We do it when F becomes empty, so that F is non-nil if and only

⁷Queues implemented with explicit state (see Chapters 6 and 7) are also usually ephemeral.

if the queue is non-empty. This invariant is maintained by the Check function, which moves the content of R to F whenever F is nil.

The Check function does a list reverse operation on R. The reverse takes time proportional to the length of R, i.e., to the number of elements it reverses. Each element that goes through the queue is passed exactly once from R to F. Allocating the reverse's execution time to each element therefore gives a constant time per element. This is why the queue is amortized.

Worst-case constant-time ephemeral queue

We can use difference lists to implement queues whose insert and delete operations have constant worst-case execution times. We use a difference list that ends in an unbound dataflow variable. This lets us insert elements in constant time by binding the dataflow variable. Here is the definition:

```

fun {NewQueue} X in q(0 X X) end

fun {Insert Q X}
  case Q of q(N S E) then E1 in E=X|E1 q(N+1 S E1) end
end

fun {Delete Q X}
  case Q of q(N S E) then S1 in S=X|S1 q(N-1 S1 E) end
end

fun {IsEmpty Q}
  case Q of q(N S E) then N==0 end
end

```

This uses the triple $q(N \ S \ E)$ to represent the queue. At any instant, the queue content is given by the difference list $S\#E$. N is the number of elements in the queue. Why is N needed? Without it, we would not know how many elements were in the queue.

Example use

The following example works with either of the above definitions:

```

declare Q1 Q2 Q3 Q4 Q5 Q6 Q7 in
  Q1={NewQueue}
  Q2={Insert Q1 peter}
  Q3={Insert Q2 paul}
  local X in Q4={Delete Q3 X} {Browse X} end
  Q5={Insert Q4 mary}
  local X in Q6={Delete Q5 X} {Browse X} end
  local X in Q7={Delete Q6 X} {Browse X} end

```

This inserts three elements and deletes them. Each element is inserted before it is deleted. Now let us see what each definition can do that the other cannot.

With the second definition, we can delete an element *before* it is inserted. Doing such a delete returns an unbound variable that will be bound to the corresponding inserted element. So the last four calls in the above example can be changed as follows:

```

local X in Q4={Delete Q3 X} {Browse X} end
local X in Q5={Delete Q4 X} {Browse X} end
local X in Q6={Delete Q5 X} {Browse X} end
Q7={Insert Q6 mary}

```

This works because the bind operation of dataflow variables, which is used both to insert and delete elements, is symmetric.

With the first definition, maintaining multiple versions of the queue simultaneously gives correct results, although the amortized time bounds no longer hold.⁸ Here is an example with two versions:

```

declare Q1 Q2 Q3 Q4 Q5 Q6 in
Q1={NewQueue}
Q2={Insert Q1 peter}
Q3={Insert Q2 paul}
Q4={Insert Q2 mary}
local X in Q5={Delete Q3 X} {Browse X} end
local X in Q6={Delete Q4 X} {Browse X} end

```

Both Q3 and Q4 are calculated from their common ancestor Q2. Q3 contains peter and paul. Q4 contains peter and mary. What do the two Browse calls display?

Persistent queues

Both definitions given above are ephemeral. What can we do if we need to use multiple versions and still require constant-time execution? A queue that supports multiple simultaneous versions is called *persistent*.⁹ Some applications need persistent queues. For example, if during a calculation we pass a queue value to another routine:

```

...
{SomeProc Qa}
Qb={Insert Qa x}
Qc={Insert Qb y}
...

```

⁸To see why not, consider any sequence of n queue operations. For the amortized constant-time bound to hold, the total time for all operations in the sequence must be proportional to n . But what happens if the sequence repeats an “expensive” operation in many versions? This is possible, since we are talking of *any* sequence. Since the time for an expensive operation and the number of versions can both be proportional to n , the total time bound grows as n^2 .

⁹This meaning of persistence should not be confused with persistence as used in transactions and databases (Sections 8.5 and 9.6), which is a completely different concept.

We assume that `SomeProc` can do queue operations but that the caller does not want to see their effects. It follows that we may have two versions of the queue. Can we write queues that keep the time bounds for this case? It can be done if we extend the declarative model with lazy execution. Then both the amortized and worst-case queues can be made persistent. We defer this solution until we present lazy execution in Section 4.5.

For now, let us propose a simple workaround that is often sufficient to make the worst-case queue persistent. It depends on there not being too many simultaneous versions. We define an operation `ForkQ` that takes a queue `Q` and creates two identical versions `Q1` and `Q2`. As a preliminary, we first define a procedure `ForkD` that creates two versions of a difference list:

```
proc {ForkD D ?E ?F}
  D1#nil=D
  E1#E0=E {Append D1 E0 E1}
  F1#F0=F {Append D1 F0 F1}
in skip end
```

The call `{ForkD D E F}` takes a difference list `D` and returns two fresh copies of it, `E` and `F`. `Append` is used to convert a list into a fresh difference list. Note that `ForkD` consumes `D`, i.e., `D` can no longer be used afterwards since its tail is bound. Now we can define `ForkQ`, which uses `ForkD` to make two versions of a queue:

```
proc {ForkQ Q ?Q1 ?Q2}
  q(N S E)=Q
  q(N S1 E1)=Q1
  q(N S2 E2)=Q2
in
  {ForkD S#E S1#E1 S2#E2}
end
```

`ForkQ` consumes `Q` and takes time proportional to the size of the queue. We can rewrite the example as follows using `ForkQ`:

```
...
{ForkQ Qa Qa1 Qa2}
{SomeProc Qa1}
Qb={Insert Qa2 x}
Qc={Insert Qb y}
...
```

This works well if it is acceptable for `ForkQ` to be an expensive operation.

3.4.6 Trees

Next to linear data structures such as lists and queues, trees are the most important recursive data structure in a programmer's repertory. A *tree* is either a

leaf node or a node that contains one or more trees. Nodes can carry additional information. Here is one possible definition:

$$\begin{aligned} \langle \text{Tree} \rangle &::= \text{leaf}(\langle \text{Value} \rangle) \\ &| \text{tree}(\langle \text{Value} \rangle \langle \text{Tree} \rangle_1 \dots \langle \text{Tree} \rangle_n) \end{aligned}$$

The basic difference between a list and a tree is that a list always has a linear structure whereas a tree can have a branching structure. A list always has an element followed by *exactly one* smaller list. A tree has an element followed by *some number* of smaller trees. This number can be any natural number, i.e., zero for leaf nodes and any positive number for non-leaf nodes.

There exist an enormous number of different kinds of trees, with different conditions imposed on their structure. For example, a list is a tree in which non-leaf nodes always have exactly one subtree. In a *binary* tree the non-leaf nodes always have exactly two subtrees. In a *ternary* tree they have exactly three subtrees. In a *balanced* tree, all subtrees of the same node have the same size (i.e., the same number of nodes) or approximately the same size.

Each kind of tree has its own class of algorithms to construct trees, traverse trees, and look up information in trees. This chapter uses several different kinds of trees. We give an algorithm for drawing binary trees in a pleasing way, we show how to use higher-order techniques for calculating with trees, and we implement dictionaries with ordered binary trees.

This section sets the stage for these developments. We will give the basic algorithms that underlie many of these more sophisticated variations. We define ordered binary trees and show how to insert information, look up information, and delete information from them.

Ordered binary tree

An *ordered binary tree* $\langle \text{OBTree} \rangle$ is a binary tree in which each node includes a pair of values:

$$\begin{aligned} \langle \text{OBTree} \rangle &::= \text{leaf} \\ &| \text{tree}(\langle \text{OValue} \rangle \langle \text{Value} \rangle \langle \text{OBTree} \rangle_1 \langle \text{OBTree} \rangle_2) \end{aligned}$$

Each non-leaf node includes the values $\langle \text{OValue} \rangle$ and $\langle \text{Value} \rangle$. The first value $\langle \text{OValue} \rangle$ is any subtype of $\langle \text{Value} \rangle$ that is totally ordered, i.e., it has boolean comparison functions. For example, $\langle \text{Int} \rangle$ (the integer type) is one possibility. The second value $\langle \text{Value} \rangle$ is carried along for the ride. No particular condition is imposed on it.

Let us call the ordered value the *key* and the second value the *information*. Then a binary tree is *ordered* if for each non-leaf node, all the keys in the first subtree are less than the node key, and all the keys in the second subtree are greater than the node key.

Storing information in trees

An ordered binary tree can be used as a repository of information, if we define three operations: looking up, inserting, and deleting entries.

To look up information in an ordered binary tree means to search whether a given key is present in one of the tree nodes, and if so, to return the information present at that node. With the orderedness condition, the search algorithm can eliminate half the remaining nodes at each step. This is called *binary search*. The number of operations it needs is proportional to the *depth* of the tree, i.e., the length of the longest path from the root to a leaf. The look up can be programmed as follows:

```
fun {Lookup X T}
  case T
  of leaf then notfound
  [] tree(Y V T1 T2) then
    if X<Y then {Lookup X T1}
    elseif X>Y then {Lookup X T2}
    else found(V) end
  end
end
```

Calling {Lookup X T} returns found(V) if a node with X is found, and notfound otherwise. Another way to write Lookup is by using **andthen** in the **case** statement:

```
fun {Lookup X T}
  case T
  of leaf then notfound
  [] tree(Y V T1 T2) andthen X==Y then found(V)
  [] tree(Y V T1 T2) andthen X<Y then {Lookup X T1}
  [] tree(Y V T1 T2) andthen X>Y then {Lookup X T2}
  end
end
```

Many developers find the second way more readable because it is more visual, i.e., it gives patterns that show what the tree looks like instead of giving instructions to decompose the tree. In a word, it is more declarative. This makes it easier to verify that it is correct, i.e., to make sure that no cases have been overlooked. In more complicated tree algorithms, pattern matching with **andthen** is a definite advantage over explicit **if** statements.

To insert or delete information in an ordered binary tree, we construct a new tree that is identical to the original except that it has more or less information. Here is the insertion operation:

```
fun {Insert X V T}
  case T
  of leaf then tree(X V leaf leaf)
  [] tree(Y W T1 T2) andthen X==Y then
```

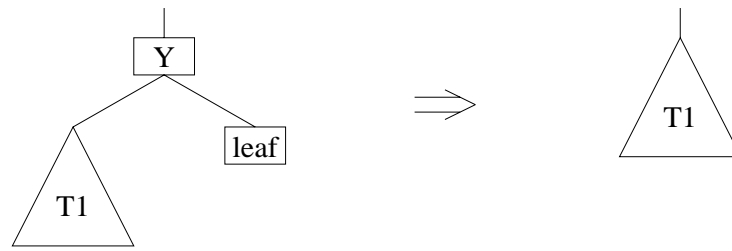


Figure 3.11: Deleting node Y when one subtree is a leaf (easy case)

```

                                tree(X V T1 T2)
[] tree(Y W T1 T2) andthen X<Y then
                                tree(Y W {Insert X V T1} T2)
[] tree(Y W T1 T2) andthen X>Y then
                                tree(Y W T1 {Insert X V T2})
end
end

```

Calling {Insert X V T} returns a new tree that has the pair (X V) inserted in the right place. If T already contains X, then the new tree replaces the old information with V.

Deletion and tree reorganizing

The deletion operation holds a surprise in store. Here is a first try at it:

```

fun {Delete X T}
  case T
  of leaf then leaf
  [] tree(Y W T1 T2) andthen X==Y then leaf
  [] tree(Y W T1 T2) andthen X<Y then
                                tree(Y W {Delete X T1} T2)
  [] tree(Y W T1 T2) andthen X>Y then
                                tree(Y W T1 {Delete X T2})
  end
end

```

Calling {Delete X T} should return a new tree that has no node with key X. If T does not contain X, then T is returned unchanged. Deletion seems simple enough, but the above definition is incorrect. Can you see why?

It turns out that Delete is not as simple as Lookup or Insert. The error in the above definition is that when $X==Y$, the *whole subtree* is removed instead of just a single node. This is only correct if the subtree is degenerate, i.e., if both T1 and T2 are leaf nodes. The fix is not completely obvious: when $X==Y$, we have to *reorganize* the subtree so that it no longer has the key Y but is still an ordered binary tree. There are two cases, illustrated in Figures 3.11 and 3.12.

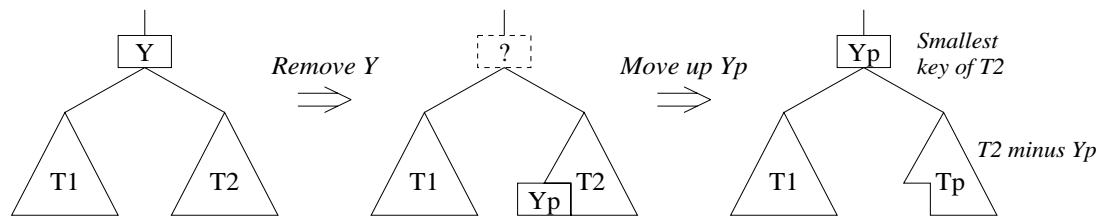


Figure 3.12: Deleting node Y when neither subtree is a leaf (hard case)

Figure 3.11 is the easy case, when one subtree is a leaf. The reorganized tree is simply the other subtree. Figure 3.12 is the hard case, when both subtrees are not leaves. How do we fill the gap after removing Y ? Another key has to take the place of Y , “percolating up” from inside one of the subtrees. The idea is to pick the smallest key of T_2 , call it Y_p , and make it the root of the reorganized tree. The remaining nodes of T_2 make a smaller subtree, call it T_p , which is put in the reorganized tree. This ensures that the reorganized tree is still ordered, since by construction all keys of T_1 are less than Y_p , which is less than all keys of T_p .

It is interesting to see what happens when we repeatedly delete a tree’s roots. This will “hollow out” the tree from the inside, removing more and more of the left-hand part of T_2 . Eventually, T_2 ’s left subtree is removed completely and the right subtree takes its place. Continuing in this way, T_2 shrinks more and more, passing through intermediate stages in which it is a complete, but smaller ordered binary tree. Finally, it disappears completely.

To implement the fix, we use a function `{RemoveSmallest T2}` that returns the smallest key of T_2 , its associated value, and a new tree that lacks this key. With this function, we can write a correct version of `Delete` as follows:

```

fun {Delete X T}
  case T
  of leaf then leaf
  [] tree(Y W T1 T2) andthen X==Y then
    case {RemoveSmallest T2}
    of none then T1
    [] Yp#Vp#Tp then tree(Yp Vp T1 Tp)
    end
  [] tree(Y W T1 T2) andthen X<Y then
    tree(Y W {Delete X T1} T2)
  [] tree(Y W T1 T2) andthen X>Y then
    tree(Y W T1 {Delete X T2})
  end
end

```

The function `RemoveSmallest` returns either a triple $Y_p\#V_p\#T_p$ or the atom `none`. We define it recursively as follows:

```

fun {RemoveSmallest T}
  case T

```

```

    of leaf then none
  [] tree(Y V T1 T2) then
    case {RemoveSmallest T1}
    of none then Y#V#T2
    [] Yp#Vp#Tp then Yp#Vp#tree(Y V Tp T2)
    end
  end
end
end

```

One could also pick the largest element of $T1$ instead of the smallest element of $T2$. This gives much the same result.

The extra difficulty of `Delete` compared to `Insert` or `Lookup` occurs frequently with tree algorithms. The difficulty occurs because an ordered tree satisfies a *global condition*, namely being ordered. Many kinds of trees are defined by global conditions. Algorithms for these trees are complex because they have to maintain the global condition. In addition, tree algorithms are harder to write than list algorithms because the recursion has to combine results from several smaller problems, not just one.

Tree traversal

Traversing a tree means to perform an operation on its nodes in some well-defined order. There are many ways to traverse a tree. Many of these are derived from one of two basic traversals, called *depth-first* and *breadth-first* traversal. Let us look at these traversals.

Depth-first is the simplest traversal. For each node, it visits first the left-most subtree, then the node itself, and then the right-most subtree. This makes it easy to program since it closely follows how nested procedure calls execute. Here is a traversal that displays each node's key and information:

```

proc {DFS T}
  case T
  of leaf then skip
  [] tree(Key Val L R) then
    {DFS L}
    {Browse Key#Val}
    {DFS R}
  end
end
end

```

The astute reader will realize that this depth-first traversal does not make much sense in the declarative model, because it does not calculate any result.¹⁰ We can fix this by adding an accumulator. Here is a traversal that calculates a list of all key/value pairs:

```

proc {DFSAcc T S1 Sn}
  case T

```

¹⁰`Browse` cannot be defined in the declarative model.

```

proc {BFS T}
  fun {TreeInsert Q T}
    if T\=leaf then {Insert Q T} else Q end
  end

  proc {BFSQueue Q1}
    if {IsEmpty Q1} then skip
    else
      X Q2={Delete Q1 X}
      tree(Key Val L R)=X
      in
        {Browse Key#Val}
        {BFSQueue {TreeInsert {TreeInsert Q2 L} R}}
      end
    end
  in
    {BFSQueue {TreeInsert {NewQueue} T}}
  end

```

Figure 3.13: Breadth-first traversal

```

of leaf then Sn=S1
[] tree(Key Val L R) then S2 S3 in
  {DFSAcc L S1 S2}
  S3=Key#Val|S2
  {DFSAcc R S3 Sn}
end
end

```

Breadth-first is a second basic traversal. It first traverses all nodes at depth 0, then all nodes at depth 1, and so forth, going one level deeper at a time. At each level, it traverses the nodes from left to right. The *depth* of a node is the length of the path from the root to the current node, not including the current node. To implement breadth-first traversal, we need a queue to keep track of all the nodes at a given depth. Figure 3.13 shows how it is done. It uses the queue data type we defined in the previous section. The next node to visit comes from the head of the queue. The node's two subtrees are added to the tail of the queue. The traversal will get around to visiting them when all the other nodes of the queue have been visited, i.e., all the nodes at the current depth.

Just like for the depth-first traversal, breadth-first traversal is only useful in the declarative model if supplemented by an accumulator. Figure 3.14 gives an example that calculates a list of all key/value pairs in a tree.

Depth-first traversal can be implemented in a similar way as breadth-first traversal, by using an explicit data structure to keep track of the nodes to visit. To make the traversal depth-first, we simply use a stack instead of a queue. Figure 3.15 defines the traversal, using a list to implement the stack.

```

proc {BFSAcc T S1 ?Sn}
  fun {TreeInsert Q T}
    if T\=leaf then {Insert Q T} else Q end
  end

  proc {BFSQueue Q1 S1 ?Sn}
    if {IsEmpty Q1} then Sn=S1
    else
      X Q2={Delete Q1 X}
      tree(Key Val L R)=X
      S2=Key#Val|S1
      in
        {BFSQueue {TreeInsert {TreeInsert Q2 R} L} S2 Sn}
      end
    end
  in
    {BFSQueue {TreeInsert {NewQueue} T} S1 Sn}
  end

```

Figure 3.14: Breadth-first traversal with accumulator

```

proc {DFS T}
  fun {TreeInsert S T}
    if T\=leaf then T|S else S end
  end

  proc {DFSStack S1}
    case S1
    of nil then skip
    [] X|S2 then
      tree(Key Val L R)=X
      in
        {Browse Key#Val}
        {DFSStack {TreeInsert {TreeInsert S2 R} L}}
      end
    end
  in
    {DFSStack {TreeInsert nil T}}
  end

```

Figure 3.15: Depth-first traversal with explicit stack

How does the new version of DFS compare with the original? Both versions use a stack to remember the subtrees to be visited. In the original, the stack is hidden: it is the semantic stack. There are *two* recursive calls. When the first call is taken, the second one is waiting on the semantic stack. In the new version, the stack is explicit. The new version is tail recursive, just like BFS, so the semantic stack does not grow. The new version simply trades space on the semantic stack for space on the store.

Let us see how much memory the DFS and BFS algorithms use. Assume we have a tree of depth n with 2^n leaf nodes and $2^n - 1$ non-leaf nodes. How big do the stack and queue arguments get? We can prove that the stack has at most n elements and the queue has at most $2^{(n-1)}$ elements. Therefore, DFS is much more economical: it uses memory proportional to the tree depth. BFS uses memory proportional to the size of the tree.

3.4.7 Drawing trees

Now that we have introduced trees and programming with them, let us write a more significant program. We will write a program to draw a binary tree in an aesthetically pleasing way. The program calculates the coordinates of each node. This program is interesting because it traverses the tree for two reasons: to calculate the coordinates and to add the coordinates to the tree itself.

The tree drawing constraints

We first define the tree's type:

```

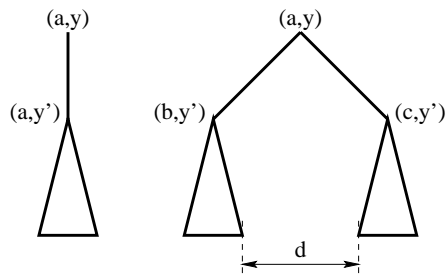
(Tree) ::= tree(key:⟨Literal⟩ val:⟨Value⟩ left: ⟨Tree⟩ right: ⟨Tree⟩)
        | leaf

```

Each node is either a leaf or has two children. In contrast to Section 3.4.6, this uses a *record* to define the tree instead of a tuple. There is a very good reason for this which will become clear when we talk about the principle of independence. Assume that we have the following constraints on how the tree is drawn:

1. There is a minimum horizontal spacing between both subtrees of every node. To be precise, the rightmost node of the left subtree is at a minimal horizontal distance from the leftmost node of the right subtree.
2. If a node has two child nodes, then its horizontal position is the arithmetic average of their horizontal positions.
3. If a node has only one child node, then the child is directly underneath it.
4. The vertical position of a node is proportional to its level in the tree.

In addition, to avoid clutter the drawing shows only the nodes of type `tree`. Figure 3.16 shows these constraints graphically in terms of the coordinates of each node. The example tree of Figure 3.17 is drawn as shown in Figure 3.19.



1. Distance d between subtrees has minimum value
2. If two children exist, a is average of b and c
3. If only one child exists, it is directly below parent
4. Vertical position y is proportional to level in the tree

Figure 3.16: The tree drawing constraints

```

tree(key:a val:111
  left:tree(key:b val:55
    left:tree(key:x val:100
      left:tree(key:z val:56 left:leaf right:leaf)
      right:tree(key:w val:23 left:leaf right:leaf))
    right:tree(key:y val:105 left:leaf
      right:tree(key:r val:77 left:leaf right:leaf)))
  right:tree(key:c val:123
    left:tree(key:d val:119
      left:tree(key:g val:44 left:leaf right:leaf)
      right:tree(key:h val:50
        left:tree(key:i val:5 left:leaf right:leaf)
        right:tree(key:j val:6 left:leaf right:leaf)))
    right:tree(key:e val:133 left:leaf right:leaf)))

```

Figure 3.17: An example tree

Calculating the node positions

The tree drawing algorithm calculates node positions by traversing the tree, passing information between nodes, and calculating values at each node. The traversal has to be done carefully so that all the information is available at the right time. Exactly what traversal is the right one depends on what the constraints are. For the above four constraints, it is sufficient to traverse the tree in a *depth-first* order. In this order, each left subtree of a node is visited before the right subtree. A basic depth-first traversal looks like this:

```

proc {DepthFirst Tree}
  case Tree
  of tree(left:L right:R ...) then
    {DepthFirst L}
    {DepthFirst R}
  [] leaf then
    skip
  end
end

```

The tree drawing algorithm does a depth-first traversal and calculates the (x,y) coordinates of each node during the traversal. As a preliminary to running the algorithm, we extend the tree nodes with the fields `x` and `y` at each node:

```
fun {AddXY Tree}
  case Tree
  of tree(left:L right:R ...) then
    {Adjoin Tree
     tree(x:_ y:_ left:{AddXY L} right:{AddXY R})}
  [] leaf then
    leaf
  end
end
```

The function `AddXY` returns a new tree with the two fields `x` and `y` added to all nodes. It uses the `Adjoin` function which can add new fields to records and override old ones. This is explained in Appendix B.3.2. The tree drawing algorithm will fill in these two fields with the coordinates of each node. If the two fields exist nowhere else in the record, then there is no conflict with any other information in the record.

To implement the tree drawing algorithm, we extend the depth-first traversal by passing two arguments *down* (namely, level in the tree and limit on leftmost position of subtree) and two arguments *up* (namely, horizontal position of the subtree's root and rightmost position of subtree). Downward-passed arguments are sometimes called *inherited* arguments. Upward-passed arguments are sometimes called *synthesized* arguments. With these extra arguments, we have enough information to calculate the positions of all nodes. Figure 3.18 gives the complete tree drawing algorithm. The `Scale` parameter gives the basic size unit of the drawn tree, i.e., the minimum distance between nodes. The initial arguments are `Level=1` and `LeftLim=Scale`. There are four cases, depending on whether a node has two subtrees, one subtree (left or right), or zero subtrees. Pattern matching in the `case` statement picks the right case. This takes advantage of the fact that the tests are done in sequential order.

3.4.8 Parsing

As a second case study of declarative programming, let us write a parser for a small imperative language with syntax similar to Pascal. This uses many of the techniques we have seen, in particular, it uses an accumulator and builds a tree.

What is a parser

A parser is part of a compiler. A compiler is a program that translates a sequence of characters, which represents a program, into a sequence of low-level instructions that can be executed on a machine. In its most basic form, a compiler consists of three parts:

```

Scale=30

proc {DepthFirst Tree Level LeftLim ?RootX ?RightLim}
  case Tree
  of tree(x:X y:Y left:leaf right:leaf ...) then
    X=RootX=RightLim=LeftLim
    Y=Scale*Level
  [] tree(x:X y:Y left:L right:leaf ...) then
    X=RootX
    Y=Scale*Level
    {DepthFirst L Level+1 LeftLim RootX RightLim}
  [] tree(x:X y:Y left:leaf right:R ...) then
    X=RootX
    Y=Scale*Level
    {DepthFirst R Level+1 LeftLim RootX RightLim}
  [] tree(x:X y:Y left:L right:R ...) then
    LRootX LRightLim RRootX RLeftLim
  in
    Y=Scale*Level
    {DepthFirst L Level+1 LeftLim LRootX LRightLim}
    RLeftLim=LRightLim+Scale
    {DepthFirst R Level+1 RLeftLim RRootX RightLim}
    X=RootX=(LRootX+RRootX) div 2
  end
end

```

Figure 3.18: Tree drawing algorithm

- **Tokenizer.** The tokenizer reads a sequence of characters and outputs a sequence of tokens.
- **Parser.** The parser reads a sequence of tokens and outputs an *abstract syntax tree*. This is sometimes called a parse tree.
- **Code generator.** The code generator traverses the syntax tree and generates low-level instructions for a real machine or an abstract machine.

Usually this structure is extended by optimizers to improve the generated code. In this section, we will just write the parser. We first define the input and output formats of the parser.

The parser's input and output languages

The parser accepts a sequence of tokens according to the grammar given in Table 3.2 and outputs an abstract syntax tree. The grammar is carefully designed to be right recursive and deterministic. This means that the choice of grammar

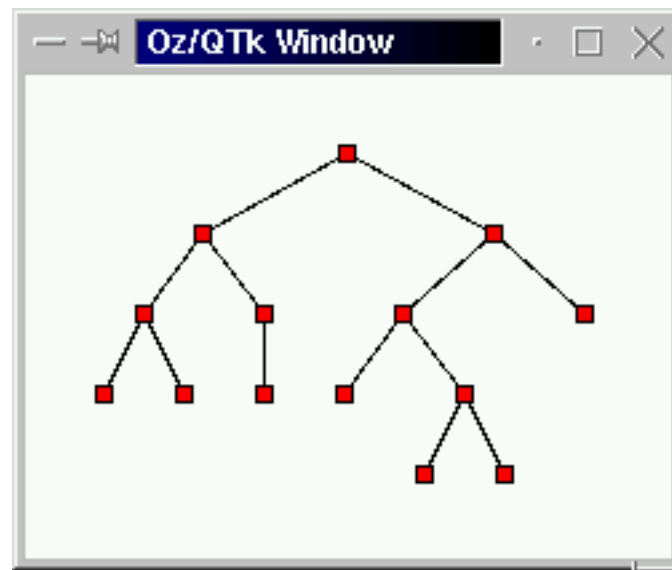


Figure 3.19: The example tree displayed with the tree drawing algorithm

rule is completely determined by the next token. This makes it possible to write a top down, left to right parser with only one token lookahead.

For example, say we want to parse a $\langle \text{Term} \rangle$. It consists of a non-empty series of $\langle \text{Fact} \rangle$ separated by $\langle \text{TOP} \rangle$ tokens. To parse it, we first parse a $\langle \text{Fact} \rangle$. Then we examine the next token. If it is a $\langle \text{TOP} \rangle$, then we know the series continues. If it is not a $\langle \text{TOP} \rangle$, then we know the series has ended, i.e., the $\langle \text{Term} \rangle$ has ended. For this parsing strategy to work, there must be no overlap between $\langle \text{TOP} \rangle$ tokens and the other possible tokens that come after a $\langle \text{Fact} \rangle$. By inspecting the grammar rules, we see that the other tokens must be taken from $\{ \langle \text{EOP} \rangle, \langle \text{COP} \rangle, ;, \text{end}, \text{then}, \text{do}, \text{else},) \}$. We confirm that all the tokens defined by this set are different from the tokens defined by $\langle \text{TOP} \rangle$.

There are two kinds of symbols in Table 3.2: nonterminals and terminals. A *nonterminal symbol* is one that is further expanded according to a grammar rule. A *terminal symbol* corresponds directly to a token in the input. It is not expanded. The nonterminal symbols are $\langle \text{Prog} \rangle$ (complete program), $\langle \text{Stat} \rangle$ (statement), $\langle \text{Comp} \rangle$ (comparison), $\langle \text{Expr} \rangle$ (expression), $\langle \text{Term} \rangle$ (term), $\langle \text{Fact} \rangle$ (factor), $\langle \text{COP} \rangle$ (comparison operator), $\langle \text{EOP} \rangle$ (expression operator), and $\langle \text{TOP} \rangle$ (term operator). To parse a program, start with $\langle \text{Prog} \rangle$ and expand until finding a sequence of tokens that matches the input.

The parser output is a tree (i.e., a nested record) with syntax given in Table 3.3. Superficially, Tables 3.2 and 3.3 have very similar content, but they are actually quite different: the first defines a sequence of tokens and the second defines a tree. The first does not show the structure of the input program—we say it is *flat*. The second exposes this structure—we say it is *nested*. Because it exposes the program's structure, we call the nested record an *abstract syntax*

$\langle \text{Prog} \rangle$::=	program $\langle \text{Id} \rangle$; $\langle \text{Stat} \rangle$ end
$\langle \text{Stat} \rangle$::=	begin { $\langle \text{Stat} \rangle$; } $\langle \text{Stat} \rangle$ end
		$\langle \text{Id} \rangle$:= $\langle \text{Expr} \rangle$
		if $\langle \text{Comp} \rangle$ then $\langle \text{Stat} \rangle$ else $\langle \text{Stat} \rangle$
		while $\langle \text{Comp} \rangle$ do $\langle \text{Stat} \rangle$
		read $\langle \text{Id} \rangle$
		write $\langle \text{Expr} \rangle$
$\langle \text{Comp} \rangle$::=	{ $\langle \text{Expr} \rangle$ $\langle \text{COP} \rangle$ } $\langle \text{Expr} \rangle$
$\langle \text{Expr} \rangle$::=	{ $\langle \text{Term} \rangle$ $\langle \text{EOP} \rangle$ } $\langle \text{Term} \rangle$
$\langle \text{Term} \rangle$::=	{ $\langle \text{Fact} \rangle$ $\langle \text{TOP} \rangle$ } $\langle \text{Fact} \rangle$
$\langle \text{Fact} \rangle$::=	$\langle \text{Integer} \rangle$ $\langle \text{Id} \rangle$ ($\langle \text{Expr} \rangle$)
$\langle \text{COP} \rangle$::=	'==' '!=' '>' '<' '=<' '>='
$\langle \text{EOP} \rangle$::=	'+' '-'
$\langle \text{TOP} \rangle$::=	'*' '/'
$\langle \text{Integer} \rangle$::=	(integer)
$\langle \text{Id} \rangle$::=	(atom)

Table 3.2: The parser's input language (which is a token sequence)

tree. It is *abstract* because it is encoded as a data structure in the language, and no longer in terms of tokens. The parser's role is to extract the structure from the flat input. Without this structure, it is extremely difficult to write the code generator and code optimizers.

The parser program

The main parser call is the function $\{\text{Prog } S1 \text{ } S_n\}$, where $S1$ is an input list of tokens and S_n is the rest of the list after parsing. This call returns the parsed output. For example:

```
declare A Sn in
A={Prog
  [program foo ';' while a '+' 3 '<' b 'do' b ':=' b '+' 1 'end']
  Sn}
{Browse A}
```

displays:

```
prog(foo while('<'('+'(a 3) b) assign(b '+'(b 1))))
```

We give commented program code for the complete parser. Prog is written as follows:

```
fun {Prog S1 Sn}
  Y Z S2 S3 S4 S5
in
  S1=program|S2
```

$\langle \text{Prog} \rangle$	$::=$	<code>prog($\langle \text{Id} \rangle$ $\langle \text{Stat} \rangle$)</code>
$\langle \text{Stat} \rangle$	$::=$	<code>`;`($\langle \text{Stat} \rangle$ $\langle \text{Stat} \rangle$)</code> <code> </code> <code>assign($\langle \text{Id} \rangle$ $\langle \text{Expr} \rangle$)</code> <code> </code> <code>`if`($\langle \text{Comp} \rangle$ $\langle \text{Stat} \rangle$ $\langle \text{Stat} \rangle$)</code> <code> </code> <code>while($\langle \text{Comp} \rangle$ $\langle \text{Stat} \rangle$)</code> <code> </code> <code>read($\langle \text{Id} \rangle$)</code> <code> </code> <code>write($\langle \text{Expr} \rangle$)</code>
$\langle \text{Comp} \rangle$	$::=$	<code>$\langle \text{COP} \rangle$($\langle \text{Expr} \rangle$ $\langle \text{Expr} \rangle$)</code>
$\langle \text{Expr} \rangle$	$::=$	<code>$\langle \text{Id} \rangle$ $\langle \text{Integer} \rangle$ $\langle \text{OP} \rangle$($\langle \text{Expr} \rangle$ $\langle \text{Expr} \rangle$)</code>
$\langle \text{COP} \rangle$	$::=$	<code>`==` `!=` `<>` `<` `=<` `>=`</code>
$\langle \text{OP} \rangle$	$::=$	<code>`+` `-` `*` `/`</code>
$\langle \text{Integer} \rangle$	$::=$	<code>(integer)</code>
$\langle \text{Id} \rangle$	$::=$	<code>(atom)</code>

Table 3.3: The parser's output language (which is a tree)

```

Y={Id S2 S3}
S3=`;`|S4
Z={Stat S4 S5}
S5=`end`|Sn
prog(Y Z)
end

```

The accumulator is threaded through all terminal and nonterminal symbols. Each nonterminal symbol has a procedure to parse it. Statements are parsed with `Stat`, which is written as follows:

```

fun {Stat S1 Sn}
  T|S2=S1
in
  case T
  of begin then
    {Sequence Stat fun {$ X} X==`;` end S2 `end`|Sn}
  [] `if` then C X1 X2 S3 S4 S5 S6 in
    {Comp C S2 S3}
    S3=`then`|S4
    X1={Stat S4 S5}
    S5=`else`|S6
    X2={Stat S6 Sn}
    `if`(C X1 X2)
  [] while then C X S3 S4 in
    C={Comp S2 S3}
    S3=`do`|S4
    X={Stat S4 Sn}
    while(C X)

```