hard data exists to quantify these differences. In our experience, the differences are not great. Programming with static typing is like word processing with a spelling checker: a good writer can get along without it, but it can improve the quality of a text.

Each approach has a role in practical application development. Static typing is recommended when the programming techniques are well-understood and when efficiency and correctness are paramount. Dynamic typing is recommended for rapid development and when programs must be as flexible as possible, such as application prototypes, operating systems, and some artificial intelligence applications.

The choice between static or dynamic typing does not have to be all or nothing. In each approach, a bit of the other can be added, gaining some of its advantages. For example, different kinds of polymorphism (where a variable might have values of several different types) add flexibility to statically-typed functional and object-oriented languages. It is an active research area to design static type systems that capture as much as possible of the flexibility of dynamic type systems, while encouraging good programming style and still permitting compile time verification.

The computation models given in this book are all subsets of the Oz language, which is dynamically typed. One research goal of the Oz project is to explore what programming techniques are possible in a computation model that integrates several programming paradigms. The only way to achieve this goal is with dynamic typing.

When the programming techniques are known, then a possible next step is to design a static type system. While research in increasing the functionality and expressiveness of Oz is still ongoing in the Mozart Consortium, the Alice project at Saarland University in Saarbrücken, Germany, has chosen to add a static type system. Alice is a statically-typed language that has much of the expressiveness of Oz. At the time of writing, Alice is interoperable with Oz (programs can be written partly in Alice and partly in Oz) since it is based on the Mozart implementation.

2.8 Exercises

1. Consider the following statement:

```
proc {P X}
   if X>0 then {P X-1} end
end
```

   Is the identifier occurrence of `P` in the procedure body free or bound? Justify your answer. Hint: this is easy to answer if you first translate to kernel syntax.

2. Section 2.4 explains how a procedure call is executed. Consider the following procedure `MulByN`:

```
declare MulByN N in
N=3
proc {MulByN X ?Y}
   Y=N*X
end
```

together with the call `{MulByN A B}`. Assume that the environment at the call contains $\{\texttt{A} \rightarrow 10, \texttt{B} \rightarrow x_1\}$. When the procedure body is executed, the mapping $\texttt{N} \rightarrow 3$ is added to the environment. Why is this a necessary step? In particular, would not $\texttt{N} \rightarrow 3$ already exist somewhere in the environment at the call? Would not this be enough to ensure that the identifier `N` already maps to `3`? Give an example where `N` does not exist in the environment at the call. Then give a second example where `N` does exist there, but is bound to a different value than `3`.

3. If a function body has an **`if`** statement with a missing **`else`** case, then an exception is raised if the **`if`** condition is false. Explain why this behavior is correct. This situation does not occur for procedures. Explain why not.

4. This exercise explores the relationship between the **`if`** statement and the **`case`** statement.

   (a) Define the **`if`** statement in terms of the **`case`** statement. This shows that the conditional does not add any expressiveness over pattern matching. It could have been added as a linguistic abstraction.

   (b) Define the **`case`** statement in terms of the **`if`** statement, using the operations `Label`, `Arity`, and `´.´` (feature selection).

   This shows that the **`if`** statement is essentially a more primitive version of the **`case`** statement.

5. This exercise tests your understanding of the full **`case`** statement. Given the following procedure:

```
proc {Test X}
   case X
   of a|Z then {Browse ´case´(1)}
   [] f(a) then {Browse ´case´(2)}
   [] Y|Z andthen Y==Z then {Browse ´case´(3)}
   [] Y|Z then {Browse ´case´(4)}
   [] f(Y) then {Browse ´case´(5)}
   else {Browse ´case´(6)} end
end
```

Without executing any code, predict what will happen when you feed `{Test [b c a]}`, `{Test f(b(3))}`, `{Test f(a)}`, `{Test f(a(3))}`, `{Test f(d)}`, `{Test [a b c]}`, `{Test [c a b]}`, `{Test a|a}`, and `{Test ´|´(a b c)}`. Use the kernel translation and the semantics if necessary to make the predictions. After making the predictions, check your understanding by running the examples in Mozart.

6. Given the following procedure:

```
proc {Test X}
   case X of f(a Y c) then {Browse ´case´(1)}
   else {Browse ´case´(2)} end
end
```

Without executing any code, predict what will happen when you feed:

```
declare X Y {Test f(X b Y)}
```

Same for:

```
declare X Y {Test f(a Y d)}
```

Same for:

```
declare X Y {Test f(X Y d)}
```

Use the kernel translation and the semantics if necessary to make the predictions. After making the predictions, check your understanding by running the examples in Mozart. Now run the following example:

```
declare X Y
if f(X Y d)==f(a Y c) then {Browse ´case´(1)}
else {Browse ´case´(2)} end
```

Does this give the same result or a different result than the previous example? Explain the result.

7. Given the following code:

```
declare Max3 Max5
proc {SpecialMax Value ?SMax}
   fun {SMax X}
      if X>Value then X else Value end
   end
end
{SpecialMax 3 Max3}
{SpecialMax 5 Max5}
```

Without executing any code, predict what will happen when you feed:

```
{Browse [{Max3 4} {Max5 4}]}
```

Check your understanding by running this example in Mozart.

8. This exercise explores the relationship between linguistic abstractions and higher-order programming.

   (a) Define the function `AndThen` as follows:

   ```
   fun {AndThen BP1 BP2}
      if {BP1} then {BP2} else false end
   end
   ```

   Does the following call:

   ```
   {AndThen fun {$} ⟨expression⟩1 end
            fun {$} ⟨expression⟩2 end}
   ```

   give the same result as ⟨expression⟩$_1$ **andthen** ⟨expression⟩$_2$? Does it avoid the evaluation of ⟨expression⟩$_2$ in the same situations?

   (b) Write a function `OrElse` that is to **orelse** as `AndThen` is to **andthen**. Explain its behavior.

9. This exercise examines the importance of tail recursion, in the light of the semantics given in the chapter. Consider the following two functions:

   ```
   fun {Sum1 N}
      if N==0 then 0 else N+{Sum1 N-1} end
   end

   fun {Sum2 N S}
      if N==0 then S else {Sum2 N-1 N+S} end
   end
   ```

   (a) Expand the two definitions into kernel syntax. It should be clear that `Sum2` is tail recursive and `Sum1` is not.

   (b) Execute the two calls `{Sum1 10}` and `{Sum2 10 0}` by hand, using the semantics of this chapter to follow what happens to the stack and the store. How large does the stack become in either case?

   (c) What would happen in the Mozart system if you would call `{Sum1 100000000}` or `{Sum2 100000000 0}`? Which one is likely to work? Which one is not? Try both on Mozart to verify your reasoning.

10. Consider the following function `SMerge` that merges two sorted lists:

    ```
    fun {SMerge Xs Ys}
       case Xs#Ys
       of nil#Ys then Ys
       [] Xs#nil then Xs
       [] (X|Xr)#(Y|Yr) then
    ```

```
         if X=<Y then
            X|{SMerge Xr Ys}
         else
            Y|{SMerge Xs Yr}
         end
      end
   end
```

Expand `SMerge` into the kernel syntax. Note that `X#Y` is a tuple of two arguments that can also be written `´#´(X Y)`. The resulting procedure should be tail recursive, if the rules of Section 2.5.2 are followed correctly.

11. Last call optimization is important for much more than just recursive calls. Consider the following mutually recursive definition of the functions `IsOdd` and `IsEven`:

```
fun {IsEven X}
   if X==0 then true else {IsOdd X-1} end
end

fun {IsOdd X}
   if X==0 then false else {IsEven X-1} end
end
```

We say that these functions are *mutually recursive* since each function calls the other. Mutual recursion can be generalized to any number of functions. A set of functions is *mutually recursive* if they can be put in a sequence such that each function calls the next and the last calls the first. For this exercise, show that the calls `{IsOdd N}` and `{IsEven N}` execute with constant stack size for all nonnegative `N`. In general, if each function in a mutually-recursive set has just one function call in its body, and this function call is a last call, then all functions in the set will execute with their stack size bounded by a constant.

12. Section 2.7.2 explains that the bind operation is actually much more general than just binding variables: it makes two partial values equal (if they are compatible). This operation is called *unification*. The purpose of this exercise is to explore why unification is interesting. Consider the three unifications `X=[a Z]`, `Y=[W b]`, and `X=Y`. Show that the variables `X`, `Y`, `Z`, and `W` are bound to the same values, no matter in which order the three unifications are done. In Chapter 4 we will see that this order-independence is important for declarative concurrency.

Chapter 3

Declarative Programming Techniques

"S'il vous plaît... dessine-moi un arbre!"
"*If you please – draw me a tree!*"
– Freely adapted from Le Petit Prince, *Antoine de Saint-Exupéry* (1900–1944)

"The nice thing about declarative programming is that you can write a specification and run it as a program. The nasty thing about declarative programming is that some clear specifications make incredibly bad programs. The hope of declarative programming is that you can move from a specification to a reasonable program without leaving the language."
– The Craft of Prolog, *Richard O'Keefe* (?–)

Consider any computational operation, i.e., a program fragment with inputs and outputs. We say the operation is *declarative* if, whenever called with the same arguments, it returns the same results independent of any other computation state. Figure 3.1 illustrates the concept. A declarative operation is *independent* (does not depend on any execution state outside of itself), *stateless*[1] (has no internal execution state that is remembered between calls), and *deterministic* (always gives the same results when given the same arguments). We will show that all programs written using the computation model of the last chapter are declarative.

Why declarative programming is important

Declarative programming is important because of two properties:

- **Declarative programs are compositional**. A declarative program consists of components that can each be written, tested, and proved correct

[1] The concept of "stateless" is sometimes called "immutable".

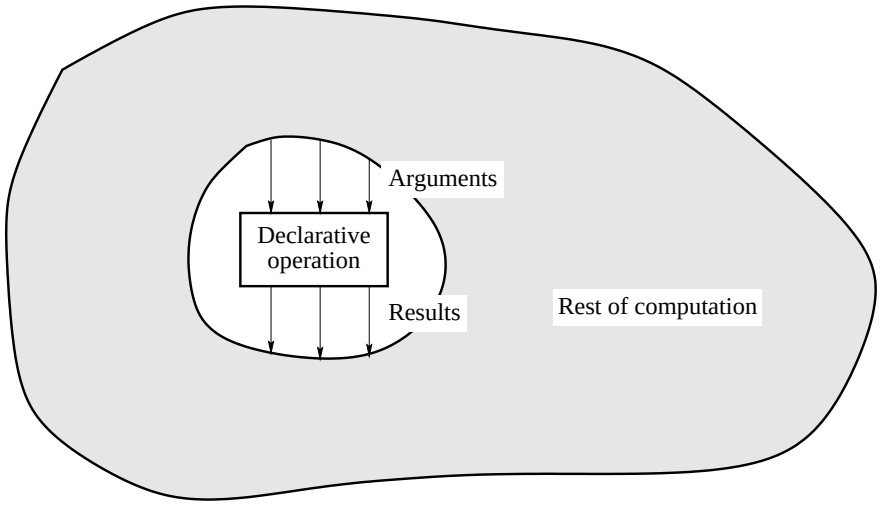


Figure 3.1: A declarative operation inside a general computation

independently of other components and of its own past history (previous calls).

- **Reasoning about declarative programs is simple**. Programs written in the declarative model are easier to reason about than programs written in more expressive models. Since declarative programs compute only values, simple algebraic and logical reasoning techniques can be used.

These two properties are important both for programming *in the large* and *in the small*, respectively. It would be nice if all programs could easily be written in the declarative model. Unfortunately, this is not the case. The declarative model is a good fit for certain kinds of programs and a bad fit for others. This chapter and the next examine the programming techniques of the declarative model and explain what kinds of programs can and cannot be easily written in it.

We start by looking more closely at the first property. Let us define a *component* as a precisely delimited program fragment with well-defined *inputs* and *outputs*. A component can be defined in terms of a set of simpler components. For example, in the declarative model a procedure is one kind of component. The application program is the topmost component in a hierarchy of components. The hierarchy bottoms out in primitive components which are provided by the system.

In a declarative program, the interaction between components is determined solely by each component's inputs and outputs. Consider a program with a declarative component. This component can be understood on its own, without having to understand the rest of the program. The effort needed to understand the whole program is the *sum* of the efforts needed for the declarative component and for the rest.

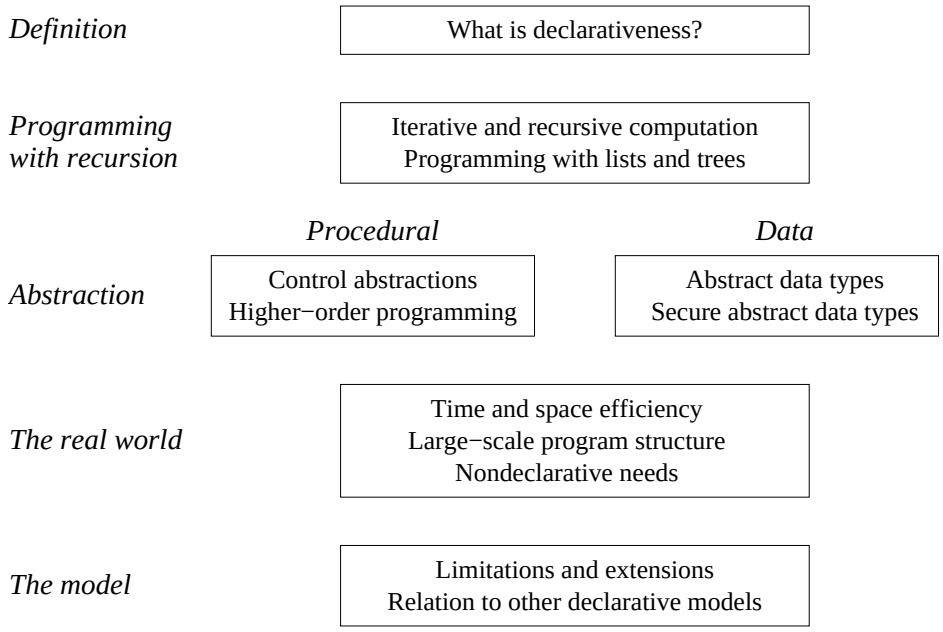


Figure 3.2: Structure of the chapter

If there would be a more intimate interaction between the component and the rest of the program, then they could not be understood independently. They would have to be understood together, and the effort needed would be much bigger. For example, it might be (roughly) proportional to the *product* of the efforts needed for each part. For a program with many components that interact intimately, this very quickly explodes, making understanding difficult or impossible. An example of such an intimate interaction is a concurrent program with shared state, as explained in Chapter 8.

Intimate interactions are often necessary. They cannot be "legislated away" by programming in a model that does not directly support them (as Section 4.7 clearly explains). But an important principle is that they should only be used when necessary and not otherwise. To support this principle, as many components as possible should be declarative.

Writing declarative programs

The simplest way to write a declarative program is to use the declarative model of the last chapter. The basic operations on data types are declarative, e.g., the arithmetic, list, and record operations. It is possible to combine declarative operations to make new declarative operations, if certain rules are followed. Combining declarative operations according to the operations of the declarative model will result in a declarative operation. This is explained in Section 3.1.3.

The standard rule in algebra that "equals can be replaced by equals" is another example of a declarative combination. In programming languages, this property

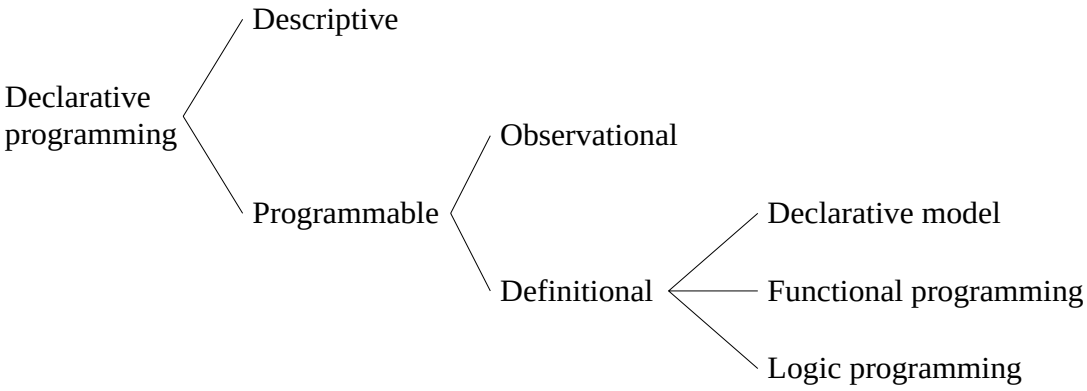


Figure 3.3: A classification of declarative programming

is called *referential transparency*. It greatly simplifies reasoning about programs. For example, if we know that $f(a) = a^2$, then we can replace $f(a)$ by a^2 in any other place where it occurs. The equation $b = 7f(a)^2$ then becomes $b = 7a^4$. This is possible because $f(a)$ is declarative: it depends only on its arguments and not on any other computation state.

The basic technique for writing declarative programs is to consider the program as a set of recursive function definitions, using higher-orderness to simplify the program structure. A *recursive function* is one whose definition body refers to the function itself, either directly or indirectly. *Direct* recursion means that the function itself is used in the body. *Indirect* recursion means that the function refers to another function that directly or indirectly refers to the original function. *Higher-orderness* means that functions can have other functions as arguments and results. This ability underlies all the techniques for building abstractions that we will show in the book. Higher-orderness can compensate somewhat for the lack of expressiveness of the declarative model, i.e., it makes it easy to code limited forms of concurrency and state in the declarative model.

Structure of the chapter

This chapter explains how to write practical declarative programs. The chapter is roughly organized into the six parts shown in Figure 3.2. The first part defines "declarativeness". The second part gives an overview of programming techniques. The third and fourth parts explain procedural and data abstraction. The fifth part shows how declarative programming interacts with the rest of the computing environment. The sixth part steps back to reflect on the usefulness of the declarative model and situate it with respect to other models.

⟨s⟩ ::=		
	skip	Empty statement
|	⟨s⟩$_1$ ⟨s⟩$_2$	Statement sequence
|	**local** ⟨x⟩ **in** ⟨s⟩ **end**	Variable creation
|	⟨x⟩$_1$=⟨x⟩$_2$	Variable-variable binding
|	⟨x⟩=⟨v⟩	Value creation

Table 3.1: The descriptive declarative kernel language

3.1 What is declarativeness?

The declarative model of Chapter 2 is an especially powerful way of writing declarative programs, since all programs written in it will be declarative by this fact alone. But it is still only one way out of many for doing declarative programming. Before explaining how to program in the declarative model, let us situate it with respect to the other ways of being declarative. Let us also explain why programs written in it are always declarative.

3.1.1 A classification of declarative programming

We have defined declarativeness in one particular way, so that reasoning about programs is simplified. But this is not the only way to make precise what declarative programming is. Intuitively, it is programming by defining the *what* (the results we want to achieve) without explaining the *how* (the algorithms, etc., needed to achieve the results). This vague intuition covers many different ideas. Let us try to explain them. Figure 3.3 classifies the most important ones. The first level of classification is based on the expressiveness. There are two possibilities:

- A *descriptive* declarativeness. This is the least expressive. The declarative "program" just defines a data structure. Table 3.1 defines a language at this level. This language can only define records! It contains just the first five statements of the kernel language in Table 2.1. Section 3.8.2 shows how to use this language to define graphical user interfaces. Other examples are a formatting language like HTML, which gives the structure of a document without telling how to do the formatting, or an information exchange language like XML, which is used to exchange information in an open format that is easily readable by all. The descriptive level is too weak to write general programs. So why is it interesting? Because it consists of data structures that are easy to calculate with. The records of Table 3.1, HTML and XML documents, and the declarative user interfaces of Section 3.8.2 can all be created and transformed easily by a program.
- A *programmable* declarativeness. This is as expressive as a Turing machine.[2]

[2] A Turing machine is a simple formal model of computation that is as powerful as any

For example, Table 2.1 defines a language at this level. See the introduction to Chapter 6 for more on the relationship between the descriptive and programmable levels.

There are two fundamentally different ways to view programmable declarativeness:

- A *definitional* view, where declarativeness is a property of the component implementation. For example, programs written in the declarative model are guaranteed to be declarative, because of properties of the model.

- An *observational* view, where declarativeness is a property of the component interface. The observational view follows the principle of abstraction: that to use a component it is enough to know its specification without knowing its implementation. The component just has to *behave* declaratively, i.e., *as if* it were independent, stateless, and deterministic, without necessarily being written in a declarative computation model.

This book uses both the definitional and observational views. When we are interested in looking inside a component, we will use the definitional view. When we are interested in how a component behaves, we will use the observational view.

Two styles of definitional declarative programming have become particularly popular: the functional and the logical. In the functional style, we say that a component defined as a mathematical function is declarative. Functional languages such as Haskell and Standard ML follow this approach. In the logical style, we say that a component defined as a logical relation is declarative. Logic languages such as Prolog and Mercury follow this approach. It is harder to formally manipulate functional or logical programs than descriptive programs, but they still follow simple algebraic laws.[3] The declarative model used in this chapter encompasses both functional and logic styles.

The observational view lets us use declarative components in a declarative program even if they are written in a nondeclarative model. For example, a database interface can be a valuable addition to a declarative language. Yet, the implementation of this interface is almost certainly not going to be logical or functional. It suffices that it *could have been* defined declaratively. Sometimes a declarative component will be written in a functional or logical style, and sometimes it will not be. In later chapters we will build declarative components in nondeclarative models. We will not be dogmatic about the matter; we will consider the component to be declarative if it behaves declaratively.

computer that can be built, as far as is known in the current state of computer science. That is, any computation that can be programmed on any computer can also be programmed on a Turing machine.

[3] For programs that do not use the nondeclarative abilities of these languages.

3.1.2 Specification languages

Proponents of declarative programming sometimes claim that it allows to dispense with the implementation, since the specification is all there is. That is, the specification *is* the program. This is true in a formal sense, but not in a practical sense. Practically, declarative programs are very much like other programs: they require algorithms, data structures, structuring, and reasoning about the order of operations. This is because declarative languages can only use mathematics that can be implemented efficiently. There is a trade-off between expressiveness and efficiency. Declarative programs are usually a lot longer than what a specification could be. So the distinction between specification and implementation still makes sense, even for declarative programs.

It is possible to define a declarative language that is much more expressive than what we use in this book. Such a language is called a *specification language*. It is usually impossible to implement specification languages efficiently. This does not mean that they are impractical; on the contrary. They are an important tool for thinking about programs. They can be used together with a *theorem prover*, i.e., a program that can do certain kinds of mathematical reasoning. Practical theorem provers are not completely automatic; they need human help. But they can take over much of the drudgery of reasoning about programs, i.e., the tedious manipulation of mathematical formulas. With the aid of the theorem prover, a developer can often prove very strong properties about his or her program. Using a theorem prover in this way is called *proof engineering*. Up to now, proof engineering is only practical for small programs. But this is enough for it to be used successfully when safety is of critical importance, e.g., when lives are at stake, such as in medical apparatus or public transportation.

Specification languages are outside the scope of this book.

3.1.3 Implementing components in the declarative model

Combining declarative operations according to the operations of the declarative model always results in a declarative operation. This section explains why this is so. We first define more precisely what it means for a statement to be declarative. Given any statement in the declarative model. Partition the free variable identifiers in the statement into inputs and outputs. Then, given any binding of the input identifiers to partial values and the output identifiers to unbound variables, executing the statement will give one of three results: (1) some binding of the output variables, (2) suspension, or (3) an exception. If the statement is declarative, then for the same bindings of the inputs, the result is always the same.

For example, consider the statement `Z=X`. Assume that `X` is the input and `Z` is the output. For any binding of `X` to a partial value, executing this statement will bind `Z` to the same partial value. Therefore the statement is declarative.

We can use this result to prove that the statement

```
if X>Y then Z=X else Z=Y end
```

is declarative. Partition the statement's three free identifiers, `X`, `Y`, `Z`, into two input identifiers `X` and `Y` and one output identifier `Z`. Then, if `X` and `Y` are bound to any partial values, the statement's execution will either block or bind `Z` to the same partial value. Therefore the statement is declarative.

We can do this reasoning for all operations in the declarative model:

- First, all basic operations in the declarative model are declarative. This includes all operations on basic types, which are explained in Chapter 2.
- Second, combining declarative operations with the constructs of the declarative model gives a declarative operation. The following five compound statements exist in the declarative model:
  - The statement sequence.
  - The **`local`** statement.
  - The **`if`** statement.
  - The **`case`** statement.
  - Procedure declaration, i.e., the statement ⟨x⟩=⟨v⟩ where ⟨v⟩ is a procedure value.

  They allow building statements out of other statements. All these ways of combining statements are deterministic (if their component statements are deterministic, then so are they) and they do not depend on any context.

3.2 Iterative computation

We will now look at how to program in the declarative model. We start by looking at a very simple kind of program, the *iterative computation*. An iterative computation is a loop whose stack size is bounded by a constant, independent of the number of iterations. This kind of computation is a basic programming tool. There are many ways to write iterative programs. It is not always obvious when a program is iterative. Therefore, we start by giving a general schema that shows how to construct many interesting iterative computations in the declarative model.

3.2.1 A general schema

An important class of iterative computations starts with an initial state S_0 and transforms the state in successive steps until reaching a final state S_{final}:

$$S_0 \rightarrow S_1 \rightarrow \cdots \rightarrow S_{\text{final}}$$

An iterative computation of this class can be written as a general schema:

```
fun {Sqrt X}
   Guess=1.0
in
   {SqrtIter Guess X}
end
fun {SqrtIter Guess X}
   if {GoodEnough Guess X} then Guess
   else
      {SqrtIter {Improve Guess X} X}
   end
end
fun {Improve Guess X}
  (Guess + X/Guess) / 2.0
end
fun {GoodEnough Guess X}
  {Abs X-Guess*Guess}/X < 0.00001
end
fun {Abs X} if X<0.0 then ~X else X end end
```

Figure 3.4: Finding roots using Newton's method (first version)

```
fun {Iterate Si}
   if {IsDone Si} then Si
   else Si+1 in
      Si+1={Transform Si}
      {Iterate Si+1}
   end
end
```

In this schema, the functions *IsDone* and *Transform* are problem dependent. Let us prove that any program that follows this schema is iterative. We will show that the stack size does not grow when executing `Iterate`. For clarity, we give just the statements on the semantic stack, leaving out the environments and the store:

- Assume the initial semantic stack is [`R={Iterate` S_0`}`].
- Assume that `{`*IsDone* S_0`}` returns **false**. Just after executing the **if**, the semantic stack is [S_1`={`*Transform* S_0`}`, `R={Iterate` S_1`}`].
- After executing `{`*Transform* S_1`}`, the semantic stack is [`R={Iterate` S_1`}`].

We see that the semantic stack has just one element at every recursive call, namely [`R={Iterate` S_{i+1}`}`].

3.2.2 Iteration with numbers

A good example of iterative computation is Newton's method for calculating the square root of a positive real number x. The idea is to start with a guess g of the square root, and to improve this guess iteratively until it is accurate enough. The improved guess g' is the average of g and x/g:

$$g' = (g + x/g)/2.$$

To see that the improved guess is beter, let us study the difference between the guess and $\sqrt{x}$:

$$\epsilon = g - \sqrt{x}$$

Then the difference between g' and $\sqrt{x}$ is:

$$\epsilon' = g' - \sqrt{x} = (g + x/g)/2 - \sqrt{x} = \epsilon^2/2g$$

For convergence, ϵ' should be smaller than ϵ. Let us see what conditions that this imposes on x and g. The condition $\epsilon' < \epsilon$ is the same as $\epsilon^2/2g < \epsilon$, which is the same as $\epsilon < 2g$. (Assuming that $\epsilon > 0$, since if it is not, we start with ϵ', which is always greater than 0.) Substituting the definition of ϵ, we get the condition $\sqrt{x} + g > 0$. If $x > 0$ and the initial guess $g > 0$, then this is always true. The algorithm therefore always converges.

Figure 3.4 shows one way of defining Newton's method as an iterative computation. The function `{SqrtIter Guess X}` calls `{SqrtIter {Improve Guess X} X}` until `Guess` satisfies the condition `{GoodEnough Guess X}`. It is clear that this is an instance of the general schema, so it is an iterative computation. The improved guess is calculated according to the formula given above. The "good enough" check is $|x - g^2|/x < 0.00001$, i.e., the square root has to be accurate to five decimal places. This check is *relative*, i.e., the error is divided by x. We could also use an *absolute* check, e.g., something like $|x - g^2| < 0.00001$, where the magnitude of the error has to be less than some constant. Why is using a relative check better when calculating square roots?

3.2.3 Using local procedures

In the Newton's method program of Figure 3.4, several "helper" routines are defined: `SqrtIter`, `Improve`, `GoodEnough`, and `Abs`. These routines are used as building blocks for the main function `Sqrt`. In this section, we will discuss where to define helper routines. The basic principle is that a helper routine defined only as an aid to define another routine should not be visible elsewhere. (We use the word "routine" for both functions and procedures.)

In the Newton example, `SqrtIter` is only needed inside `Sqrt`, `Improve` and `GoodEnough` are only needed inside `SqrtIter`, and `Abs` is a utility function that could be used elsewhere. There are two basic ways to express this visibility, with somewhat different semantics. The first way is shown in Figure 3.5: the helper

```
local
   fun {Improve Guess X}
     (Guess + X/Guess) / 2.0
   end
   fun {GoodEnough Guess X}
     {Abs X-Guess*Guess}/X < 0.00001
   end
   fun {SqrtIter Guess X}
      if {GoodEnough Guess X} then Guess
      else
         {SqrtIter {Improve Guess X} X}
      end
   end
in
   fun {Sqrt X}
      Guess=1.0
   in
      {SqrtIter Guess X}
   end
end
```

Figure 3.5: Finding roots using Newton's method (second version)

routines are defined outside of `Sqrt` in a **`local`** statement. The second way is shown in Figure 3.6: each helper routine is defined inside of the routine that needs it.[4]

In Figure 3.5, there is a trade-off between readability and visibility: `Improve` and `GoodEnough` could be defined local to `SqrtIter` only. This would result in two levels of local declarations, which is harder to read. We have decided to put all three helper routines in the same local declaration.

In Figure 3.6, each helper routine sees the arguments of its enclosing routine as external references. These arguments are precisely those with which the helper routines are called. This means we could simplify the definition by removing these arguments from the helper routines. This gives Figure 3.7.

There is a trade-off between putting the helper definitions outside the routine that needs them or putting them inside:

- Putting them *inside* (Figures 3.6 and 3.7) lets them see the arguments of the main routines as external references, according to the lexical scoping rule (see Section 2.4.3). Therefore, they need fewer arguments. But each time the main routine is invoked, new helper routines are created. This means that new procedure values are created.

- Putting them *outside* (Figures 3.4 and 3.5) means that the procedure values are created once and for all, for all calls to the main routine. But then the

[4] We leave out the definition of `Abs` to avoid needless repetition.

```
fun {Sqrt X}
   fun {SqrtIter Guess X}
      fun {Improve Guess X}
        (Guess + X/Guess) / 2.0
      end
      fun {GoodEnough Guess X}
        {Abs X-Guess*Guess}/X < 0.00001
      end
   in
      if {GoodEnough Guess X} then Guess
      else
         {SqrtIter {Improve Guess X} X}
      end
   end
   Guess=1.0
in
   {SqrtIter Guess X}
end
```

Figure 3.6: Finding roots using Newton's method (third version)

```
fun {Sqrt X}
   fun {SqrtIter Guess}
      fun {Improve}
        (Guess + X/Guess) / 2.0
      end
      fun {GoodEnough}
        {Abs X-Guess*Guess}/X < 0.00001
      end
   in
      if {GoodEnough} then Guess
      else
         {SqrtIter {Improve}}
      end
   end
   Guess=1.0
in
   {SqrtIter Guess}
end
```

Figure 3.7: Finding roots using Newton's method (fourth version)

```
fun {Sqrt X}
   fun {Improve Guess}
     (Guess + X/Guess) / 2.0
   end
   fun {GoodEnough Guess}
     {Abs X-Guess*Guess}/X < 0.00001
   end
   fun {SqrtIter Guess}
      if {GoodEnough Guess} then Guess
      else
         {SqrtIter {Improve Guess}}
      end
   end
   Guess=1.0
in
   {SqrtIter Guess}
end
```

Figure 3.8: Finding roots using Newton's method (fifth version)

helper routines need more arguments so that the main routine can pass information to them.

In Figure 3.7, new definitions of `Improve` and `GoodEnough` are created on each iteration of `SqrtIter`, whereas `SqrtIter` itself is only created once. This suggests a good trade-off, where `SqrtIter` is local to `Sqrt` and both `Improve` and `GoodEnough` are outside `SqrtIter`. This gives the final definition of Figure 3.8, which we consider the best in terms of both efficiency and visibility.

3.2.4 From general schema to control abstraction

The general schema of Section 3.2.1 is a programmer aid. It helps the programmer design efficient programs but it is not seen by the computation model. Let us go one step further and provide the general schema as a program component that can be used by other components. We say that the schema becomes a *control abstraction*, i.e., an abstraction that can be used to provide a desired control flow. Here is the general schema:

```
fun {Iterate Si}
   if {IsDone Si} then Si
   else Si+1 in
      Si+1={Transform Si}
      {Iterate Si+1}
   end
end
```

This schema implements a general **while** loop with a calculated result. To make the schema into a control abstraction, we have to parameterize it by extracting the parts that vary from one use to another. There are two such parts: the functions *IsDone* and *Transform*. We make these two parts into parameters of `Iterate`:

```
fun {Iterate S IsDone Transform}
   if {IsDone S} then S
   else S1 in
      S1={Transform S}
      {Iterate S1 IsDone Transform}
   end
end
```

To use this control abstraction, the arguments `IsDone` and `Transform` are given one-argument functions. Passing functions as arguments to functions is part of a range of programming techniques called *higher-order programming*. These techniques are further explained in Section 3.6. We can make `Iterate` behave exactly like `SqrtIter` by passing it the functions `GoodEnough` and `Improve`. This can be written as follows:

```
fun {Sqrt X}
   {Iterate
      1.0
      fun {$ G} {Abs X-G*G}/X<0.00001 end
      fun {$ G} (G+X/G)/2.0 end}
end
```

This uses two function values as arguments to the control abstraction. This is a powerful way to structure a program because it separates the general control flow from this particular use. Higher-order programming is especially helpful for structuring programs in this way. If this control abstraction is used often, the next step could be to provide it as a linguistic abstraction.

3.3 Recursive computation

Iterative computations are a special case of a more general kind of computation, called *recursive computation*. Let us see the difference between the two. Recall that an iterative computation can be considered as simply a loop in which a certain action is repeated some number of times. Section 3.2 implements this in the declarative model by introducing a control abstraction, the function `Iterate`. The function first tests a condition. If the condition is false, it does an action and then calls itself.

Recursion is more general than this. A recursive function can call itself anywhere in the body and can call itself more than once. In programming, recursion occurs in two major ways: in functions and in data types. A function is recursive if its definition has at least one call to itself. The iteration abstraction of

Section 3.2 is a simple case. A data type is recursive if it is defined in terms of itself. For example, a list is defined in terms of a smaller list. The two forms of recursion are strongly related since recursive functions can be used to calculate with recursive data types.

We saw that an iterative computation has a constant stack size. This is not always the case for a recursive computation. Its stack size may grow as the input grows. Sometimes this is unavoidable, e.g., when doing calculations with trees, as we will see later. In other cases, it can be avoided. An important part of declarative programming is to avoid a growing stack size whenever possible. This section gives an example of how this is done. We start with a typical case of a recursive computation that is not iterative, namely the naive definition of the factorial function. The mathematical definition is:

$$0! = 1$$
$$n! = n \cdot (n-1)! \text{ if } n > 0$$

This is a recurrence equation, i.e., the factorial $n!$ is defined in terms of a factorial with a smaller argument, namely $(n-1)!$. The naive program follows this mathematical definition. To calculate `{Fact N}` there are two possibilities, namely `N=0` or `N>0`. In the first case, return `1`. In the second case, calculate `{Fact N-1}`, multiply by `N`, and return the result. This gives the following program:

```
fun {Fact N}
   if N==0 then 1
   elseif N>0 then N*{Fact N-1}
   else raise domainError end
   end
end
```

This defines the factorial of a big number in terms of the factorial of a smaller number. Since all numbers are nonnegative, they will bottom out at zero and the execution will finish.

Note that factorial is a partial function. It is not defined for negative `N`. The program reflects this by raising an exception for negative `N`. The definition in Chapter 1 has an error since for negative `N` it goes into an infinite loop.

We have done two things when writing `Fact`. First, we followed the mathematical definition to get a correct implementation. Second, we reasoned about termination, i.e., we showed that the program terminates for all legal arguments, i.e., arguments inside the function's domain.

3.3.1 Growing stack size

This definition of factorial gives a computation whose maximum stack size is proportional to the function argument `N`. We can see this by using the semantics. First translate `Fact` into the kernel language:

```
proc {Fact N ?R}
```

```
      if N==0 then R=1
      elseif N>0 then N1 R1 in
         N1=N-1
         {Fact N1 R1}
         R=N*R1
      else raise domainError end
      end
   end
```

Already we can guess that the stack size might grow, since the multiplication comes *after* the recursive call. That is, during the recursive call the stack has to keep information about the multiplication for when the recursive call returns. Let us follow the semantics and calculate by hand what happens when executing the call `{Fact 5 R}`. For clarity, we simplify slightly the presentation of the abstract machine by substituting the value of a store variable into the environment. That is, the environment $\{..., \texttt{N} \rightarrow n, ...\}$ is written as $\{..., \texttt{N} \rightarrow 5, ...\}$ if the store is $\{..., n = 5, ...\}$.

- The initial semantic stack is $[(\texttt{\{Fact N R\}}, \{\texttt{N} \rightarrow 5, \texttt{R} \rightarrow r_0\})]$.
- At the first call:

$$\begin{array}{ll}[& (\texttt{\{Fact N1 R1\}}, \{\texttt{N1} \rightarrow 4, \texttt{R1} \rightarrow r_1, ...\}),\\ & (\texttt{R=N*R1}, \{\texttt{R} \rightarrow r_0, \texttt{R1} \rightarrow r_1 \texttt{N} \rightarrow 5, ...\})]\end{array}$$

- At the second call:

$$\begin{array}{ll}[& (\texttt{\{Fact N1 R1\}}, \{\texttt{N1} \rightarrow 3, \texttt{R1} \rightarrow r_2, ...\}),\\ & (\texttt{R=N*R1}, \{\texttt{R} \rightarrow r_1, \texttt{R1} \rightarrow r_2, \texttt{N} \rightarrow 4, ...\}),\\ & (\texttt{R=N*R1}, \{\texttt{R} \rightarrow r_0, \texttt{R1} \rightarrow r_1, \texttt{N} \rightarrow 5, ...\})]\end{array}$$

- At the third call:

$$\begin{array}{ll}[& (\texttt{\{Fact N1 R1\}}, \{\texttt{N1} \rightarrow 2, \texttt{R1} \rightarrow r_3, ...\}),\\ & (\texttt{R=N*R1}, \{\texttt{R} \rightarrow r_2, \texttt{R1} \rightarrow r_3, \texttt{N} \rightarrow 3, ...\}),\\ & (\texttt{R=N*R1}, \{\texttt{R} \rightarrow r_1, \texttt{R1} \rightarrow r_2, \texttt{N} \rightarrow 4, ...\}),\\ & (\texttt{R=N*R1}, \{\texttt{R} \rightarrow r_0, \texttt{R1} \rightarrow r_1, \texttt{N} \rightarrow 5, ...\})]\end{array}$$

It is clear that the stack grows bigger by one statement per call. The last recursive call is the fifth, which returns immediately with $r_5 = 1$. Then five multiplications are done to get the final result $r_0 = 120$.

3.3.2 Substitution-based abstract machine

This example shows that the abstract machine of Chapter 2 can be rather cumbersome for hand calculation. This is because it keeps both variable identifiers and store variables, using environments to map from one to the other. This is

realistic; it is how the abstract machine is implemented on a real computer. But it is not so nice for hand calculation.

We can make a simple change to the abstract machine that makes it much easier to use for hand calculation. The idea is to replace the identifiers in the statements by the store entities that they refer to. This is called doing a *substitution*. For example, the statement `R=N*R1` becomes $r_2 = 3 * r_3$ when substituted according to $\{\texttt{R} \rightarrow r_2, \texttt{N} \rightarrow 3, \texttt{R1} \rightarrow r_3\}$.

The substitution-based abstract machine has no environments. It directly substitutes identifiers by store entities in statements. For the recursive factorial example, this gives the following:

- The initial semantic stack is [`{Fact 5` r_0`}`].
- At the first call: [`{Fact 4` r_1`}`, r_0`=5*`r_1].
- At the second call: [`{Fact 3` r_2`}`, r_1`=4*`r_2, r_0`=5*`r_1].
- At the third call: [`{Fact 2` r_3`}`, r_2`=3*`r_3, r_1`=4*`r_2, r_0`=5*`r_1].

As before, we see that the stack grows by one statement per call. We summarize the differences between the two versions of the abstract machine:

- The environment-based abstract machine, defined in Chapter 2, is faithful to the implementation on a real computer, which uses environments. However, environments introduce an extra level of indirection, so they are hard to use for hand calculation.
- The substitution-based abstract machine is easier to use for hand calculation, because there are many fewer symbols to manipulate. However, substitutions are costly to implement, so they are generally not used in a real implementation.

Both versions do the same store bindings and the same manipulations of the semantic stack.

3.3.3 Converting a recursive to an iterative computation

Factorial is simple enough that is can be rearranged to become iterative. Let us see how this is done. Later on, we will give a systematic way of making iterative computations. For now, we just give a hint. In the previous calculation:

```
R=(5*(4*(3*(2*(1*1)))))
```

it is enough to rearrange the numbers:

```
R=(((((1*5)*4)*3)*2)*1)
```

Then the calculation can be done incrementally, starting with `1*5`. This gives `5`, then `20`, then `60`, then `120`, and finally `120`. The iterative definition of factorial that does things this way is:

```
fun {Fact N}
   fun {FactIter N A}
      if N==0 then A
      elseif N>0 then {FactIter N-1 A*N}
      else raise domainError end
      end
   end
in
   {FactIter N 1}
end
```

The function that does the iteration, `FactIter`, has a second argument `A`. This argument is crucial; without it an iterative factorial is impossible. The second argument is not apparent in the simple mathematical definition of factorial we used first. We had to do some reasoning to bring it in.

3.4 Programming with recursion

Recursive computations are at the heart of declarative programming. This section shows how to write in this style. We show the basic techniques for programming with lists, trees, and other recursive data types. We show how to make the computation iterative when possible. The section is organized as follows:

- The first step is **defining** recursive data types. Section 3.4.1 gives a simple notation that lets us define the most important recursive data types.
- The most important recursive data type is the **list**. Section 3.4.2 presents the basic programming techniques for lists.
- Efficient declarative programs have to define iterative computations. Section 3.4.3 presents **accumulators**, a systematic technique to achieve this.
- Computations often build data structures incrementally. Section 3.4.4 presents **difference lists**, an efficient technique to achieve this while keeping the computation iterative.
- An important data type related to the list is the **queue**. Section 3.4.5 shows how to implement queues efficiently. It also introduces the basic idea of *amortized* efficiency.
- The second most important recursive data type, next to linear structures such as lists and queues, is the **tree**. Section 3.4.6 gives the basic programming techniques for trees.
- Sections 3.4.7 and 3.4.8 give two realistic **case studies**, a tree drawing algorithm and a parser, that between them use many of the techniques of this section.

3.4.1 Type notation

The list type is a subset of the record type. There are other useful subsets of the record type, e.g., binary trees. Before going into writing programs, let us introduce a simple notation to define lists, trees, and other subtypes of records. This will help us to write functions on these types.

A *list* `Xs` is either `nil` or `X|Xr` where `Xr` is a list. Other subsets of the record type are also useful. For example, a binary tree can be defined as `leaf(key:K value:V)` or `tree(key:K value:V left:LT right:RT)` where `LT` and `RT` are both binary trees. How can we write these types in a concise way? Let us create a notation based on the context-free grammar notation for defining the syntax of the kernel language. The nonterminals represent either types or values. Let us use the type hierarchy of Figure 2.16 as a basis: all the types in this hierarchy will be available as predefined nonterminals. So ⟨Value⟩ and ⟨Record⟩ both exist, and since they are sets of values, we can say ⟨Record⟩ ⊂ ⟨Value⟩. Now we can define lists:

```
⟨List⟩ ::= ⟨Value⟩ ´|´ ⟨List⟩
         | nil
```

This means that a value is in ⟨List⟩ if it has one of two forms. Either it is `X|Xr` where `X` is in ⟨Value⟩ and `Xr` is in ⟨List⟩. Or it is the atom `nil`. This is a recursive definition of ⟨List⟩. It can be proved that there is just one set ⟨List⟩ that is the smallest set that satisfies this definition. The proof is beyond the scope of this book, but can be found in any introductory book on semantics, e.g., [208]. We take this smallest set as the value of ⟨List⟩. Intuitively, ⟨List⟩ can be constructed by starting with `nil` and repeatedly applying the grammar rule to build bigger and bigger lists.

We can also define lists whose elements are of a given type:

```
⟨List T⟩ ::= T ´|´ ⟨List T⟩
           | nil
```

Here T is a type variable and ⟨List T⟩ is a type function. Applying the type function to any type returns the type of a list of that type. For example, ⟨List ⟨Int⟩⟩ is the list of integer type. Observe that ⟨List ⟨Value⟩⟩ is equal to ⟨List⟩ (since they have identical definitions).

Let us define a binary tree whose keys are literals and whose elements are of type T:

```
⟨BTree T⟩ ::= tree(key: ⟨Literal⟩ value: T
                   left: ⟨BTree T⟩ right: ⟨BTree T⟩)
            | leaf(key: ⟨Literal⟩ value: T)
```

The type of a procedure is ⟨**proc** {$ T_1, ...,T_n}⟩, where T_1, ..., T_n are the types of its arguments. The procedure's type is sometimes called the *signature* of the procedure, because it gives some key information about the procedure in a concise

form. The type of a function is ⟨**fun** {$ T$_1$, ...,T$_n$}: T⟩, which is equivalent to ⟨**proc** {$ T$_1$, ...,T$_n$, T}⟩. For example, the type ⟨**fun** {$ ⟨List⟩ ⟨List⟩}: ⟨List⟩ ⟩ is a function with two list arguments that returns a list.

Limits of the notation

This type notation can define many useful sets of values, but its expressiveness is definitely limited. Here are some cases where the notation is not good enough:

- The notation cannot define the positive integers, i.e., the subset of ⟨Int⟩ whose elements are all greater than zero.
- The notation cannot define sets of partial values. For example, difference lists cannot be defined.

We can extend the notation to handle the first case, e.g., by adding boolean conditions.[5] In the examples that follow, we will add these conditions in the text when they are needed. This means that the type notation is *descriptive*: it gives logical assertions about the set of values that a variable may take. There is no claim that the types could be checkable by a compiler. On the contrary, they often cannot be checked. Even types that are simple to specify, such as the positive integers, cannot in general be checked by a compiler.

3.4.2 Programming with lists

List values are very concise to create and to take apart, yet they are powerful enough to encode any kind of complex data structure. The original Lisp language got much of its power from this idea [120]. Because of lists' simple structure, declarative programming with them is easy and powerful. This section gives the basic techniques of programming with lists:

- *Thinking recursively*: the basic approach is to solve a problem in terms of smaller versions of the problem.
- *Converting recursive to iterative computations*: naive list programs are often wasteful because their stack size grows with the input size. We show how to use state transformations to make them practical.
- *Correctness of iterative computations*: a simple and powerful way to reason about iterative computations is by using state invariants.
- *Constructing programs by following the type*: a function that calculates with a given type almost always has a recursive structure that closely mirrors the type definition.

[5] This is similar to the way we define language syntax in Section 2.1.1: a context-free notation with extra conditions when they are needed.

We end up this section with a bigger example, the mergesort algorithm. Later sections show how to make the writing of iterative functions more systematic by introducing accumulators and difference lists. This lets us write iterative functions from the start. We find that these techniques "scale up", i.e., they work well even for large declarative programs.

Thinking recursively

A list is a *recursive* data structure: it is defined in terms of a smaller version of itself. To write a function that calculates on lists we have to follow this recursive structure. The function consists of two parts:

- A *base case*. For small lists (say, of zero, one, or two elements), the function computes the answer directly.
- A *recursive case*. For bigger lists, the function computes the result in terms of the results of one or more smaller lists.

As our first example, we take a simple recursive function that calculates the length of a list according to this technique:

```
fun {Length Ls}
   case Ls
   of nil then 0
   [] _|Lr then 1+{Length Lr}
   end
end
{Browse {Length [a b c]}}
```

Its type signature is ⟨**fun** {$ ⟨List⟩}: ⟨Int⟩⟩, a function of one list that returns an integer. The base case is the empty list `nil`, for which the function returns `0`. The recursive case is any other list. If the list has length n, then its tail has length $n-1$. The tail is smaller than the original list, so the program will terminate.

Our second example is a function that appends two lists `Ls` and `Ms` together to make a third list. The question is, on which list do we use induction? Is it the first or the second? We claim that the induction has to be done on the first list. Here is the function:

```
fun {Append Ls Ms}
   case Ls
   of nil then Ms
   [] X|Lr then X|{Append Lr Ms}
   end
end
```

Its type signature is ⟨**fun** {$ ⟨List⟩ ⟨List⟩}: ⟨List⟩⟩. This function follows exactly the following two properties of append:

- append(`nil`, m) $= m$

- append($x|l,\ m$) $=$ x `|` append(l, m)

The recursive case always calls `Append` with a smaller first argument, so the program terminates.

Recursive functions and their domains

Let us define the function `Nth` to get the nth element of a list.

```
fun {Nth Xs N}
   if N==1 then Xs.1
   elseif N>1 then {Nth Xs.2 N-1}
   end
end
```

Its type is ⟨**fun** `{$` ⟨List⟩ ⟨Int⟩`}:` ⟨Value⟩⟩. Remember that a list `Xs` is either `nil` or a tuple `X|Y` with two arguments. `Xs.1` gives `X` and `Xs.2` gives `Y`. What happens when we feed the following:

```
{Browse {Nth [a b c d] 5}}
```

The list has only four elements. Trying to ask for the fifth element means trying to do `Xs.1` or `Xs.2` when `Xs=nil`. This will raise an exception. An exception is also raised if `N` is not a positive integer, e.g., when `N=0`. This is because there is no **`else`** clause in the **`if`** statement.

This is an example of a general technique to define functions: always use statements that raise exceptions when values are given outside their domains. This will maximize the chances that the function as a whole will raise an exception when called with an input outside its domain. We cannot guarantee that an exception will always be raised in this case, e.g., `{Nth 1|2|3 2}` returns 2 while `1|2|3` is not a list. Such guarantees are hard to come by. They can sometimes be obtained in statically-typed languages.

The **`case`** statement also behaves correctly in this regard. Using a **`case`** statement to recurse over a list will raise an exception when its argument is not a list. For example, let us define a function that sums all the elements of a list of integers:

```
fun {SumList Xs}
   case Xs
   of nil then 0
   [] X|Xr then X+{SumList Xr}
   end
end
```

Its type is ⟨**fun** `{$` ⟨List ⟨Int⟩⟩`}:` ⟨Int⟩⟩. The input must be a list of integers because `SumList` internally uses the integer 0. The following call:

```
{Browse {SumList [1 2 3]}}
```

displays 6. Since `Xs` can be one of two values, namely `nil` or `X|Xr`, it is natural to use a **`case`** statement. As in the `Nth` example, not using an **`else`** in the case

will raise an exception if the argument is outside the domain of the function. For example:

```
{Browse {SumList 1|foo}}
```

raises an exception because `1|foo` is not a list, and the definition of `SumList` assumes that its input is a list.

Naive definitions are often slow

Let us define a function to reverse the elements of a list. Start with a recursive definition of list reversal:

- Reverse of `nil` is `nil`.
- Reverse of `X|Xs` is `Z`, where
  reverse of `Xs` is `Ys`, and
  append `Ys` and `[X]` to get `Z`.

This works because `X` is moved from the front to the back. Following this recursive definition, we can immediately write a function:

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

Its type is ⟨**fun** `{$` ⟨List⟩`}:` ⟨List⟩⟩. Is this function efficient? To find out, we have to calculate its execution time given an input list of length n. We can do this rigorously with the techniques of Section 3.5. But even without these techniques, we can see intuitively what happens. There will be n recursive calls followed by n calls to `Append`. Each `Append` call will have a list of length $n/2$ on average. The total execution time is therefore proportional to $n \cdot n/2$, namely n^2. This is rather slow. We would expect that reversing a list, which is not exactly a complex calculation, would take time proportional to the input length and not to its square.

This program has a second defect: the stack size grows with the input list length, i.e., it defines a recursive computation that is not iterative. Naively following the recursive definition of reverse has given us a rather inefficient result! Luckily, there are simple techniques for getting around both these inefficiencies. They will let us define linear-time iterative computations whenever possible. We will see two useful techniques: state transformations and difference lists.

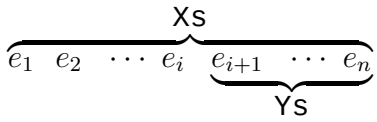
Converting recursive to iterative computations

Let us see how to convert recursive computations into iterative ones. Instead of using `Reverse`, we take a simpler function that calculates the length of a list:

```
fun {Length Xs}
   case Xs of nil then 0
   [] _|Xr then 1+{Length Xr}
   end
end
```

Note that the `SumList` function has the same structure. This function is linear-time but the stack size is proportional to the recursion depth, which is equal to the length of `Xs`. Why does this problem occur? It is because the addition `1+{Length Xr}` happens *after* the recursive call. The recursive call is not last, so the function's environment cannot be recovered before it.

How can we calculate the list length with an iterative computation, which has bounded stack size? To do this, we have to formulate the problem as a sequence of *state transformations*. That is, we start with a state S_0 and we transform it successively, giving S_1, S_2, ..., until we reach the final state S_{final}, which contains the answer. To calculate the list length, we can take *the length i of the part of the list already seen* as the state. Actually, this is only part of the state. The rest of the state is *the part* `Ys` *of the list not yet seen*. The complete state S_i is then the pair $(i, \texttt{Ys})$. The general intermediate case is as follows for state S_i (where the full list `Xs` is $[e_1\ e_2\ \cdots\ e_n]$):



At each recursive call, i will be incremented by 1 and `Ys` reduced by one element. This gives us the function:

```
fun {IterLength I Ys}
   case Ys
   of nil then I
   [] _|Yr then {IterLength I+1 Yr}
   end
end
```

Its type is ⟨**fun** `{$` ⟨Int⟩ ⟨List⟩`}:` ⟨Int⟩⟩. Note the difference with the previous definition. Here the addition `I+1` is done *before* the recursive call to `IterLength`, which is the last call. We have defined an iterative computation.

In the call `{IterLength I Ys}`, the initial value of `I` is `0`. We can hide this initialization by defining `IterLength` as a local procedure. The final definition of `Length` is therefore:

```
local
   fun {IterLength I Ys}
      case Ys
      of nil then I
      [] _|Yr then {IterLength I+1 Yr}
      end
   end
```

```
in
   fun {Length Xs}
      {IterLength 0 Xs}
   end
end
```

This defines an iterative computation to calculate the list length. Note that we define `IterLength` *outside* of `Length`. This avoids creating a new procedure value each time `Length` is called. There is no advantage to defining `IterLength` inside `Length`, since it does not use `Length`'s argument `Xs`.

We can use the same technique on `Reverse` as we used for `Length`. In the case of `Reverse`, the state uses *the reverse of the part of the list already seen* instead of its length. Updating the state is easy: we just put a new list element in front. The initial state is `nil`. This gives the following version of `Reverse`:

```
local
   fun {IterReverse Rs Ys}
      case Ys
      of nil then Rs
      [] Y|Yr then {IterReverse Y|Rs Yr}
      end
   end
in
   fun {Reverse Xs}
      {IterReverse nil Xs}
   end
end
```

This version of `Reverse` is both a linear-time and an iterative computation.

Correctness with state invariants

Let us prove that `IterLength` is correct. We will use a general technique that works well for `IterReverse` and other iterative computations. The idea is to define a property $P(S_i)$ of the state that we can prove is always true, i.e., it is a *state invariant*. If P is chosen well, then the correctness of the computation follows from $P(S_{\text{final}})$. For `IterLength` we define P as follows:

$$P((i, \texttt{Ys})) \quad \equiv \quad (\text{length}(\texttt{Xs}) = i + \text{length}(\texttt{Ys}))$$

where length(`L`) gives the length of the list `L`. This combines i and `Ys` in such a way that we suspect it is a state invariant. We use induction to prove this:

- First prove $P(S_0)$. This follows directly from $S_0 = (0, \texttt{Xs})$.
- Assuming $P(S_i)$ and S_i is not the final state, prove $P(S_{i+1})$. This follows from the semantics of the **`case`** statement and the function call. Write $S_i = (i, \texttt{Ys})$. We are not in the final state, so `Ys` is of nonzero length. From the semantics, `I+1` adds 1 to i and the **`case`** statement removes one element from `Ys`. Therefore $P(S_{i+1})$ holds.