

3.7.1 A declarative stack

To start this section, let us give a simple example of an abstract data type, a stack $\langle \text{Stack } T \rangle$ whose elements are of type T . Assume the stack has four operations, with the following types:

```

    <fun {NewStack}: <Stack T>>
    <fun {Push <Stack T> T}: <Stack T>>
    <fun {Pop <Stack T> T}: <Stack T>>
    <fun {IsEmpty <Stack T>}: <Bool>>

```

This set of operations and their types defines the interface of the abstract data type. These operations satisfy certain laws:

- $\{\text{IsEmpty } \{\text{NewStack}\}\} = \text{true}$. A new stack is always empty.
- For any E and $S0$, $S1 = \{\text{Push } S0 \ E\}$ and $S0 = \{\text{Pop } S1 \ E\}$ hold. Pushing an element and then popping gives the same element back.
- $\{\text{Pop } \{\text{EmptyStack}\}\}$ raises an error. No elements can be popped off an empty stack.

These laws are independent of any particular implementation, or said differently, all implementations have to satisfy these laws. Here is an implementation of the stack that satisfies the laws:

```

    fun {NewStack} nil end
    fun {Push S E} E|S end
    fun {Pop S E} case S of X|S1 then E=X S1 end end
    fun {IsEmpty S} S==nil end

```

Here is another implementation that satisfies the laws:

```

    fun {NewStack} stackEmpty end
    fun {Push S E} stack(E S) end
    fun {Pop S E} case S of stack(X S1) then E=X S1 end end
    fun {IsEmpty S} S==stackEmpty end

```

A program that uses the stack will work with either implementation. This is what we mean by saying that stack is an *abstract* data type.

A functional programming look

Attentive readers will notice an unusual aspect of these two definitions: `Pop` is written using a functional syntax, but one of its arguments is an output! We could have written `Pop` as follows:

```

    fun {Pop S} case S of X|S1 then X#S1 end end

```

which returns the two outputs as a pair, but we chose not to. Writing $\{\text{Pop } S \ E\}$ is an example of programming with a *functional look*, which uses functional syntax for operations that are not necessarily mathematical functions. We consider that

```

fun {NewDictionary} nil end
fun {Put Ds Key Value}
  case Ds
  of nil then [Key#Value]
  [] (K#V)|Dr andthen Key==K then
    (Key#Value) | Dr
  [] (K#V)|Dr andthen K>Key then
    (Key#Value)|(K#V)|Dr
  [] (K#V)|Dr andthen K<Key then
    (K#V)|{Put Dr Key Value}
  end
end
fun {CondGet Ds Key Default}
  case Ds
  of nil then Default
  [] (K#V)|Dr andthen Key==K then
    V
  [] (K#V)|Dr andthen K>Key then
    Default
  [] (K#V)|Dr andthen K<Key then
    {CondGet Dr Key Default}
  end
end
fun {Domain Ds}
  {Map Ds fun {$ K#_} K end}
end

```

Figure 3.27: Declarative dictionary (with linear list)

this is justified for programs that have a clear directionality in the flow of data. It can be interesting to highlight this directionality even if the program is not functional. In some cases this can make the program more concise and more readable. The functional look should be used sparingly, though, and only in cases where it is clear that the operation is not a mathematical function. We will use the functional look occasionally throughout the book, when we judge it appropriate.

For the stack, the functional look lets us highlight the symmetry between `Push` and `Pop`. It makes it clear syntactically that both operations take a stack and return a stack. Then, for example, the output of `Pop` can be immediately passed as input to a `Push`, without needing an intermediate **case** statement.

3.7.2 A declarative dictionary

Let us give another example, an extremely useful abstract data type called a *dictionary*. A dictionary is a finite mapping from a set of simple constants to

a set of language entities. Each constant maps to one language entity. The constants are called *keys* because they unlock the path to the entity, in some intuitive sense. We will use atoms or integers as constants. We would like to be able to create the mapping dynamically, i.e., by adding new keys during the execution. This gives the following set of basic functions on the new type $\langle \text{Dict} \rangle$:

- $\langle \text{fun } \{\text{NewDictionary}\} : \langle \text{Dict} \rangle \rangle$ returns a new empty dictionary.
- $\langle \text{fun } \{\text{Put } \langle \text{Dict} \rangle \langle \text{Feature} \rangle \langle \text{Value} \rangle\} : \langle \text{Dict} \rangle \rangle$ takes a dictionary and returns a new dictionary that adds the mapping $\langle \text{Feature} \rangle \rightarrow \langle \text{Value} \rangle$. If $\langle \text{Feature} \rangle$ already exists, then the new dictionary replaces it with $\langle \text{Value} \rangle$.
- $\langle \text{fun } \{\text{Get } \langle \text{Dict} \rangle \langle \text{Feature} \rangle\} : \langle \text{Value} \rangle \rangle$ returns the value corresponding to $\langle \text{Feature} \rangle$. If there is none, an exception is raised.
- $\langle \text{fun } \{\text{Domain } \langle \text{Dict} \rangle\} : \langle \text{List } \langle \text{Feature} \rangle \rangle \rangle$ returns a list of the keys in $\langle \text{Dict} \rangle$.

For this example we define the $\langle \text{Feature} \rangle$ type as $\langle \text{Atom} \rangle \mid \langle \text{Int} \rangle$.

List-based implementation

Figure 3.27 shows an implementation in which the dictionary is represented as a list of pairs $\text{Key}\#\text{Value}$ that are sorted on the key. Instead of Get , we define a slightly more general access operation, CondGet :

- $\langle \text{fun } \{\text{CondGet } \langle \text{Dict} \rangle \langle \text{Feature} \rangle \langle \text{Value} \rangle_1\} : \langle \text{Value} \rangle_2 \rangle$ returns the value corresponding to $\langle \text{Feature} \rangle$. If $\langle \text{Feature} \rangle$ is not present, then it returns $\langle \text{Value} \rangle_1$.

CondGet is almost as easy to implement as Get and is very useful, as we will see in the next example.

This implementation is extremely slow for large dictionaries. Given a uniform distribution of keys, Put needs on average to look at half the list. CondGet needs on average to look at half the list, whether the element is present or not. We see that the number of operations is $O(n)$ for dictionaries with n keys. We say that the implementation does a *linear search*.

Tree-based implementation

A more efficient implementation of dictionaries is possible by using an ordered binary tree, as defined in Section 3.4.6. Put is simply Insert and CondGet is very similar to Lookup . This gives the definitions of Figure 3.28. In this implementation, the Put and CondGet operations take $O(\log n)$ time and space for a tree with n nodes, given that the tree is “reasonably balanced”. That is, for each node, the sizes of the left and right subtrees should not be “too different”.

```

fun {NewDictionary} leaf end
fun {Put Ds Key Value}
    % ... similar to Insert
end
fun {CondGet Ds Key Default}
    % ... similar to Lookup
end
fun {Domain Ds}
    proc {DomainD Ds ?S1 Sn}
        case Ds
        of leaf then
            S1=Sn
            [] tree(K _ L R) then S2 S3 in
                {DomainD L S1 S2}
                S2=K|S3
                {DomainD R S3 Sn}
        end
    end D
in
    {DomainD Ds D nil} D
end

```

Figure 3.28: Declarative dictionary (with ordered binary tree)

State-based implementation

We can do even better than the tree-based implementation by leaving the declarative model behind and using explicit state (see Section 6.5.1). This gives a stateful dictionary, which is a slightly different type than the declarative dictionary. But it gives the same functionality. Using state is an advantage because it reduces the execution time of Put and CondGet operations to amortized constant time.

3.7.3 A word frequency application

To compare our four dictionary implementations, let us use them in a simple application. Let us write a program to count word frequencies in a string. Later on, we will see how to use this to count words in a file. Figure 3.29 defines the function WordFreq, which is given a list of characters Cs and returns a list of pairs W#N, where W is a word (a maximal sequence of letters and digits) and N is the number of times the word occurs in Cs. The function WordFreq is defined in terms of the following functions:

- {WordChar C} returns true iff C is a letter or digit.
- {WordToAtom PW} converts a reversed list of word characters into an atom containing those characters. The function StringToAtom is used to create the atom.

```

fun {WordChar C}
  (&a=<C andthen C=<&z) orelse
  (&A=<C andthen C=<&Z) orelse (&0=<C andthen C=<&9)
end

fun {WordToAtom PW}
  {StringToAtom {Reverse PW}}
end

fun {IncWord D W}
  {Put D W {CondGet D W 0}+1}
end

fun {CharsToWords PW Cs}
  case Cs
  of nil andthen PW==nil then
    nil
  [] nil then
    [{WordToAtom PW}]
  [] C|Cr andthen {WordChar C} then
    {CharsToWords {Char.toLower C}|PW Cr}
  [] C|Cr andthen PW==nil then
    {CharsToWords nil Cr}
  [] C|Cr then
    {WordToAtom PW}|{CharsToWords nil Cr}
  end
end

fun {CountWords D Ws}
  case Ws
  of W|Wr then {CountWords {IncWord D W} Wr}
  [] nil then D
  end
end

fun {WordFreq Cs}
  {CountWords {NewDictionary} {CharsToWords nil Cs}}
end

```

Figure 3.29: Word frequencies (with declarative dictionary)

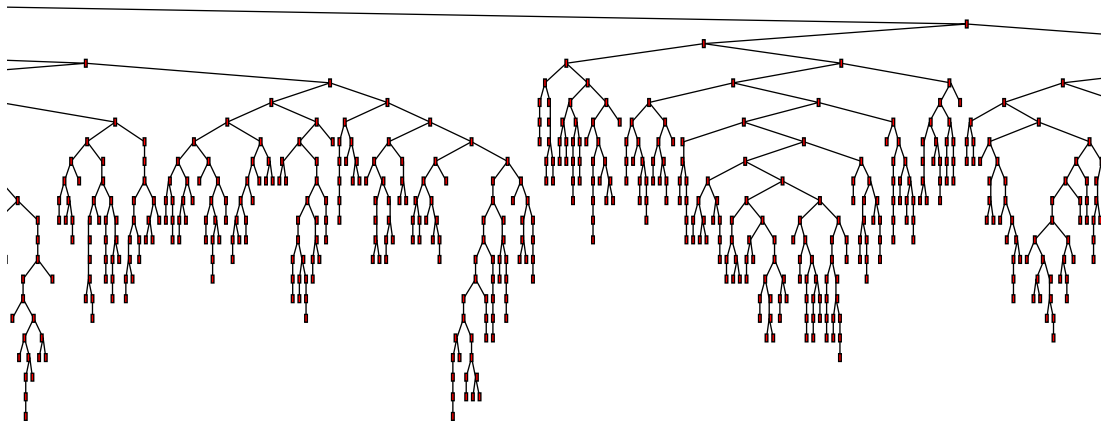


Figure 3.30: Internal structure of binary tree dictionary in WordFreq (in part)

- `{IncWord D W}` takes a dictionary `D` and an atom `W`. Returns a new dictionary in which the `W` field is incremented by 1. Remark how easy this is to write with `CondGet`, which takes care of the case when `W` is not yet in the dictionary.
- `{CharsToWords nil Cs}` takes a list of characters `Cs` and returns a list of atoms, where the characters in each atom's print name form a word in `Cs`. The function `Char.toLower` is used to convert uppercase letters to lowercase, so that "The" and "the" are considered the same word.
- `{CountWords D Ws}` takes an empty dictionary and the output of `CharsToWords`. It returns a dictionary in which each key maps to the number of times the word occurs.

Here is a sample execution. The following input:

```
declare
T="Oh my darling, oh my darling, oh my darling Clementine.
  She is lost and gone forever, oh my darling Clementine."
{Browse {WordFreq T}}
```

displays this word frequency count:

```
[she#1 is#1 clementine#2 lost#1 my#4 darling#4 gone#1 and#1
oh#4 forever#1]
```

We have run `WordFreq` on a more substantial text, namely an early draft of this book. The text contains 712626 characters, giving a total of 110457 words of which 5561 are different. We have run `WordFreq` with three implementations of dictionaries: using lists (see previous example), using binary trees (see Section 3.7.2), and using state (the built-in implementation of dictionaries; see Section 6.8.2). Figure 3.30 shows part of the internal structure of the binary tree

dictionary, drawn with the algorithm of Section 3.4.7. The code we measured is in Section 3.8.1. Running it gives the following times (accurate to 10%):¹⁶

Dictionary implementation	Execution time	Time complexity
Using lists	620 seconds	$O(n)$
Using ordered binary trees	8 seconds	$O(\log n)$
Using state	2 seconds	$O(1)$

The time is the wall-clock time to do everything, i.e., read the text file, run `WordFreq`, and write a file containing the word counts. The difference between the three times is due completely to the different dictionary implementations. Comparing the times gives a good example of the practical effect of using different implementations of an important data type. The complexity shows how the time to insert or look up one item depends on the size of the dictionary.

3.7.4 Secure abstract data types

In both the stack and dictionary data types, the internal representation of values is visible to users of the type. If the users are disciplined programmers then this might not be a problem. But this is not always the case. A user can be tempted to look at a representation or even to construct new values of the representation.

For example, a user of the stack type can use `Length` to see how many elements are on the stack, *if* the stack is implemented as a list. The temptation to do this can be very strong if there is no other way to find out what the size of the stack is. Another temptation is to fiddle with the stack contents. Since any list is also a legal stack value, the user can build new stack values, e.g., by removing or adding elements.

In short, any user can add new stack operations anywhere in the program. This means that the stack's implementation is potentially spread out over the whole program instead of being limited to a small part. This is a disastrous state of affairs, for two reasons:

- The program is much harder to maintain. For example, say we want to improve the efficiency of a dictionary by replacing the list-based implementation by a tree-based implementation. We would have to scour the whole program to find out which parts depend on the list-based implementation. There is also a problem of error confinement: if the program has bugs in one part then this can spill over into the abstract data types, making them buggy as well, which then contaminates other parts of the program.
- The program is susceptible to malicious interference. This is a more subtle problem that has to do with security. It does not occur with programs written by people who trust each other. It occurs rather with open programs.

¹⁶Using Mozart 1.1.0 under Red Hat Linux release 6.1 on a Dell Latitude CPx notebook computer with Pentium III processor at 500 MHz.

An *open program* is one that can interact with other programs that are only known at run-time. What if the other program is malicious and wants to disrupt the execution of the open program? Because of the evolution of the Internet, the proportion of open programs is increasing.

How do we solve these problems? The basic idea is to protect the internal representation of the abstract datatype's values, e.g., the stack values, from unauthorized interference. The value to be protected is put inside a *protection boundary*. There are two ways to use this boundary:

- *Stationary value*. The value never leaves the boundary. A well-defined set of operations can enter the boundary to calculate with the value. The result of the calculation stays inside the boundary.
- *Mobile value*. The value can leave and reenter the boundary. When it is outside, operations can be done on it. Operations with proper authorization can take the value out of the boundary and calculate with it. The result is put back inside the boundary.

With either of these solutions, reasoning about the type's implementation is much simplified. Instead of looking at the whole program, we need only look at how the type's operations are implemented.

The first solution is like computerized banking. Each client has an account with some amount of money. A client can do a transaction that transfers money from his or her account to another account. But since clients never actually go to the bank, the money never actually leaves the bank. The second solution is like a safe. It stores money and can be opened by clients who have the key. Each client can take money out of the safe or put money in. Once out, the client can give the money to another client. But when the money is in the safe, it is safe.

In the next section we build a secure ADT using the second solution. This way is the easiest to understand for the declarative model. The authorization we need to enter the protection boundary is a kind of "key". We add it as a new concept to the declarative model, called *name*. Section 3.7.7 then explains that a key is an example of a very general security idea, called a *capability*. In Chapter 6, Section 6.4 completes the story on secure ADTs by showing how to implement the first solution and by explaining the effect of explicit state on security.

3.7.5 The declarative model with secure types

The declarative model defined so far does not let us construct a protection boundary. To do it, we need to extend the model. We need two extensions, one to protect values and one to protect unbound variables. Table 3.6 shows the resulting kernel language with its two new operations. We now explain these two operations.

$\langle s \rangle ::=$	
skip	Empty statement
$\langle s \rangle_1 \langle s \rangle_2$	Statement sequence
local $\langle x \rangle$ in $\langle s \rangle$ end	Variable creation
$\langle x \rangle_1 = \langle x \rangle_2$	Variable-variable binding
$\langle x \rangle = \langle v \rangle$	Value creation
if $\langle x \rangle$ then $\langle s \rangle_1$ else $\langle s \rangle_2$ end	Conditional
case $\langle x \rangle$ of $\langle \text{pattern} \rangle$ then $\langle s \rangle_1$ else $\langle s \rangle_2$ end	Pattern matching
$\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$	Procedure application
try $\langle s \rangle_1$ catch $\langle x \rangle$ then $\langle s \rangle_2$ end	Exception context
raise $\langle x \rangle$ end	Raise exception
$\{ \text{NewName } \langle x \rangle \}$	Name creation
$\langle y \rangle = ! ! \langle x \rangle$	Read-only view

Table 3.6: The declarative kernel language with secure types

Protecting values

One way to make values secure is by adding a “wrapping” operation with a “key”. That is, the internal representation is put inside a data structure that is inaccessible except to those that know a special value, the key. Knowing the key allows to create new wrappings and to look inside existing wrappings made with the same key.

We implement this with a new basic type called a *name*. A name is a constant like an atom except that it has a much more restricted set of operations. In particular, names do not have a textual representation: they cannot be printed or typed in at the keyboard. Unlike for atoms, it is not possible to convert between names and strings. The only way to know a name is by being passed a reference to it within a program. The name type comes with just two operations:

Operation	Description
$\{ \text{NewName} \}$	Return a fresh name
$N1 = N2$	Compare names $N1$ and $N2$

A *fresh* name is one that is guaranteed to be different from all other names in the system. Alert readers will notice that `NewName` is not declarative because calling it twice returns different results. In fact, the creation of fresh names is a stateful operation. The guarantee of uniqueness means that `NewName` has some internal memory. However, if we use `NewName` just for making declarative ADTs secure then this is not a problem. The resulting secure ADT is still declarative.

To make a data type secure, it suffices to put it inside a function that has an external reference to the name. For example, take the value `S`:

```
S=[a b c]
```

This value is an internal state of the stack type we defined before. We can make it secure as follows:

```
Key={NewName}
SS=fun {$ K} if K==Key then S end end
```

This first creates a new name in `Key`. Then it makes a function that can return `S`, but only if the correct argument is given. We say that this “wraps” the value `S` inside `SS`. If one knows `Key`, then accessing `S` from `SS` is easy:

```
S={SS Key}
```

We say this “unwraps” the value `S` from `SS`. If one does not know `Key`, unwrapping is impossible. There is no way to know `Key` except for being passed it explicitly in the program. Calling `SS` with a wrong argument will simply raise an exception.

A wrapper

We can define an abstract data type to do the wrapping and unwrapping. The type defines two operations, `Wrap` and `Unwrap`. `Wrap` takes any value and returns a protected value. `Unwrap` takes any protected value and returns the original value. The `Wrap` and `Unwrap` operations come in pairs. The only way to unwrap a wrapped value is by using the corresponding unwrap operation. With names we can define a procedure `NewWrapper` that returns new `Wrap/Unwrap` pairs:

```
proc {NewWrapper ?Wrap ?Unwrap}
  Key={NewName}
in
  fun {Wrap X}
    fun {$ K} if K==Key then X end end
  end
  fun {Unwrap W}
    {W Key}
  end
end
```

For maximum protection, each abstract data type can use its own `Wrap/Unwrap` pair. Then they are protected from each other as well as from the main program. Given the value `S` as before:

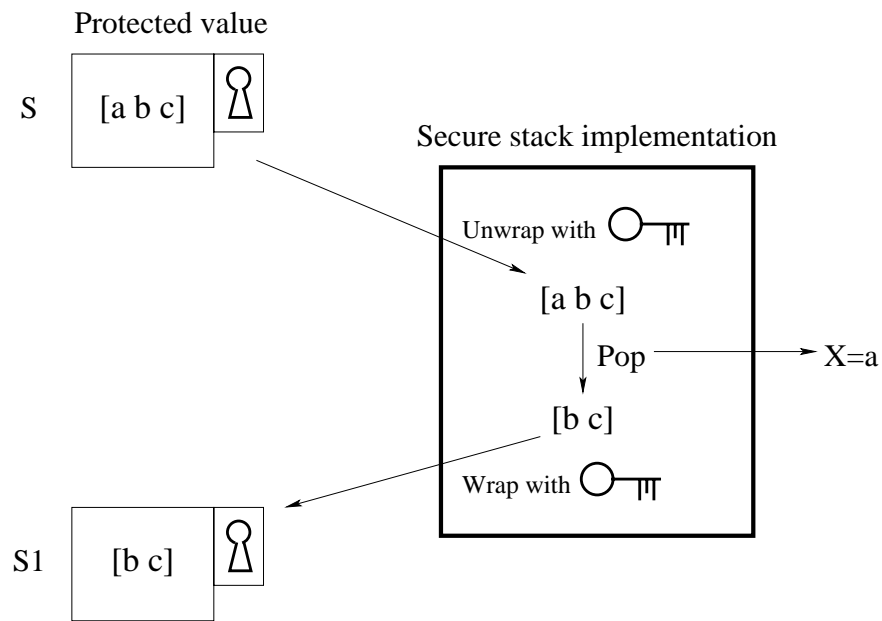
```
S=[a b c]
```

we protect it as follows:

```
SS={Wrap S}
```

We can get the original value back as follows:

```
S={Unwrap SS}
```

Figure 3.31: Doing `S1={Pop S X}` with a secure stack

A secure stack

Now we can make the stack secure. The idea is to unwrap incoming values and wrap outgoing values. To perform a legal operation on a secure type value, the routine unwraps the secure value, performs the intended operation to get a new value, and then wraps the new value to guarantee security. This gives the following implementation:

```

local Wrap Unwrap in
  {NewWrapper Wrap Unwrap}
  fun {NewStack} {Wrap nil} end
  fun {Push S E} {Wrap E|{Unwrap S}} end
  fun {Pop S E}
    case {Unwrap S} of X|S1 then E=X {Wrap S1} end
  end
  fun {IsEmpty S} {Unwrap S}==nil end
end

```

Figure 3.31 illustrates the `Pop` operation. The box with keyhole represents a protected value. The key represents the name, which is used internally by `Wrap` and `Unwrap` to lock and unlock a box. Lexical scoping guarantees that wrapping and unwrapping are only possible inside the stack implementation. Namely, the identifiers `Wrap` and `Unwrap` are only visible inside the `local` statement. Outside this scope, they are hidden. Because `Unwrap` is hidden, there is absolutely no way to see inside a stack value. Because `wrap` is hidden, there is absolutely no way to “forge” stack values.

Protecting unbound variables

Sometimes it is useful for a data type to output an unbound variable. For example, a stream is a list with an unbound tail. We would like anyone to be able to read the stream but only the data type implementation to be able to extend it. Using standard unbound variables this does not work, for example:

$$S = a | b | c | x$$

The variable x is not secure since anyone who knows S can bind x .

The problem is that anyone who has a reference to an unbound variable can bind the variable. One solution is to have a restricted version of the variable that can only be read, not bound. We call this a *read-only view* of a variable. We extend the declarative model with one function:

Operation	Description
$!x$	Return a read-only view of x

Any attempt to bind a read-only view will block. Any binding of x will be transferred to the read-only view. To protect a stream, its tail should be a read-only view.

In the abstract machine, read-only views sit in a new store called the *read-only store*. We modify the bind operation so that before binding a variable to a determined value, it checks whether the variable is in the read-only store. If so, the bind suspends. When the variable becomes determined, then the bind operation can continue.

Creating fresh names

To conclude this section, let us see how to create fresh names in the implementation of the declarative model. How can we guarantee that a name is globally unique? This is easy for programs running in one process: names can be implemented as successive integers. But this approach fails miserably for open programs. For them, *globally* potentially means among all running programs in all the world's computers. There are basically two approaches to create names that are globally unique:

- *The centralized approach.* There is a name factory somewhere in the world. To get a fresh name, you need to send a message to this factory and the reply contains a fresh name. The name factory does not have to be physically in one place; it can be spread out over many computers. For example, the IP protocol supposes a unique IP address for every computer in the world that is connected to the Internet. IP addresses can change over time, though, e.g., if network address translation is done or dynamic allocation of IP addresses is done using the DHCP protocol. We therefore complement the IP address with a high-resolution timestamp giving the creation time of `NewName`. This gives a unique constant that can be used to implement a local name factory on each computer.

- *The decentralized approach.* A fresh name is just a vector of random bits. The random bits are generated by an algorithm that depends on enough external information so that different computers will not generate the same vector. If the vector is long enough, then the that names are not unique will be arbitrarily small. Theoretically, the probability is always nonzero, but in practice this technique works well.

Now that we have a unique name, how do we make sure that it is unforgeable? This requires cryptographic techniques that are beyond the scope of this book [166].

3.7.6 A secure declarative dictionary

Now let us see how to make the declarative dictionary secure. It is quite easy. We can use the same technique as for the stack, namely by using a wrapper and an unwrapper. Here is the new definition:

```
local
  Wrap Unwrap
  {NewWrapper Wrap Unwrap}
  % Previous definitions:
  fun {NewDictionary2} ... end
  fun {Put2 Ds K Value} ... end
  fun {CondGet2 Ds K Default} ... end
  fun {Domain2 Ds} ... end
in
  fun {NewDictionary}
    {Wrap {NewDictionary2}}
  end
  fun {Put Ds K Value}
    {Wrap {Put2 {Unwrap Ds} K Value}}
  end
  fun {CondGet Ds K Default}
    {CondGet2 {Unwrap Ds} K Default}
  end
  fun {Domain Ds}
    {Domain2 {Unwrap Ds}}
  end
end
```

Because `Wrap` and `Unwrap` are only known inside the scope of the `local`, the wrapped dictionary cannot be unwrapped by anyone outside of this scope. This technique works for both the list and tree implementations of dictionaries.

3.7.7 Capabilities and security

We say a computation is *secure* if it has well-defined and controllable properties, independent of the existence of other (possibly malicious) entities (either

computations or humans) in the system [4]. We call these entities “adversaries”. Security allows to protect both from malicious computations and innocent (but buggy) computations. The property of being secure is global; “cracks” in a system can occur at any level, from the hardware to software to the human organization housing the system. Making a computer system secure involves not only computer science but also many aspects of human society [5].

A short, precise, and concrete description of how the system will ensure its security is called its *security policy*. Designing, implementing, and verifying security policies is crucial for building secure systems, but is outside the scope of this book.

In this section, we consider only a small part of the vast discipline of security, namely the programming language viewpoint. To implement a security policy, a system uses *security mechanisms*. Throughout this book, we will discuss security mechanisms that are part of a programming language, such as lexical scoping and names. We will ask ourselves what properties a language must possess in order to build secure programs, that is, programs that can resist attacks by adversaries that stay within the language.¹⁷ We call such a language a *secure language*. Having a secure language is an important requirement for building secure computer programs. Designing and implementing a secure language is an important topic in programming language research. It involves both semantic properties and properties of the implementation.

Capabilities

The protection techniques we have introduced to make secure abstract data types are special cases of a security concept called a *capability*. Capabilities are at the heart of modern research on secure languages. For example the secure language E hardens references to language entities so that they behave as capabilities [123, 183]. The *wrap/unwrap* pairs we introduced previously are called *sealer/unsealer* pairs in E. Instead of using external references to protect values, sealer/unsealer pairs encrypt and decrypt the values. In this view, the name is used as an encryption and decryption key.

The capability concept was invented in the 1960’s, in the context of operating system design. Operating systems have always had to protect users from each other while still allowing them do their work. Since this early work, it has become clear that the concept belongs in the programming language and is generally useful for building secure programs [124]. Capabilities can be defined in many ways, but the following definition is reasonable for a programming language. A *capability* is an unforgeable language entity that gives its owner the right to perform a given set of actions. The set of actions is defined inside the capability and may change over time. By *unforgeable* we mean that it is not possible for any implementation, even one that is intimately connected to the hardware architec-

¹⁷Staying withing the language can be guaranteed by always running programs within a virtual machine that accepts only binaries of legal programs.

ture such as one in assembly language, to create a capability. In the E literature this property is summarized by the phrase “connectivity begets connectivity”: the only way to get a new capability is by being passed it explicitly through an existing capability [125].

All values of data types are capabilities in this sense, since they give their owners the ability to do all operations of that type, but no more. An *owner* of a language entity is any program fragment that references that entity. For example, a record R gives its owner the ability to do many operations including field selection $R.F$ and arity $\{\text{Arity } R\}$. A procedure P gives its owner the ability to call P . A name gives its owner the ability to compare its value with other values. An unbound variable gives its owner the ability to bind it and to read its value. A read-only variable gives its owner the ability to read its value, but not to bind it.

New capabilities can be defined during a program’s execution as instances of ADTs. For the models of this book, the simplest way is to use procedure values. A reference to a procedure value gives its owner the right to call the procedure, i.e., to do whatever action the procedure was designed to do. Furthermore, a procedure reference cannot be forged. In a program, the only way to know the reference is if it is passed explicitly. The procedure can hide all its sensitive information in its external references. For this to work, the language must guarantee that knowing a procedure does not automatically give one the right to examine the procedure’s external references!

Principle of least privilege

An important design principle for secure systems is the principle of least privilege: each entity should be given the least authority (or “privilege”) that is necessary for it to get its job done. This is also called the principle of least authority (POLA) or the “need to know” principle. Determining exactly what the least authority is in all cases is an undecidable problem: there cannot exist an algorithm to solve it in all cases. This is because the authority depends on what the entity does during its execution. If we would have an algorithm, it would be powerful enough to solve the Halting Problem, which has been proved not to have a solution.

In practice, we do not need to know the exact least authority. Sufficient security can be achieved with approximations to it. The programming language should make it easy to do these approximations. Capabilities, as we defined them above, have this ability. With them, it is easy to make the approximation as precise as is needed. For example, an entity can be given the authority to create a file with a given name and maximum size in a given directory. For files, coarser granularities are usually enough, such as the authority to create a file in a given directory. Capabilities can handle both the fine and coarse-grained cases easily.

Capabilities and explicit state

Declarative capabilities, i.e., capabilities written in a declarative computation model, lack one crucial property to make them useful in practice. The set of actions they authorize cannot be changed over time. In particular, none of their actions can be revoked. To make a capability revocable, the computation model needs an additional concept, namely explicit state. This is explained in Section 6.4.3.

3.8 Nondeclarative needs

Declarative programming, because of its “pure functional” view of programming, is somewhat detached from the real world, in which entities have memories (state) and can evolve independently and proactively (concurrency). To connect a declarative program to the real world, some nondeclarative operations are needed. This section talks about two classes of such operations: file I/O (input/output) and graphical user interfaces. A third class of operations, standalone compilation, is given in Section 3.9.

Later on we will see that the nondeclarative operations of this section fit into more general computation models than the declarative one, in particular stateful and concurrent models. In a general sense, this section ties in with the discussion on the limits of declarative programming in Section 4.7. Some of the operations manipulate state that is external to the program; this is just a special case of the system decomposition principle explained in Section 6.7.2.

The new operations introduced by this section are collected in *modules*. A module is simply a record that groups together related operations. For example, the module `List` groups many list operations, such as `List.append` and `List.member` (which can also be referenced as `Append` and `Member`). This section introduces the three modules `File` (for file I/O of text), `QtK` (for graphical user interfaces), and `Pickle` (for file I/O of any values). Some of these modules (like `Pickle`) are immediately known by Mozart when it starts up. The other modules can be loaded by calling `Module.link`. In what follows, we show how to do this for `File` and `QtK`. More information about modules and how to use them is given later, in Section 3.9.

3.8.1 Text input/output with a file

A simple way to interface declarative programming with the real world is by using files. A *file* is a sequence of values that is stored external to the program on a permanent storage medium such as a hard disk. A *text file* is a sequence of characters. In this section, we show how to read and write text files. This is enough for using declarative programs in a practical way. The basic pattern of access is simple:

Input file $\xrightarrow{\text{read}}$ compute function $\xrightarrow{\text{write}}$ output file

We use the module `File`, which can be found on the book's Web site. Later on we will do more sophisticated file operations, but this is enough for now.

Loading the module `File`

The first step is to load the module `File` into the system, as explained in Appendix A.1.2. We assume that you have a compiled version of the module `File`, in the file `File.ozf`. Then execute the following:

```
declare [File]={Module.link ['File.ozf']}
```

This calls `Module.link` with a list of paths to compiled modules. Here there is just one. The module is loaded, linked it into the system, initialized, and bound to `File`.¹⁸ Now we are ready to do file operations.

Reading a file

The operation `File.readList` reads the whole content of the file into a string:

```
L={File.readList "foo.txt"}
```

This example reads the file `foo.txt` into `L`. We can also write this as:

```
L={File.readList `foo.txt`}
```

Remember that `"foo.txt"` is a string (a list of character codes) and ``foo.txt`` is an atom (a constant with a print representation). The file name can be represented in both ways. There is a third way to represent file names: as virtual strings. A *virtual string* is a tuple with label ``#`` that represents a string. We could therefore just as well have entered the following:

```
L={File.readList foo#`.`#txt}
```

The tuple `foo#`.`#txt`, which we can also write as ``#`(foo `.` txt)`, represents the string `"foo.txt"`. Using virtual strings avoids the need to do explicit string concatenations. All Mozart built-in operations that expect strings will work also with virtual strings. All three ways of loading `foo.txt` have the same effect. They bind `L` to a list of the character codes in the file `foo.txt`.

Files can also be referred to by *URL*. A URL gives a convenient global address for files since it is widely supported through the World-Wide Web infrastructure. It is just as easy to read a file through its URL as through its file name:

```
L={File.readList `http://www.mozart-oz.org/features.html`}
```

That's all there is to it. URLs can only be used to read files, but not to write files. This is because URLs are handled by Web servers, which are usually set up to allow only reading.

¹⁸To be precise, the module is loaded *lazily*: it will only actually be loaded the first time that we use it.

Mozart has other operations that allow to read a file either incrementally or lazily, instead of all at once. This is important for very large files that do not fit into the memory space of the Mozart process. To keep things simple for now, we recommend that you read files all at once. Later on we will see how to read a file incrementally.

Writing a file

Writing a file is usually done incrementally, by appending one string at a time to the file. The module `File` provides three operations: `File.writeOpen` to open the file, which must be done first, `File.write` to append a string to the file, and `File.writeClose` to close the file, which must be done last. Here is an example:

```
{File.writeOpen 'foo.txt'}
{File.write 'This comes in the file.\n'}
{File.write 'The result of 43*43 is '#43*43#'\n'}
{File.write "Strings are ok too.\n"}
{File.writeClose}
```

After these operations, the file 'foo.txt' has three lines of text, as follows:

```
This comes in the file.
The result of 43*43 is 1849.
Strings are ok too.
```

Example execution

In Section 3.7.3 we defined the function `WordFreq` that calculates the word frequencies in a string. We can use this function to calculate word frequencies and store them in a file:

```
% 1. Read input file
L={File.readList 'book.raw'}
% 2. Compute function
D={WordFreq L}
% 3. Write output file
{File.writeOpen 'word.freq'}
for X in {Domain D} do
    {File.write {Get D X}# occurrences of word '#X#\n'}
end
{File.writeClose}
```

Section 3.7.3 gives some timing figures of this code using different dictionary implementations.

3.8.2 Text input/output with a graphical user interface

The most direct way to interface programs with a human user is through a graphical user interface. This section shows a simple yet powerful way to define graphical user interfaces, namely by means of concise, mostly declarative specifications. This is an excellent example of a *descriptive* declarative language, as explained in Section 3.1. The descriptive language is recognized by the `QTK` module of the Mozart system. The user interface is specified as a nested record, supplemented with objects and procedures. (Objects are introduced in Chapter 7. For now, you can consider them as procedures with internal state, like the examples of Chapter 1.)

This section shows how to build user interfaces to input and output textual data to a window. This is enough for many declarative programs. We give a brief overview of the `QTK` module, just enough to build these user interfaces. Later on we will build more sophisticated graphical user interfaces. Chapter 10 gives a fuller discussion of declarative user interface programming in general and of its realization in `QTK`.

Declarative specification of widgets

A window on the screen consists of a set of widgets. A *widget* is a rectangular area in the window that has a particular interactive behavior. For example, some widgets can display text or graphic information, and other widgets can accept user interaction such as keyboard input and mouse clicks. We specify each widget declaratively with a record whose label and features define the widget type and initial state. We specify the window declaratively as a nested record (i.e., a tree) that defines the logical structure of the widgets in the window. Here are the five widgets we will use for now:

- The `label` widget can display a text. The widget is specified by the record:

```
label(text:VS)
```

where `VS` is a virtual string.

- The `text` widget is used to display and enter large quantities of text. It can use scrollbars to display more text than can fit on screen. With a vertical (i.e., top-down) scrollbar, the widget is specified by the record:

```
text(handle:H tds scrollbar:true)
```

When the window is created, the variable `H` will be bound to an object used to control the widget. We call such an object a *handler*. You can consider the object as a one-argument procedure: `{H set(VS)}` displays a text and `{H get(VS)}` reads the text.

- The `button` widget specifies a button and an action to execute when the button is pressed. The widget is specified by the record:

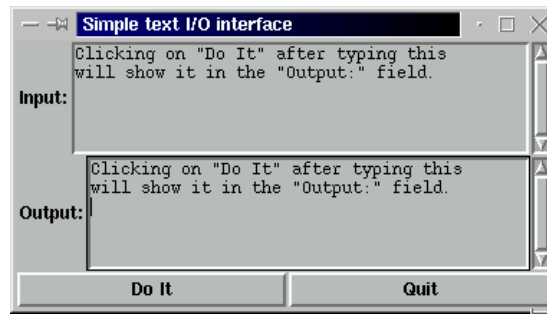


Figure 3.32: A simple graphical I/O interface for text

```
button(text:VS action:P)
```

where *VS* is a virtual string and *P* is a zero-argument procedure. $\{P\}$ is called whenever the button is pressed.¹⁹ For each window, all its actions are executed sequentially.

- The *td* (top-down) and *lr* (left-right) widgets specify an arrangement of other widgets in top-down or left-right order:

```
lr(W1 W2 ... Wn)
td(W1 W2 ... Wn)
```

where *W1*, *W2*, ..., *Wn* are other widget specifications.

Declarative specification of resize behavior

When a window is resized, the widgets inside should behave properly, i.e., either changing size or staying the same size, depending on what the interface should do. We specify each widget's resize behavior declaratively, by means of an optional *glue* feature in the widget's record. The *glue* feature indicates whether the widget's borders should or should not be "glued" to its enclosing widget. The *glue* feature's argument is an atom consisting of any combination of the four characters *n* (north), *s* (south), *w* (west), *e* (east), indicating for each direction whether the border should be glued or not. Here are some examples:

- *No glue*. The widget keeps its natural size and is centered in the space allotted to it, both horizontally and vertically.
- *glue:nswe* glues to all four borders, stretching to fit both horizontally and vertically.
- *glue:we* glues horizontally left and right, stretching to fit. Vertically, the widget is not stretched but centered in the space allotted to it.

¹⁹To be precise, whenever the left mouse button is both clicked and released while the mouse is over the button. This allows the user to correct any mistaken click on the button.

- `glue:w` glues to the left edge and does not stretch.
- `glue:wns` glues vertically top and bottom, stretching to fit vertically, and glues to the left edge, not stretching horizontally.

Loading the module QTk

The first step is to load the QTk module into the system. Since QTk is part of the Mozart Standard Library, it suffices to give the right path name:

```
declare [QTk]={Module.link [ `x-oz://system/wp/QTk.ozf` ]}
```

Now that QTk is loaded, we can use it to build interfaces according to the specifications of the previous section.

Building the interface

The QTk module has a function `QTk.build` that takes an interface specification, which is just a nested record of widgets, and builds a window containing these widgets. Let us build a simple interface with one button that displays ouch in the browser whenever the button is clicked:

```
D=td(button(text:"Press me"
           action:proc {$} {Browse ouch} end))
W={QTk.build D}
{W show}
```

The record `D` always has to start with `td` or `lr`, even if the window has just one widget. `QTk.build` returns an object `W` that represents the window. The window starts out being hidden. It can be displayed or hidden again by calling `{W show}` or `{W hide}`. Here is a bigger example that implements a complete text I/O interface:

```
declare
In Out
A1=proc {$} X in {In get(X)} {Out set(X)} end
A2=proc {$} {W close} end
D=td(title:"Simple text I/O interface"
     lr(label(text:"Input:")
         text(handle:In  tdscrollbar:true glue:nswe)
         glue:nswe)
     lr(label(text:"Output:")
         text(handle:Out tdscrollbar:true glue:nswe)
         glue:nswe)
     lr(button(text:"Do It" action:A1 glue:nswe)
         button(text:"Quit" action:A2 glue:nswe)
         glue:we))
W={QTk.build D}
{W show}
```

At first glance, this may seem complicated, but look again: there are six widgets (two `label`, two `text`, two `button`) arranged with `td` and `lr` widgets. The `Qtk.build` function takes the description `D`. It builds the window of Figure 3.32 and creates the handler objects `In` and `Out`. Compare the record `D` with Figure 3.32 to see how they correspond.

There are two action procedures, `A1` and `A2`, one for each button. The action `A1` is attached to the “Do It” button. Clicking on the button calls `A1`, which transfers text from the first text widget to the second text widget. This works as follows. The call `{In get(X)}` gets the text of the first text widget and binds it to `X`. Then `{Out set(X)}` sets the text in the second text widget to `X`. The action `A2` is attached to the “Quit” button. It calls `{W close}`, which closes the window permanently.

Putting `nswe` glue almost everywhere allows the window to behavior properly when resized. The `lr` widget with the two buttons has `we` glue only, so that the buttons do not expand vertically. The `label` widgets have no glue, so they have fixed sizes. The `td` widget at the top level needs no glue since we assume it is always glued to its window.

3.8.3 Stateless data I/O with files

Input/output of a string is simple, since a string consists of characters that can be stored directly in a file. What about other values? It would be a great help to the programmer if it would be possible to save *any* value to a file and to load it back later. The System module `Pickle` provides exactly this ability. It can save and load any complete value:

```
{Pickle.save X FN}      % Save X in file FN
{Pickle.load FNURL ?X} % Load X from file (or URL) FNURL
```

All data structures used in declarative programming can be saved and loaded except for those containing unbound variables. For example, consider this program fragment:

```
declare
fun {Fact N}
  if N==0 then 1 else N*{Fact N-1} end
end
F100={Fact 100}
F100Gen1=fun {$} F100 end
F100Gen2=fun {$} {Fact 100} end
FNGen1=fun {$ N} F={Fact N} in fun {$} F end end
FNGen2=fun {$ N} fun {$} {Fact N} end end
```

`F100` is a (rather big) integer; the four other entities are functions. The following operation saves the four functions to a file:

```
{Pickle.save [F100Gen1 F100Gen2 FNGen1 FNGen2] `factfile`}
```

To be precise, this saves a value consisting of list of four elements in the file `factfile`. In this example, all elements are functions. The functions have been chosen to illustrate various degrees of delayed calculation. The first two return the result of calculating 100!. The first, `F100Gen1`, knows the integer and returns it directly, and the second, `F100Gen2`, calculates the value each time it is called. The third and fourth, when called with an integer argument n , return a function that when itself called, returns $n!$. The third, `FNGen1`, calculates $n!$ when called, so the returned function just returns a known integer. The fourth, `FNGen2`, does no calculation but lets the returned function calculate $n!$ when called.

To use the contents of `factfile`, it must first be loaded:

```
declare
  [F1 F2 F3 F4]={Pickle.load `factfile`}
in
  {Browse {F1}}
  {Browse {F2}}
  {Browse {{F3 100}}}}
  {Browse {{F4 100}}}}
```

This displays 100! four times. Of course, the following is also possible:

```
declare F1 F2 F3 F4 in
  {Browse {F1}}
  {Browse {F2}}
  {Browse {{F3 100}}}}
  {Browse {{F4 100}}}}
[F1 F2 F3 F4]={Pickle.load `factfile`}
```

After the file is loaded, this displays exactly the same as before. This illustrates yet again how dataflow makes it possible to use a variable before binding it.

We emphasize that the loaded value is *exactly* the same as the one that was saved. There is no difference at all between them. This is true for all possible values: numbers, records, procedures, names, atoms, lists, and so on, including other values that we will see later on in the book. Executing this on one process:

```
... % First statement (defines X)
{Pickle.save X `myfile`}
```

and then this on a second process:

```
X={Pickle.load `myfile`}
... % Second statement (uses X)
```

is rigorously identical to executing the following on a third process:

```
... % First statement (defines X)
{Pickle.save X `myfile`}
_={Pickle.load `myfile`}
... % Second statement (uses X)
```

If the calls to `Pickle` are removed, like this:

```
... % First statement (defines X)
... % Second statement (uses X)
```

then there are two minor differences:

- The first case creates and reads the file ‘myfile’. The second case does not.
- The first case raises an exception if there was a problem in creating or reading the file.

3.9 Program design in the small

Now that we have seen many programming techniques, the next logical step is to use them to solve problems. This step is called *program design*. It starts from a problem we want to solve (usually explained in words, sometimes not very precisely) gives the high-level structure of the program, i.e., what programming techniques we need to use and how they are connected together, and ends up with a complete program that solves the problem.

For program design, there is an important distinction between “programming in the small” and “programming in the large”. We will call the resulting programs “small programs” and “large programs”. The distinction has nothing to do with the program’s size, but rather with how many people were involved in its development. Small programs are written by one person over a short period of time. Large programs are written by more than one person or over a long period of time. The same person now and one year from now should be considered as two people, since the person will forget many details over a year. This section gives an introduction to programming in the small; we leave programming in the large to Section 6.7.

3.9.1 Design methodology

Assume we have a problem that can be solved by writing a small program. Let us see how to design the program. We recommend the following design methodology, which is a mixture of creativity and rigorous thinking:

- **Informal specification.** We start by writing down as precisely as we can what the program should do: what its inputs and outputs are and how the outputs relate to the inputs. This description is called an *informal specification*. Even though it is precise, we call it “informal” because it is written in English. “Formal” specifications are written in a mathematical notation.
- **Examples.** To make the specification perfectly clear, it is always a good idea to imagine examples of what the program does in particular cases. The

examples should “stress” the program: use it in boundary conditions and in the most unexpected ways we can imagine.

- **Exploration.** To find out what programming techniques we will need, a good way is to use the interactive interface to experiment with program fragments. The idea is to write small operations that we think might be needed for the program. We use the operations that the system already provides as a basis. This step gives us a clearer view of what the program’s structure should be.
- **Structure and coding.** Now we can lay out the program’s structure. We make a rough outline of the operations needed to calculate the outputs from the inputs and how they fit together. We then fill in the blanks by writing the actual program code. The operations should be simple: each operation should do just one thing. To improve the structure we can group related operations in modules.
- **Testing and reasoning.** Finally, we have to verify that our program does the right thing. We try it on a series of test cases, including the examples we came up with before. We correct errors until the program works well. We can also reason about the program and its complexity, using the formal semantics for parts that are not clear. Testing and reasoning are complementary: it is important to do both to get a high-quality program.

These steps are not meant to be obligatory, but rather to serve as inspiration. Feel free to adapt them to your own circumstances. For example, when imagining examples it can be clear that the specification has to be changed. However, take care never to forget the most important step, which is *testing*.

3.9.2 Example of program design

To illustrate these steps, let us retrace the development of the word frequency application of Section 3.7.3. Here is a first attempt at an informal specification:

Given a file name, the application opens a window and displays a list of pairs, where each pair consists of a word and an integer giving the number of times the word occurs in the file.

Is this specification precise enough? What about a file containing a word that is not valid English or a file containing non-Ascii characters? Our specification is not precise enough: it does not define what a “word” is. To make it more precise we have to know the purpose of the application. Say that we just want to get a general idea of word frequencies, independent of any particular language. Then we can define a word simply as:

A “word” is a maximal contiguous sequence of letters and digits.

This means that words are separated by at least one character that is not a letter or a digit. This accepts a word that is not valid English but does not accept words containing non-Ascii characters. Is this good enough? What about words with a hyphen (such as “true-blue”) or idiomatic expressions that act as units (such as “trial and error”)? In the interest of simplicity, let us reject these for now. But we may have to change the specification later to accept them, depending on how we use the word frequency application.

Now we have arrived at our specification. Note the essential role played by *examples*. They are important signposts on the way to a precise specification. The examples were expressly designed to test the limits of the specification.

The next step is to design the program’s structure. The appropriate structure seems to be a pipeline: first read the file into a list of characters and then convert the list of characters into a list of words, where a word is represented as a character string. To count the words we need a data structure that is indexed by words. The declarative dictionary of Section 3.7.2 would be ideal, but it is indexed by atoms. Luckily, there is an operation to convert character strings to atoms: `StringToAtom` (see Appendix B). With this we can write our program. Figure 3.29 gives the heart: a function `WordFreq` that takes a list of characters and returns a dictionary. We can test this code on various examples, and especially on the examples we used to write the specification. To this we will add the code to read the file and display the output in a window; for this we use the file operations and graphical user interface operations of Section 3.8. It is important to package the application cleanly, as a software component. This is explained in the next two sections.

3.9.3 Software components

What is a good way to organize a program? One could write the program as one big monolithic whole, but this can be confusing. A better way is to partition the program into logical units, each of which implements a set of operations that are related in some way. Each logical unit has two parts, an *interface* and an *implementation*. Only the interface is visible from outside the logical unit. A logical unit may use others as part of its implementation.

A program is then simply a directed graph of logical units, where an edge between two logical units means that the first needs the second for its implementation. Popular usage calls these logical units “modules” or “components”, without defining precisely what these words mean. This section introduces the basic concepts, defines them precisely, and shows how they can be used to help design small declarative programs. Section 6.7 explains how these ideas can be used to help design large programs.

```

<statement> ::= functor <variable>
               [ import { <variable> [ at <atom> ]
                       | <variable> ^ ( ^ { (<atom> | <int>) [ ^ : ^ <variable> ] } + ^ ) ^
                       } + ]
               [ export { [ (<atom> | <int>) ^ : ^ ] <variable> } + ]
               define { <declarationPart> } + [ in <statement> ] end
               | ...

```

Table 3.7: Functor syntax

Modules and functors

We call *module* a part of a program that groups together related operations into an entity that has an interface and an implementation. In this book, we will implement modules in a simple way:

- The module’s *interface* is a record that groups together related language entities (usually procedures, but anything is allowed including classes, objects, etc.).
- The module’s *implementation* is a set of language entities that are accessible by the interface operations but hidden from the outside. The implementation is hidden using lexical scoping.

We will consider module specifications as entities separate from modules. A *module specification* is a kind of template that creates a module each time it is instantiated. A module specification is sometimes called a *software component*. Unfortunately, the term “software component” is widely used with many different meanings [187]. To avoid any confusion in this book, we will call our module specifications *functors*. A functor is a function whose arguments are the modules it needs and whose result is a new module. (To be precise, the functor takes module interfaces as arguments, creates a new module, and returns that module’s interface!) Because of the functor’s role in structuring programs, we provide it as a linguistic abstraction. A functor has three parts: an **import** part, which specifies what other modules it needs, an **export** part, which specifies the module interface, and a **define** parts, which gives the module implementation including its initialization code. The syntax for functor declarations allows to use them as either statements or expressions, like the syntax for procedures. Table 3.7 gives the syntax of functor declarations as statements.

In the terminology of software engineering, a software component is a unit of independent deployment, a unit of third-party development, and has no persistent state (following the definition given in [187]). Functors satisfy this definition and are therefore a kind of software component. With this terminology, a module is a *component instance*; it is the result of installing a functor in a particular module

environment. The module environment consists of a set of modules, each of which may have an execution state.

Functors in the Mozart system are compilation units. That is, the system has support for handling functors in files, both as *source code* (i.e., human-readable text) and *object code* (i.e., compiled form). Source code can be *compiled*, or translated, into object code. This makes it easy to use functors to exchange software between developers. For example, the Mozart system has a library, called *MOGUL* (for *Mozart Global User Library*), in which third-party developers can put any kind of information. Usually, they put in functors and applications.

An application is *standalone* if it can be run without the interactive interface. It consists of a main functor, which is evaluated when the program starts. It imports the modules it needs, which causes other functors to be evaluated. The main functor is used for its effect of starting the application and not for its resulting module, which is silently ignored. Evaluating, or “installing”, a functor creates a new module in three steps. First, the modules it needs are identified. Second, the initialization code is executed. Third, the module is loaded the first time it is needed during execution. This technique is called *dynamic linking*, as opposed to *static linking*, in which the modules are loaded when execution starts. At any time, the set of currently installed modules is called the *module environment*.

Implementing modules and functors

Let us see how to construct software components in steps. First we give an example module. Then we show how to convert this module into a software component. Finally, we turn it into a linguistic abstraction.

Example module In general a module is a record, and its interface is accessed through the record’s fields. We construct a module called `MyList` that provides interface procedures for appending, sorting, and testing membership of lists. This can be written as follows:

```
declare MyList in
  local
    proc {Append ... } ... end
    proc {MergeSort ...} ... end
    proc {Sort ... } ... {MergeSort ...} ... end
    proc {Member ...} ... end
  in
    MyList = `export` (append: Append
                      sort: Sort
                      member: Member
                      ...)
  end
```

The procedure `MergeSort` is inaccessible outside of the `local` statement. The other procedures cannot be accessed directly, but only through the fields of the `MyList` module, which is a record. For example, `Append` is accessible as `MyList.append`. Most of the library modules of Mozart, i.e., the `Base` and `System` modules, follow this structure.

A software component Using procedural abstraction, we can turn this module into a software component. The software component is a function that returns a module:

```

fun {MyListFunctor}
  proc {Append ... } ... end
  proc {MergeSort ...} ... end
  proc {Sort ... } ... {MergeSort ...} ... end
  proc {Member ...} ... end
in
  `export` (append: Append
           sort: Sort
           member: Member
           ...)
end

```

Each time `MyListFunctor` is called, it creates and returns another `MyList` module. In general, `MyListFunctor` could have arguments, which are the other modules needed for `MyList`.

From this definition, it is clear that functors are just values in the language. They share the following properties with procedure values:

- A functor definition can be evaluated at run time, giving a functor.
- A functor can have external references to other language entities. For example, it is easy to make a functor that contains data calculated at run time. This is useful, for example, to include large tables or image data in source form.
- A functor can be stored in a file by using the `Pickle` module. This file can be read by any Mozart process. This makes it easy to create libraries of third-party functors, such as `MOGUL`.
- A functor is lightweight; it can be used to encapsulate a single entity such as one object or class, in order to make explicit the modules needed by the entity.

Because functors are values, it is possible to manipulate them in sophisticated ways within the language. For example, a software component can be built that implements *component-based programming*, in which components determine at run time which components they need and when to link them. Even more flexibility is possible when dynamic typing is used. A component can link an

arbitrary component at run time, by installing any functors and calling them according to their needs.

Linguistic support This software component abstraction is a reasonable one to organize large programs. To make it easier to use, to ensure that it is not used incorrectly, and to make clear the intention of the programmer (avoiding confusion with other higher-order programming techniques), we turn it into a linguistic abstraction. The function `MyListFunctor` corresponds to the following functor syntax:

```
functor
export
    append:Append
    sort:Sort
    member:Member
    ...
define
    proc {Append ... } ... end
    proc {MergeSort ...} ... end
    proc {Sort ... } ... {MergeSort ...} ... end
    proc {Member ...} ... end
end
```

Note that the statement between **define** and **end** does implicit variable declaration, exactly like the statement between **local** and **in**.

Assume that this functor has been compiled and stored in the file `MyList.ozf` (we will see below how to compile a functor). Then the module can be created as follows in the interactive interface:

```
declare [MyList]={Module.link ['MyList.ozf']}
```

The function `Module.link` is defined in the System module `Module`. It takes a list of functors, loads them from the file system, links them together (i.e., evaluates them together, so that each module sees its imported modules), and returns a corresponding list of modules. The `Module` module allows doing many other operations on functors and modules.

Importing modules Software components can depend on other software components. To be precise, instantiating a software component creates a module. The instantiation might need other modules. In the new syntax, we declare this with **import** declarations. To import a library module it is enough to give the name of its functor. On the other hand, to import a user-defined module requires stating the file name or URL of the file where the functor is stored.²⁰ This is reasonable, since the system knows where the library modules are stored, but

²⁰Other naming schemes are possible, in which functors have some logical name in a component management system.