

Figure 2.13: A partial value with no unbound variables, i.e., a complete value

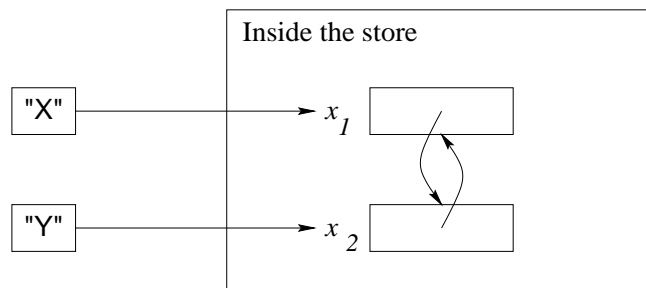


Figure 2.14: Two variables bound together

be bound to several partial values, as long as they are compatible with each other. We say a set of partial values is *compatible* if the unbound variables in them can be bound in such a way as to make them all equal. For example, `person(age:25)` and `person(age:x)` are compatible (because  $x$  can be bound to 25), but `person(age:25)` and `person(age:26)` are not.

### 2.2.7 Variable-variable binding

Variables can be bound to variables. For example, consider two unbound variables  $x_1$  and  $x_2$  referred to by the identifiers `x` and `y`. After doing the bind `x=y`, we get the situation in Figure 2.14. The two variables  $x_1$  and  $x_2$  are equal to each other. The figure shows this by letting each variable refer to the other. We say that  $\{x_1, x_2\}$  form an *equivalence set*.<sup>4</sup> We also write this as  $x_1 = x_2$ . Three variables that are bound together are written as  $x_1 = x_2 = x_3$  or  $\{x_1, x_2, x_3\}$ . Drawn in a figure, these variables would form a circular chain. Whenever one variable in an equivalence set is bound, then all variables see the binding. Figure 2.15 shows the result of doing `x=[1 2 3]`.

<sup>4</sup>From a formal viewpoint, the two variables form an equivalence class with respect to equality.

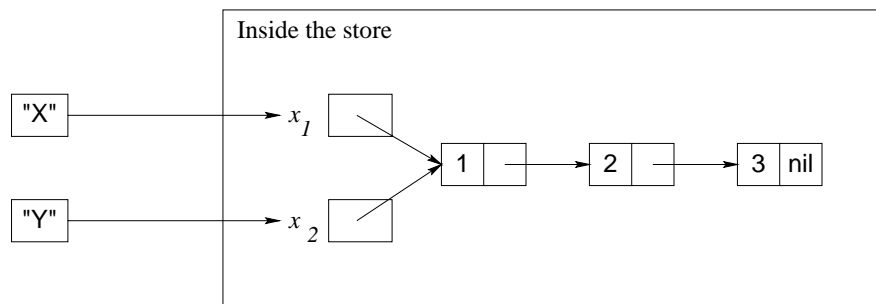


Figure 2.15: The store after binding one of the variables

### 2.2.8 Dataflow variables

In the declarative model, *creating* a variable and *binding* it are done separately. What happens if we try to use the variable before it is bound? We call this a *variable use error*. Some languages create and bind variables in one step, so that use errors cannot occur. This is the case for functional programming languages. Other languages allow creating and binding to be separate. Then we have the following possibilities when there is a use error:

1. Execution continues and no error message is given. The variable's content is undefined, i.e. it is "garbage": whatever is found in memory. This is what C++ does.
2. Execution continues and no error message is given. The variable is initialized to a default value when it is declared, e.g., to 0 for an integer. This is what Java does.
3. Execution stops with an error message (or an exception is raised). This is what Prolog does for arithmetic operations.
4. Execution waits until the variable is bound and then continues.

These cases are listed in increasing order of niceness. The first case is very bad, since different executions of the same program can give different results. What's more, since the existence of the error is not signaled, the programmer is not even aware when this happens. The second is somewhat better. If the program has a use error, then at least it will always give the same result, even if it is a wrong one. Again the programmer is not made aware of the error's existence.

The third and fourth cases are reasonable in certain situations. In the third, a program with a use error will signal this fact, instead of silently continuing. This is reasonable in a sequential system, since there really is an error. It is unreasonable in a concurrent system, since the result becomes nondeterministic: depending on the timing, sometimes an error is signaled and sometimes not. In the fourth, the program will wait until the variable is bound, and then continue. This is unreasonable in a sequential system, since the program will wait forever.

$\langle s \rangle ::=$	
<b>skip</b>	Empty statement
$\langle s \rangle_1 \langle s \rangle_2$	Statement sequence
<b>local</b> $\langle x \rangle$ <b>in</b> $\langle s \rangle$ <b>end</b>	Variable creation
$\langle x \rangle_1 = \langle x \rangle_2$	Variable-variable binding
$\langle x \rangle = \langle v \rangle$	Value creation
<b>if</b> $\langle x \rangle$ <b>then</b> $\langle s \rangle_1$ <b>else</b> $\langle s \rangle_2$ <b>end</b>	Conditional
<b>case</b> $\langle x \rangle$ <b>of</b> $\langle \text{pattern} \rangle$ <b>then</b> $\langle s \rangle_1$ <b>else</b> $\langle s \rangle_2$ <b>end</b>	Pattern matching
$\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$	Procedure application

Table 2.1: The declarative kernel language

It is reasonable in a concurrent system, where it could be part of normal operation that some other thread binds the variable.<sup>5</sup> The computation models of this book use the fourth case.

Declarative variables that cause the program to wait until they are bound are called *dataflow* variables. The declarative model uses dataflow variables because they are tremendously useful in concurrent programming, i.e., for programs with activities that run independently. If we do two concurrent operations, say  $A=23$  and  $B=A+1$ , then with the fourth solution this will *always* run correctly and give the answer  $B=24$ . It doesn't matter whether  $A=23$  is tried first or whether  $B=A+1$  is tried first. With the other solutions, there is no guarantee of this. This property of *order-independence* makes possible the declarative concurrency of Chapter 4. It is at the heart of why dataflow variables are a good idea.

## 2.3 Kernel language

The declarative model defines a simple kernel language. All programs in the model can be expressed in this language. We first define the kernel language syntax and semantics. Then we explain how to build a full language on top of the kernel language.

### 2.3.1 Syntax

The kernel syntax is given in Tables 2.1 and 2.2. It is carefully designed to be a subset of the full language syntax, i.e., all statements in the kernel language are valid statements in the full language.

<sup>5</sup>Still, during development, a good debugger should capture undesirable suspensions if there are no other running threads.

$\langle v \rangle$	$::=$	$\langle \text{number} \rangle \mid \langle \text{record} \rangle \mid \langle \text{procedure} \rangle$
$\langle \text{number} \rangle$	$::=$	$\langle \text{int} \rangle \mid \langle \text{float} \rangle$
$\langle \text{record} \rangle, \langle \text{pattern} \rangle$	$::=$	$\langle \text{literal} \rangle$ $\mid \langle \text{literal} \rangle (\langle \text{feature} \rangle_1: \langle x \rangle_1 \dots \langle \text{feature} \rangle_n: \langle x \rangle_n)$
$\langle \text{procedure} \rangle$	$::=$	<b>proc</b> { \$ $\langle x \rangle_1 \dots \langle x \rangle_n$ } $\langle s \rangle$ <b>end</b>
$\langle \text{literal} \rangle$	$::=$	$\langle \text{atom} \rangle \mid \langle \text{bool} \rangle$
$\langle \text{feature} \rangle$	$::=$	$\langle \text{atom} \rangle \mid \langle \text{bool} \rangle \mid \langle \text{int} \rangle$
$\langle \text{bool} \rangle$	$::=$	<b>true</b> $\mid$ <b>false</b>

Table 2.2: Value expressions in the declarative kernel language

### Statement syntax

Table 2.1 defines the syntax of  $\langle s \rangle$ , which denotes a statement. There are eight statements in all, which we will explain later.

### Value syntax

Table 2.2 defines the syntax of  $\langle v \rangle$ , which denotes a value. There are three kinds of value expressions, denoting numbers, records, and procedures. For records and patterns, the arguments  $\langle x \rangle_1, \dots, \langle x \rangle_n$  must all be distinct identifiers. This ensures that all variable-variable bindings are written as explicit kernel operations.

### Variable identifier syntax

Table 2.1 uses the nonterminals  $\langle x \rangle$  and  $\langle y \rangle$  to denote a variable identifier. We will also use  $\langle z \rangle$  to denote identifiers. There are two ways to write a variable identifier:

- An uppercase letter followed by zero or more alphanumeric characters (letters or digits or underscores), for example `X`, `X1`, or `ThisIsALongVariable_IsntIt`.
- Any sequence of printable characters enclosed within ``` (back-quote) characters, e.g., ``this is a 25$\variable!``.

A precise definition of identifier syntax is given in Appendix C. All newly-declared variables are unbound before any statement is executed. All variable identifiers must be declared explicitly.

## 2.3.2 Values and types

A *type* or *data type* is a set of values together with a set of operations on those values. A value is “of a type” if it is in the type’s set. The declarative model is *typed* in the sense that it has a well-defined set of types, called *basic types*. For example, programs can calculate with integers or with records, which are all

of integer type or record type, respectively. Any attempt to use an operation with values of the wrong type is detected by the system and will raise an error condition (see Section 2.6). The model imposes no other restrictions on the use of types.

Because all uses of types are checked, it is not possible for a program to behave outside of the model, e.g., to crash because of undefined operations on its internal data structures. It is still possible for a program to raise an error condition, for example by dividing by zero. In the declarative model, a program that raises an error condition will terminate immediately. There is nothing in the model to handle errors. In Section 2.6 we extend the declarative model with a new concept, *exceptions*, to handle errors. In the extended model, type errors can be handled within the model.

In addition to basic types, programs can define their own types, which are called *abstract data types*, ADT for short. Chapter 3 and later chapters show how to define ADTs.

### Basic types

The basic types of the declarative model are numbers (integers and floats), records (including atoms, booleans, tuples, lists, and strings), and procedures. Table 2.2 gives their syntax. The nonterminal  $\langle v \rangle$  denotes a partially constructed value. Later in the book we will see other basic types, including chunks, functors, cells, dictionaries, arrays, ports, classes, and objects. Some of these are explained in Appendix B.

### Dynamic typing

There are two basic approaches to typing, namely dynamic and static typing. In static typing, all variable types are known at compile time. In dynamic typing, the variable type is known only when the variable is bound. The declarative model is dynamically typed. The compiler tries to verify that all operations use values of the correct type. But because of dynamic typing, some type checks are necessarily left for run time.

### The type hierarchy

The basic types of the declarative model can be classified into a hierarchy. Figure 2.16 shows this hierarchy, where each node denotes a type. The hierarchy is ordered by *set inclusion*, i.e., all values of a node's type are also values of the parent node's type. For example, all tuples are records and all lists are tuples. This implies that all operations of a type are also legal for a subtype, e.g., all list operations work also for strings. Later on in the book we will extend this hierarchy. For example, literals can be either atoms (explained below) or another kind of constant called names (see Section 3.7.5). The parts where the hierarchy is incomplete are given as "...".

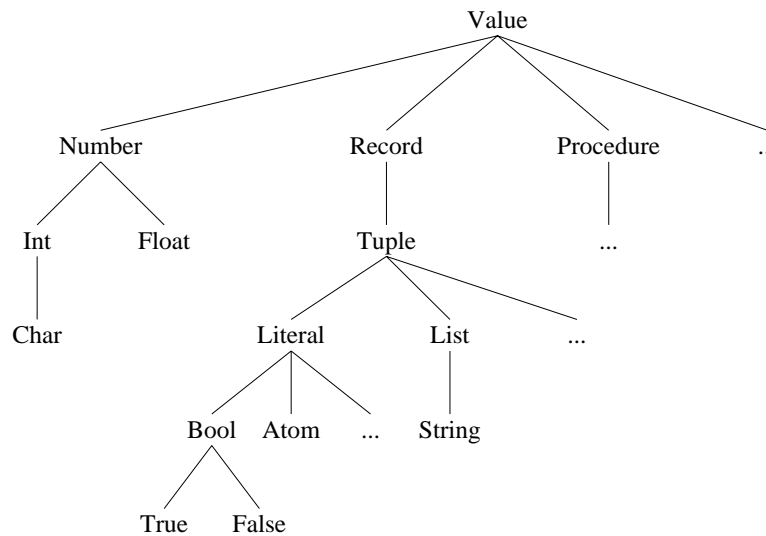


Figure 2.16: The type hierarchy of the declarative model

### 2.3.3 Basic types

We give some examples of the basic types and how to write them. See Appendix B for more complete information.

- **Numbers.** Numbers are either integers or floating point numbers. Examples of integers are 314, 0, and ~10 (minus 10). Note that the minus sign is written with a tilde “~”. Examples of floating point numbers are 1.0, 3.4, 2.0e2, and ~2.0E~2.
- **Atoms.** An atom is a kind of symbolic constant that can be used as a single element in calculations. There are several different ways to write atoms. An atom can be written as a sequence of characters starting with a lowercase letter followed by any number of alphanumeric characters. An atom can also be written as any sequence of printable characters enclosed in single quotes. Examples of atoms are `a_person`, `donkeyKong3`, and `##### hello #####`.
- **Booleans.** A boolean is either the symbol **true** or the symbol **false**.
- **Records.** A record is a compound data structure. It consists of a label followed by a set of pairs of features and variable identifiers. Features can be atoms, integers, or booleans. Examples of records are `person(age:X1 name:X2)` (with features `age` and `name`), `person(1:X1 2:X2)`, `^(1:H 2:T)`, `^(1:H 2:T)`, `nil`, and `person`. An atom is a record with no features.
- **Tuples.** A tuple is a record whose features are consecutive integers starting from 1. The features do not have to be written in this case. Examples of

tuples are `person(1:x1 2:x2)` and `person(x1 x2)`, both of which mean the same.

- **Lists.** A list is either the atom `nil` or the tuple `⌈ | ⌈(H T)` (label is vertical bar), where `T` is either unbound or bound to a list. This tuple is called a *list pair* or a *cons*. There is syntactic sugar for lists:
  - The `⌈ | ⌈` label can be written as an infix operator, so that `H|T` means the same as `⌈ | ⌈(H T)`.
  - The `⌈ | ⌈` operator associates to the right, so that `1|2|3|nil` means the same as `1|(2|(3|nil))`.
  - Lists that end in `nil` can be written with brackets `[ ... ]`, so that `[1 2 3]` means the same as `1|2|3|nil`. These lists are called *complete lists*.
- **Strings.** A string is a list of character codes. Strings can be written with double quotes, so that `"E=mc^2"` means the same as `[69 61 109 99 94 50]`.
- **Procedures.** A procedure is a value of the procedure type. The statement:

$$\langle x \rangle = \text{proc } \{ \$ \langle y \rangle_1 \dots \langle y \rangle_n \} \langle s \rangle \text{ end}$$

binds  $\langle x \rangle$  to a new procedure value. That is, it simply declares a new procedure. The `$` indicates that the procedure value is anonymous, i.e., created without being bound to an identifier. There is a syntactic short-cut that is more familiar:

$$\text{proc } \{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \} \langle s \rangle \text{ end}$$

The `$` is replaced by an identifier. This creates the procedure value and immediately tries to bind it to  $\langle x \rangle$ . This short-cut is perhaps easier to read, but it blurs the distinction between creating the value and binding it to an identifier.

### 2.3.4 Records and procedures

We explain why chose records and procedures as basic concepts in the kernel language. This section is intended for readers with some programming experience who wonder why we designed the kernel language the way we did.

#### The power of records

Records are the basic way to structure data. They are the building blocks of most data structures, including lists, trees, queues, graphs, etc., as we will see in Chapter 3. Records play this role to some degree in most programming languages.

But we shall see that their power can go much beyond this role. The extra power appears in greater or lesser degree depending on how well or how poorly the language supports them. For maximum power, the language should make it easy to create them, take them apart, and manipulate them. In the declarative model, a record is created by simply writing it down, with a compact syntax. A record is taken apart by simply writing down a pattern, also with a compact syntax. Finally, there are many operations to manipulate records: to add, remove, or select fields, to convert to a list and back, etc. In general, languages that provide this level of support for records are called *symbolic* languages.

When records are strongly supported, they can be used to increase the effectiveness of many other techniques. This book focuses on three in particular: object-oriented programming, graphical user interface (GUI) design, and component-based programming. In object-oriented programming, Chapter 7 shows how records can represent messages and method heads, which are what objects use to communicate. In GUI design, Chapter 10 shows how records can represent “widgets”, the basic building blocks of a user interface. In component-based programming, Section 3.9 shows how records can represent modules, which group together related operations.

### Why procedures?

A reader with some programming experience may wonder why our kernel language has procedures as a basic construct. Fans of object-oriented programming may wonder why we do not use objects instead. Fans of functional programming may wonder why we do not use functions. We could have chosen either possibility, but we did not. The reasons are quite straightforward.

Procedures are more appropriate than objects because they are simpler. Objects are actually quite complicated, as Chapter 7 explains. Procedures are more appropriate than functions because they do not necessarily define entities that behave like mathematical functions.<sup>6</sup> For example, we define both components and objects as abstractions based on procedures. In addition, procedures are flexible because they do not make any assumptions about the number of inputs and outputs. A function always has exactly one output. A procedure can have any number of inputs and outputs, including zero. We will see that procedures are extremely powerful building blocks, when we talk about higher-order programming in Section 3.6.

---

<sup>6</sup>From a theoretical point of view, procedures are “processes” as used in concurrent calculi such as the  $\pi$  calculus. The arguments are channels. In this chapter we use processes that are composed sequentially with single-shot channels. Chapters 4 and 5 show other types of channels (with sequences of messages) and do concurrent composition of processes.



Operation	Description	Argument type
<code>A==B</code>	Equality comparison	Value
<code>A\=B</code>	Nonequality comparison	Value
<code>{IsProcedure P}</code>	Test if procedure	Value
<code>A&lt;=B</code>	Less than or equal comparison	Number or Atom
<code>A&lt;B</code>	Less than comparison	Number or Atom
<code>A&gt;=B</code>	Greater than or equal comparison	Number or Atom
<code>A&gt;B</code>	Greater than comparison	Number or Atom
<code>A+B</code>	Addition	Number
<code>A-B</code>	Subtraction	Number
<code>A*B</code>	Multiplication	Number
<code>A div B</code>	Division	Int
<code>A mod B</code>	Modulo	Int
<code>A/B</code>	Division	Float
<code>{Arity R}</code>	Arity	Record
<code>{Label R}</code>	Label	Record
<code>R.F</code>	Field selection	Record

Table 2.3: Examples of basic operations

### 2.3.5 Basic operations

Table 2.3 gives the basic operations that we will use in this chapter and the next. There is syntactic sugar for many of these operations so that they can be written concisely as expressions. For example, `X=A*B` is syntactic sugar for `{Number.´*´ A B X}`, where `Number.´*´` is a procedure associated with the type `Number`.<sup>7</sup> All operations can be denoted in some long way, e.g., `Value.´==´`, `Value.´<´`, `Int.´div´`, `Float.´/´`. The table uses the syntactic sugar when it exists.

- **Arithmetic.** Floating point numbers have the four basic operations, `+`, `-`, `*`, and `/`, with the usual meanings. Integers have the basic operations `+`, `-`, `*`, `div`, and `mod`, where `div` is integer division (truncate the fractional part) and `mod` is the integer modulo, i.e., the remainder after a division. For example, `10 mod 3=1`.
- **Record operations.** Three basic operations on records are `Arity`, `Label`, and `“.”` (dot, which means field selection). For example, given:

```
X=person(name:"George" age:25)
```

then `{Arity X}=[age name]`, `{Label X}=person`, and `X.age=25`. The call to `Arity` returns a list that contains first the integer features in ascending order and then the atom features in ascending lexicographic order.

<sup>7</sup>To be precise, `Number` is a module that groups the operations of the `Number` type and `Number.´*´` selects the multiplication operation.

- **Comparisons.** The boolean comparison functions include `==` and `\=`, which can compare any two values for equality, as well as the numeric comparisons `=<`, `<`, `>=`, and `>`, which can compare two integers, two floats, or two atoms. Atoms are compared according to the lexicographic order of their print representations. In the following example, `Z` is bound to the maximum of `X` and `Y`:

```
declare X Y Z T in
X=5 Y=10
T=(X>=Y)
if T then Z=X else Z=Y end
```

There is syntactic sugar so that an `if` statement accepts an expression as its condition. The above example can be rewritten as:

```
declare X Y Z in
X=5 Y=10
if X>=Y then Z=X else Z=Y end
```

- **Procedure operations.** There are three basic operations on procedures: defining them (with the `proc` statement), calling them (with the curly brace notation), and testing whether a value is a procedure with the `IsProcedure` function. The call `{IsProcedure P}` returns `true` if `P` is a procedure and `false` otherwise.

Appendix B gives a more complete set of basic operations.

## 2.4 Kernel language semantics

The kernel language execution consists of evaluating functions over partial values. To see this, we give the semantics of the kernel language in terms of a simple operational model. The model is designed to let the programmer reason about both correctness and complexity in a simple way. It is a kind of abstract machine, but at a high level of abstraction that leaves out details such as registers and explicit memory addresses.

### 2.4.1 Basic concepts

Before giving the formal semantics, let us give some examples to give intuition on how the kernel language executes. This will motivate the semantics and make it easier to understand.

#### A simple execution

During normal execution, statements are executed one by one in textual order. Let us look at a simple execution:

```
local A B C D in
  A=11
  B=2
  C=A+B
  D=C*C
end
```

This looks simple enough; it will bind `D` to 169. Let us look more closely at what it does. The **local** statement creates four new variables in the store, and makes the four identifiers `A`, `B`, `C`, `D` refer to them. (For convenience, this extends slightly the **local** statement of Table 2.1.) This is followed by two bindings, `A=11` and `B=2`. The addition `C=A+B` adds the values of `A` and `B` and binds `C` to the result 13. The multiplication `D` multiplies the value of `C` by itself and binds `D` to the result 169. This is quite simple.

### Variable identifiers and static scoping

We saw that the **local** statement does two things: it creates a new variable and it sets up an identifier to refer to the variable. The identifier only refers to the variable inside the **local** statement, i.e., between the **local** and the **end**. We call this the *scope* of the identifier. Outside of the scope, the identifier does not mean the same thing. Let us look closer at what this implies. Consider the following fragment:

```
local X in
  X=1
  local x in
    X=2
    {Browse X}
  end
  {Browse X}
end
```

What does it display? It displays first 2 and then 1. There is just one identifier, `x`, but at different points during the execution, it refers to different variables.

Let us summarize this idea. The meaning of an identifier like `x` is determined by the innermost **local** statement that declares `x`. The area of the program where `x` keeps this meaning is called the scope of `x`. We can find out the scope of an identifier by simply inspecting the text of the program; we do not have to do anything complicated like execute or analyze the program. This scoping rule is called *lexical scoping* or *static scoping*. Later we will see another kind of scoping rule, dynamic scoping, that is sometimes useful. But lexical scoping is by far the most important kind of scoping rule because it is localized, i.e., the meaning of an identifier can be determined by looking at a small part of the program.

## Procedures

Procedures are one of the most important basic building blocks of any language. We give a simple example that shows how to define and call a procedure. Here is a procedure that binds *z* to the maximum of *x* and *y*:

```
proc {Max X Y ?Z}
  if X>=Y then Z=X else Z=Y end
end
```

To make the definition easier to read, we mark the output argument with a question mark “?”. This has absolutely no effect on execution; it is just a comment. Calling {Max 3 5 C} binds *C* to 5. How does the procedure work, exactly? When *Max* is called, the identifiers *x*, *y*, and *z* are bound to 3, 5, and the unbound variable referenced by *C*. When *Max* binds *z*, then it binds this variable. Since *C* also references this variable, this also binds *C*. This way of passing parameters is called *call by reference*. Procedures output results by being passed references to unbound variables, which are bound inside the procedure. This book mostly uses call by reference, both for dataflow variables and for mutable variables. Section 6.4.4 explains some other parameter passing mechanisms.

## Procedures with external references

Let us examine the body of *Max*. It is just an **if** statement:

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

This statement has one particularity, though: it cannot be executed! This is because it does not define the identifiers *x*, *y*, and *z*. These undefined identifiers are called *free identifiers*. Sometimes these are called free variables, although strictly speaking they are not variables. When put inside the procedure *Max*, the statement *can* be executed, because all the free identifiers are declared as procedure arguments.

What happens if we define a procedure that only declares *some* of the free identifiers as arguments? For example, let’s define the procedure *LB* with the same procedure body as *Max*, but only two arguments:

```
proc {LB X ?Z}
  if X>=Y then Z=X else Z=Y end
end
```

What does this procedure do when executed? Apparently, it takes any number *x* and binds *z* to *x* if *x*≥*y*, but to *y* otherwise. That is, *z* is always at least *y*. What is the value of *y*? It is not one of the procedure arguments. It has to be the value of *y* *when the procedure is defined*. This is a consequence of static scoping. If *y*=9 when the procedure is defined, then calling {LB 3 Z} binds *z* to 9. Consider the following program fragment:

```
local Y LB in
  Y=10
```

```

proc {LB X ?Z}
  if X>=Y then Z=X else Z=Y end
end
local Y=15 Z in
  {LB 5 Z}
end
end

```

What does the call {LB 5 Z} bind Z to? It will be bound to 10. The binding Y=15 when LB is called is ignored; it is the binding Y=10 at the procedure definition that is important.

### Dynamic scoping versus static scoping

Consider the following simple example:

```

local P Q in
  proc {Q X} {Browse stat(X)} end
  proc {P X} {Q X} end
  local Q in
    proc {Q X} {Browse dyn(X)} end
    {P hello}
  end
end
end

```

What should this display, `stat(hello)` or `dyn(hello)`? Static scoping says that it will display `stat(hello)`. In other words, P uses the version of Q that exists at P's definition. But there is another solution: P could use the version of Q that exists at P's call. This is called *dynamic scoping*. Both have been used as the default scoping rule in programming languages. The original Lisp language was dynamically scoped. Common Lisp and Scheme, which are descended from Lisp, are statically scoped by default. Common Lisp still allows to declare dynamically-scoped variables, which it calls *special variables* [181]. Which default is better? The correct default is procedure values with static scoping. This is because a procedure that works when it is defined will continue to work, independent of the environment where it is called. This is an important software engineering property.

Dynamic scoping remains useful in some well-defined areas. For example, consider the case of a procedure whose code is transferred across a network from one computer to another. Some of this procedure's external references, for example calls to common library operations, can use dynamic scoping. This way, the procedure will use local code for these operations instead of remote code. This is much more efficient.<sup>8</sup>

---

<sup>8</sup>However, there is no guarantee that the operation will behave in the same way on the target machine. So even for distributed programs the default should be static scoping.

### Procedural abstraction

Let us summarize what we learned from **Max** and **LB**. Three concepts play an important role:

- *Procedural abstraction.* Any statement can be made into a procedure by putting it inside a procedure declaration. This is called *procedural abstraction*. We also say that the statement is abstracted into a procedure.
- *Free identifiers.* A free identifier in a statement is an identifier that is not defined in that statement. It might be defined in an enclosing statement.
- *Static scoping.* A procedure can have external references, which are free identifiers in the procedure body that are not declared as arguments. **LB** has one external reference. **Max** has none. The value of an external reference is its value when the procedure is defined. This is a consequence of static scoping.

Procedural abstraction and static scoping together form one of the most powerful tools presented in this book. In the semantics, we will see that they can be implemented in a simple way.

### Dataflow behavior

In the single-assignment store, variables can be unbound. On the other hand, some statements need bound variables, otherwise they cannot execute. For example, what happens when we execute:

```

local X Y Z in
  X=10
  if X>=Y then Z=X else Z=Y end
end

```

The comparison  $X \geq Y$  returns **true** or **false**, if it can decide which is the case. If  $Y$  is unbound, it cannot decide, strictly speaking. What does it do? Continuing with either **true** or **false** would be incorrect. Raising an error would be a drastic measure, since the program has done nothing wrong (it has done nothing right either). We decide that the program will simply stop its execution, without signaling any kind of error. If some other activity (to be determined later) binds  $Y$  then the stopped execution can continue as if nothing had perturbed the normal flow of execution. This is called *dataflow behavior*. Dataflow behavior underlies a second powerful tool presented in this book, namely *concurrency*. In the semantics, we will see that dataflow behavior can be implemented in a simple way.

#### 2.4.2 The abstract machine

We will define the kernel semantics as an *operational* semantics, i.e., it defines the meaning of the kernel language through its execution on an abstract machine. We

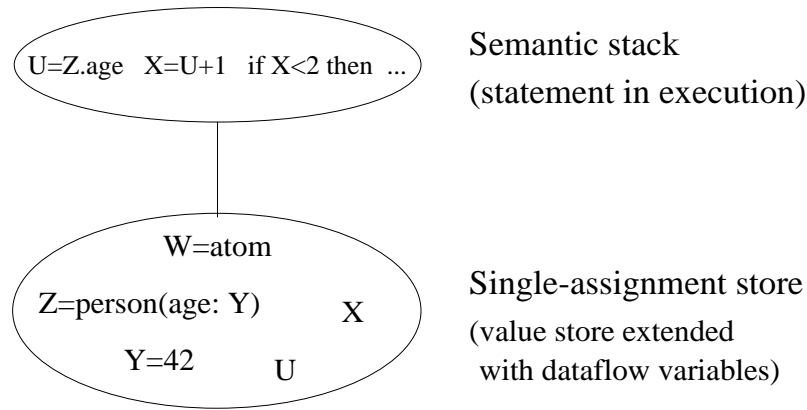


Figure 2.17: The declarative computation model

first define the basic concepts of the abstract machine: environments, semantic statement, statement stack, execution state, and computation. We then show how to execute a program. Finally, we explain how to calculate with environments, which is a common semantic operation.

### Overview of concepts

A running program is defined in terms of a computation, which is a sequence of execution states. Let us define exactly what this means. We need the following concepts:

- A *single-assignment store*  $\sigma$  is a set of store variables. These variables are partitioned into (1) sets of variables that are equal but unbound and (2) variables that are bound to a number, record, or procedure. For example, in the store  $\{x_1, x_2 = x_3, x_4 = a | x_2\}$ ,  $x_1$  is unbound,  $x_2$  and  $x_3$  are equal and unbound, and  $x_4$  is bound to the partial value  $a | x_2$ . A store variable bound to a value is indistinguishable from that value. This is why a store variable is sometimes called a *store entity*.
- An *environment*  $E$  is a mapping from variable identifiers to entities in  $\sigma$ . This is explained in Section 2.2. We will write  $E$  as a set of pairs, e.g.,  $\{X \rightarrow x, Y \rightarrow y\}$ , where  $X, Y$  are identifiers and  $x, y$  refer to store entities.
- A *semantic statement* is a pair  $(\langle s \rangle, E)$  where  $\langle s \rangle$  is a statement and  $E$  is an environment. The semantic statement relates a statement to what it references in the store. The set of possible statements is given in Section 2.3.
- An *execution state* is a pair  $(ST, \sigma)$  where  $ST$  is a stack of semantic statements and  $\sigma$  is a single-assignment store. Figure 2.17 gives a picture of the execution state.

- A *computation* is a sequence of execution states starting from an initial state:  $(ST_0, \sigma_0) \rightarrow (ST_1, \sigma_1) \rightarrow (ST_2, \sigma_2) \rightarrow \dots$

A single transition in a computation is called a *computation step*. A computation step is *atomic*, i.e., there are no visible intermediate states. It is as if the step is done “all at once”. In this chapter, all computations are *sequential*, i.e., the execution state contains exactly *one* statement stack, which is transformed by a linear sequence of computation steps.

### Program execution

Let us execute a program in this semantics. A program is simply a statement  $\langle s \rangle$ . Here is how to execute the program:

- The initial execution state is:

$$([\langle s \rangle, \phi], \phi)$$

That is, the initial store is empty (no variables, empty set  $\phi$ ) and the initial execution state has just one semantic statement  $(\langle s \rangle, \phi)$  in the stack  $ST$ . The semantic statement contains  $\langle s \rangle$  and an empty environment  $(\phi)$ . We use brackets [...] to denote the stack.

- At each step, the first element of  $ST$  is popped and execution proceeds according to the form of the element.
- The final execution state (if there is one) is a state in which the semantic stack is empty.

A semantic stack  $ST$  can be in one of three run-time states:

- *Runnable*:  $ST$  can do a computation step.
- *Terminated*:  $ST$  is empty.
- *Suspended*:  $ST$  is not empty, but it cannot do any computation step.

### Calculating with environments

A program execution often does calculations with environments. An environment  $E$  is a function that maps variable identifiers  $\langle x \rangle$  to store entities (both unbound variables and values). The notation  $E(\langle x \rangle)$  retrieves the entity associated with the identifier  $\langle x \rangle$  from the store. To define the semantics of the abstract machine instructions, we need two common operations on environments, namely *adjunction* and *restriction*.

Adjunction defines a new environment by adding a mapping to an existing one. The notation:

$$E + \{ \langle x \rangle \rightarrow x \}$$



denotes a new environment  $E'$  constructed from  $E$  by adding the mapping  $\{\langle x \rangle \rightarrow x\}$ . This mapping overrides any other mapping from the identifier  $\langle x \rangle$ . That is,  $E'(\langle x \rangle)$  is equal to  $x$ , and  $E'(\langle y \rangle)$  is equal to  $E(\langle y \rangle)$  for all identifiers  $\langle y \rangle$  different from  $\langle x \rangle$ . When we need to add more than one mapping at once, we write  $E + \{\langle x \rangle_1 \rightarrow x_1, \dots, \langle x \rangle_n \rightarrow x_n\}$ .

Restriction defines a new environment whose domain is a subset of an existing one. The notation:

$$E|_{\{\langle x \rangle_1, \dots, \langle x \rangle_n\}}$$

denotes a new environment  $E'$  such that  $\text{dom}(E') = \text{dom}(E) \cap \{\langle x \rangle_1, \dots, \langle x \rangle_n\}$  and  $E'(\langle x \rangle) = E(\langle x \rangle)$  for all  $\langle x \rangle \in \text{dom}(E')$ . That is, the new environment does not contain any identifiers other than those mentioned in the set.

### 2.4.3 Non-suspendable statements

We first give the semantics of the statements that can never suspend.

#### The **skip** statement

The semantic statement is:

$$(\mathbf{skip}, E)$$

Execution is complete after this pair is popped from the semantic stack.

#### Sequential composition

The semantic statement is:

$$(\langle s \rangle_1 \langle s \rangle_2, E)$$

Execution consists of the following actions:

- Push  $(\langle s \rangle_2, E)$  on the stack.
- Push  $(\langle s \rangle_1, E)$  on the stack.

#### Variable declaration (the **local** statement)

The semantic statement is:

$$(\mathbf{local} \langle x \rangle \mathbf{in} \langle s \rangle \mathbf{end}, E)$$

Execution consists of the following actions:

- Create a new variable  $x$  in the store.
- Let  $E'$  be  $E + \{\langle x \rangle \rightarrow x\}$ , i.e.,  $E'$  is the same as  $E$  except that it adds a mapping from  $\langle x \rangle$  to  $x$ .
- Push  $(\langle s \rangle, E')$  on the stack.

### Variable-variable binding

The semantic statement is:

$$(\langle x \rangle_1 = \langle x \rangle_2, E)$$

Execution consists of the following action:

- Bind  $E(\langle x \rangle_1)$  and  $E(\langle x \rangle_2)$  in the store.

### Value creation

The semantic statement is:

$$(\langle x \rangle = \langle v \rangle, E)$$

where  $\langle v \rangle$  is a partially constructed value that is either a record, number, or procedure. Execution consists of the following actions:

- Create a new variable  $x$  in the store.
- Construct the value represented by  $\langle v \rangle$  in the store and let  $x$  refer to it. All identifiers in  $\langle v \rangle$  are replaced by their store contents as given by  $E$ .
- Bind  $E(\langle x \rangle)$  and  $x$  in the store.

We have seen how to construct record and number values, but what about procedure values? In order to explain them, we have first to explain the concept of lexical scoping.

### Lexical scoping revisited

A statement  $\langle s \rangle$  can contain many occurrences of variable identifiers. For each identifier occurrence, we can ask the question: where was this identifier declared? If the declaration is in some statement (part of  $\langle s \rangle$  or not) that textually surrounds (i.e., encloses) the occurrence, then we say that the declaration obeys *lexical scoping*. Because the scope is determined by the source code text, this is also called *static scoping*.

Identifier occurrences in a statement can be *bound* or *free* with respect to that statement. An identifier occurrence  $x$  is *bound* with respect to a statement  $\langle s \rangle$  if it is declared inside  $\langle s \rangle$ , i.e., in a **local** statement, in the pattern of a **case** statement, or as argument of a procedure declaration. An identifier occurrence that is not bound is *free*. Free occurrences can only exist in incomplete program fragments, i.e., statements that cannot run. In a running program, it is always true that every identifier occurrence is bound.

### Bound identifier occurrences and bound variables

Do not confuse a bound identifier occurrence with a bound variable! A bound identifier occurrence does not exist at run time; it is a textual variable name that textually occurs inside a construct that declares it (e.g., a procedure or variable declaration). A bound variable exists at run time; it is a dataflow variable that is bound to a partial value.

Here is an example with both free and bound occurrences:

```
local Arg1 Arg2 in
  Arg1=111*111
  Arg2=999*999
  Res=Arg1+Arg2
end
```

In this statement, all variable identifiers are declared with lexical scoping. The identifier occurrences `Arg1` and `Arg2` are bound and the occurrence `Res` is free. This statement cannot be run. To make it runnable, it has to be part of a bigger statement that declares `Res`. Here is an extension that can run:

```
local Res in
  local Arg1 Arg2 in
    Arg1=111*111
    Arg2=999*999
    Res=Arg1+Arg2
  end
  {Browse Res}
end
```

This can run since it has no free identifier occurrences.

### Procedure values (closures)

Let us see how to construct a procedure value in the store. It is not as simple as one might imagine because procedures can have external references. For example:

```
proc {LowerBound X ?Z}
  if X>=Y then Z=X else Z=Y end
end
```

In this example, the `if` statement has three free variables, `x`, `y`, and `z`. Two of them, `x` and `z`, are also formal parameters. The third, `y`, is not a formal parameter. It has to be defined by the environment where the procedure is declared. The procedure value itself must have a mapping from `y` to the store. Otherwise, we could not call the procedure since `y` would be a kind of dangling reference.

Let us see what happens in the general case. A procedure expression is written as:

**proc** { \$  $\langle y \rangle_1 \dots \langle y \rangle_n$  }  $\langle s \rangle$  **end**

The statement  $\langle s \rangle$  can have free variable identifiers. Each free identifier is either a formal parameter or not. The first kind are defined anew each time the procedure is called. They form a subset of the formal parameters  $\{\langle y \rangle_1, \dots, \langle y \rangle_n\}$ . The second kind are defined once and for all when the procedure is declared. We call them the *external references* of the procedure. Let us write them as  $\{\langle z \rangle_1, \dots, \langle z \rangle_k\}$ . Then the procedure value is a pair:

( **proc** { \$  $\langle y \rangle_1 \dots \langle y \rangle_n$  }  $\langle s \rangle$  **end**,  $CE$  )

Here  $CE$  (the *contextual environment*) is  $E|_{\{\langle z \rangle_1, \dots, \langle z \rangle_n\}}$ , where  $E$  is the environment when the procedure is declared. This pair is put in the store just like any other value.

Because it contains an environment as well as a procedure definition, a procedure value is often called a *closure* or a *lexically-scoped closure*. This is because it “closes” (i.e., packages up) the environment at procedure definition time. This is also called environment capture. When the procedure is called, the contextual environment is used to construct the environment of the executing procedure body.

### 2.4.4 Suspendable statements

There are three statements remaining in the kernel language:

$\langle s \rangle ::= \dots$   
     | **if**  $\langle x \rangle$  **then**  $\langle s \rangle_1$  **else**  $\langle s \rangle_2$  **end**  
     | **case**  $\langle x \rangle$  **of**  $\langle \text{pattern} \rangle$  **then**  $\langle s \rangle_1$  **else**  $\langle s \rangle_2$  **end**  
     |  $\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$

What should happen with these statements if  $\langle x \rangle$  is unbound? From the discussion in Section 2.2.8, we know what should happen. The statements should simply wait until  $\langle x \rangle$  is bound. We say that they are *suspendable* statements. They have an *activation condition*, which is a condition that must be true for execution to continue. The condition is that  $E(\langle x \rangle)$  must be *determined*, i.e., bound to a number, record, or procedure.

In the declarative model of this chapter, once a statement suspends it will never continue, because there is no other execution that could make the activation condition true. The program simply stops executing. In Chapter 4, when we introduce concurrent programming, we will have executions with more than one semantic stack. A suspended stack  $ST$  can become runnable again if another stack does an operation that makes  $ST$ 's activation condition true. In that chapter we shall see that communication from one stack to another through the activation condition is the basis of dataflow execution. For now, let us stick with just one semantic stack.

### Conditional (the **if** statement)

The semantic statement is:

$$(\mathbf{if} \langle x \rangle \mathbf{then} \langle s \rangle_1 \mathbf{else} \langle s \rangle_2 \mathbf{end}, E)$$

Execution consists of the following actions:

- If the activation condition is true ( $E(\langle x \rangle)$  is determined), then do the following actions:
  - If  $E(\langle x \rangle)$  is not a boolean (**true** or **false**) then raise an error condition.
  - If  $E(\langle x \rangle)$  is **true**, then push  $(\langle s \rangle_1, E)$  on the stack.
  - If  $E(\langle x \rangle)$  is **false**, then push  $(\langle s \rangle_2, E)$  on the stack.
- If the activation condition is false, then execution does not continue. The execution state is kept as is. We say that execution *suspends*. The stop can be temporary. If some other activity in the system makes the activation condition true, then execution can resume.

### Procedure application

The semantic statement is:

$$(\{\langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n\}, E)$$

Execution consists of the following actions:

- If the activation condition is true ( $E(\langle x \rangle)$  is determined), then do the following actions:
  - If  $E(\langle x \rangle)$  is not a procedure value or is a procedure with a number of arguments different from  $n$ , then raise an error condition.
  - If  $E(\langle x \rangle)$  has the form  $(\mathbf{proc} \{ \$ \langle z \rangle_1 \dots \langle z \rangle_n \} \langle s \rangle \mathbf{end}, CE)$  then push  $(\langle s \rangle, CE + \{\langle z \rangle_1 \rightarrow E(\langle y \rangle_1), \dots, \langle z \rangle_n \rightarrow E(\langle y \rangle_n)\})$  on the stack.
- If the activation condition is false, then suspend execution.

### Pattern matching (the **case** statement)

The semantic statement is:

$$(\mathbf{case} \langle x \rangle \mathbf{of} \langle \mathbf{lit} \rangle (\langle \mathbf{feat} \rangle_1: \langle x \rangle_1 \dots \langle \mathbf{feat} \rangle_n: \langle x \rangle_n) \mathbf{then} \langle s \rangle_1 \mathbf{else} \langle s \rangle_2 \mathbf{end}, E)$$

(Here  $\langle \mathbf{lit} \rangle$  and  $\langle \mathbf{feat} \rangle$  are synonyms for  $\langle \mathbf{literal} \rangle$  and  $\langle \mathbf{feature} \rangle$ .) Execution consists of the following actions:

- If the activation condition is true ( $E(\langle x \rangle)$  is determined), then do the following actions:
  - If the label of  $E(\langle x \rangle)$  is  $\langle \text{lit} \rangle$  and its arity is  $[\langle \text{feat} \rangle_1 \dots \langle \text{feat} \rangle_n]$ , then push  $(\langle s \rangle_1, E + \{ \langle x \rangle_1 \rightarrow E(\langle x \rangle). \langle \text{feat} \rangle_1, \dots, \langle x \rangle_n \rightarrow E(\langle x \rangle). \langle \text{feat} \rangle_n \})$  on the stack.
  - Otherwise push  $(\langle s \rangle_2, E)$  on the stack.
- If the activation condition is false, then suspend execution.

### 2.4.5 Basic concepts revisited

Now that we have seen the kernel semantics, let us look again at the examples of Section 2.4.1 to see exactly what they are doing. We look at three examples; we suggest you do the others as exercises.

#### Variable identifiers and static scoping

We saw before that the following statement  $\langle s \rangle$  displays first 2 and then 1:

$$\langle s \rangle \equiv \left\{ \begin{array}{l} \text{local } x \text{ in} \\ \quad x=1 \\ \quad \left\{ \begin{array}{l} \text{local } x \text{ in} \\ \quad x=2 \\ \quad \{ \text{Browse } x \} \\ \text{end} \end{array} \right. \\ \quad \langle s \rangle_2 \equiv \{ \text{Browse } x \} \\ \text{end} \end{array} \right.$$

The same identifier  $x$  first refers to 2 and then refers to 1. We can understand better what happens by executing  $\langle s \rangle$  in our abstract machine.

1. The initial execution state is:

$$( [(\langle s \rangle, \phi)], \phi )$$

Both the environment and the store are empty ( $E = \phi$  and  $\sigma = \phi$ ).

2. After executing the outermost **local** statement and the binding  $x=1$ , we get:

$$( [(\langle s \rangle_1 \langle s \rangle_2, \{x \rightarrow x\})], \{x = 1\} )$$

The identifier  $x$  refers to the store variable  $x$ , which is bound to 1. The next statement to be executed is the sequential composition  $\langle s \rangle_1 \langle s \rangle_2$ .

3. After executing the sequential composition, we get:

$$( [\langle s \rangle_1, \{x \rightarrow x\}], [\langle s \rangle_2, \{x \rightarrow x\}]), \\ \{x = 1\} )$$

Each of the statements  $\langle s \rangle_1$  and  $\langle s \rangle_2$  has its own environment. At this point, the two environments have identical values.

4. Let us start executing  $\langle s \rangle_1$ . The first statement in  $\langle s \rangle_1$  is a **local** statement. Executing it gives:

$$( [(x=2 \ \text{Browse } x), \{x \rightarrow x'\}], [\langle s \rangle_2, \{x \rightarrow x\}]), \\ \{x', x = 1\} )$$

This creates the new variable  $x'$  and calculates the new environment  $\{x \rightarrow x\} + \{x \rightarrow x'\}$ , which is  $\{x \rightarrow x'\}$ . The second mapping of  $x$  overrides the first.

5. After the binding  $x=2$  we get:

$$( [(\text{Browse } x), \{x \rightarrow x'\}], [\text{Browse } x, \{x \rightarrow x\}]), \\ \{x' = 2, x = 1\} )$$

(Remember that  $\langle s \rangle_2$  is a **Browse**.) Now we see why the two **Browse** calls display different values. It is because they have different environments. The inner **local** statement is given its own environment, in which  $x$  refers to another variable. This does not affect the outer **local** statement, which keeps its environment no matter what happens in any other instruction.

## Procedure definition and call

Our next example defines and calls the procedure **Max**, which calculates the maximum of two numbers. With the semantics we can see precisely what happens

during the definition and execution of `Max`. Here is the example in kernel syntax:

$$\langle s \rangle \equiv \left\{ \begin{array}{l} \text{local Max in} \\ \quad \text{local A in} \\ \quad \quad \text{local B in} \\ \quad \quad \quad \text{local C in} \\ \quad \quad \quad \quad \text{Max=proc } \{ \$ X Y Z \} \\ \quad \quad \quad \quad \quad \langle s \rangle_3 \equiv \left\{ \begin{array}{l} \text{local T in} \\ \quad T = (X \geq Y) \\ \quad \langle s \rangle_4 \equiv \text{if T then Z=X else Z=Y end} \\ \quad \text{end} \\ \quad \text{end} \\ \quad A=3 \\ \quad B=5 \\ \quad \langle s \rangle_2 \equiv \{ \text{Max A B C} \} \\ \quad \text{end} \\ \quad \text{end} \\ \quad \text{end} \\ \text{end} \end{array} \right. \end{array} \right.$$

This statement is in the kernel language syntax. We can see it as the expanded form of:

```
local Max C in
  proc {Max X Y ?Z}
    if X>=Y then Z=X else Z=Y end
  end
  {Max 3 5 C}
end
```

This is much more readable but it means exactly the same as the verbose version. We have added the following three short-cuts:

- Declaring more than one variable in a **local** declaration. This is translated into nested **local** declarations.
- Using “in-line” values instead of variables, e.g., `{P 3}` is a short-cut for **local X in X=3 {P X} end**.
- Using nested operations, e.g., putting the operation `X>=Y` in place of the boolean in the **if** statement.

We will use these short-cuts in all examples from now on.

Let us now execute statement  $\langle s \rangle$ . For clarity, we omit some of the intermediate steps.

1. The initial execution state is:

$$([(\langle s \rangle, \phi)], \phi)$$



Both the environment and the store are empty ( $E = \phi$  and  $\sigma = \phi$ ).

2. After executing the four **local** declarations, we get:

$$( [\langle s \rangle_1, \{\text{Max} \rightarrow m, A \rightarrow a, B \rightarrow b, C \rightarrow c\}], \\ \{m, a, b, c\} )$$

The store contains the four variables  $m$ ,  $a$ ,  $b$ , and  $c$ . The environment of  $\langle s \rangle_1$  has mappings to these variables.

3. After executing the bindings of **Max**, **A**, and **B**, we get:

$$( [\{\text{Max } A \ B \ C\}, \{\text{Max} \rightarrow m, A \rightarrow a, B \rightarrow b, C \rightarrow c\}], \\ \{m = (\text{proc } \{\$ \ X \ Y \ Z\} \langle s \rangle_3 \text{ end}, \phi), a = 3, b = 5, c\} )$$

The variables  $m$ ,  $a$ , and  $b$  are now bound to values. The procedure is ready to be called. Notice that the contextual environment of **Max** is empty because it has no free identifiers.

4. After executing the procedure application, we get:

$$( [\langle s \rangle_3, \{X \rightarrow a, Y \rightarrow b, Z \rightarrow c\}], \\ \{m = (\text{proc } \{\$ \ X \ Y \ Z\} \langle s \rangle_3 \text{ end}, \phi), a = 3, b = 5, c\} )$$

The environment of  $\langle s \rangle_3$  now has mappings from the new identifiers **X**, **Y**, and **Z**.

5. After executing the comparison  $X \geq Y$ , we get:

$$( [\langle s \rangle_4, \{X \rightarrow a, Y \rightarrow b, Z \rightarrow c, T \rightarrow t\}], \\ \{m = (\text{proc } \{\$ \ X \ Y \ Z\} \langle s \rangle_3 \text{ end}, \phi), a = 3, b = 5, c, t = \text{false}\} )$$

This adds the new identifier **T** and its variable  $t$  bound to **false**.

6. Execution is complete after statement  $\langle s \rangle_4$  (the conditional):

$$( [], \{m = (\text{proc } \{\$ \ X \ Y \ Z\} \langle s \rangle_3 \text{ end}, \phi), a = 3, b = 5, c = 5, t = \text{false}\} )$$

The statement stack is empty and  $c$  is bound to 5.

### Procedure with external references (part 1)

The second example defines and calls the procedure **LowerBound**, which ensures that a number will never go below a given lower bound. The example is interesting because **LowerBound** has an external reference. Let us see how the following code executes:

```

local LowerBound Y C in
  Y=5
  proc {LowerBound X ?Z}
    if X>=Y then Z=X else Z=Y end
  end
  {LowerBound 3 C}
end

```

This is very close to the `Max` example. The body of `LowerBound` is identical to the body of `Max`. The only difference is that `LowerBound` has an external reference. The procedure value is:

( **proc** { $\$$  X Z} **if** X>=Y **then** Z=X **else** Z=Y **end end**, {Y  $\rightarrow$  y} )

where the store contains:

$y = 5$

When the procedure is defined, i.e., when the procedure value is created, the environment has to contain a mapping of `Y`. Now let us apply this procedure. We assume that the procedure is called as {LowerBound A C}, where A is bound to 3. Before the application we have:

( [( {LowerBound A C}, {Y  $\rightarrow$  y, LowerBound  $\rightarrow$  lb, A  $\rightarrow$  a, C  $\rightarrow$  c} )],  
 { lb = ( **proc** { $\$$  X Z} **if** X>=Y **then** Z=X **else** Z=Y **end end**, {Y  $\rightarrow$  y} ),  
 y = 5, a = 3, c } )

After the application we get:

( [( **if** X>=Y **then** Z=X **else** Z=Y **end**, {Y  $\rightarrow$  y, X  $\rightarrow$  a, Z  $\rightarrow$  c} )],  
 { lb = ( **proc** { $\$$  X Z} **if** X>=Y **then** Z=X **else** Z=Y **end end**, {Y  $\rightarrow$  y} ),  
 y = 5, a = 3, c } )

The new environment is calculated by starting with the contextual environment ({Y  $\rightarrow$  y} in the procedure value) and adding mappings from the formal arguments X and Z to the actual arguments *a* and *c*.

### Procedure with external references (part 2)

In the above execution, the identifier `Y` refers to *y* in both the calling environment as well as the contextual environment of `LowerBound`. How would the execution change if the following statement were executed instead of {LowerBound 3 C}:

```

local Y in
  Y=10
  {LowerBound 3 C}
end

```

Here `Y` no longer refers to *y* in the calling environment. Before looking at the answer, please put down the book, take a piece of paper, and work it out. Just before the application we have almost the same situation as before:

$$\begin{aligned}
 & ( [ ( \{ \text{LowerBound } A \ C \}, \{ Y \rightarrow y', \text{LowerBound} \rightarrow lb, A \rightarrow a, C \rightarrow c \} ) ], \\
 & \{ lb = (\text{proc } \{ \$ X \ Z \} \text{ if } X \geq Y \text{ then } Z=X \text{ else } Z=Y \text{ end end}, \{ Y \rightarrow y \}), \\
 & y' = 10, y = 5, a = 3, c \} )
 \end{aligned}$$

The calling environment has changed slightly:  $Y$  refers to a new variable  $y'$ , which is bound to 10. When doing the application, the new environment is calculated in exactly the same way as before, starting from the contextual environment and adding the formal arguments. This means that the  $y'$  is ignored! We get exactly the same situation as before in the semantic stack:

$$\begin{aligned}
 & ( [ ( \text{if } X \geq Y \text{ then } Z=X \text{ else } Z=Y \text{ end}, \{ Y \rightarrow y, X \rightarrow a, Z \rightarrow c \} ) ], \\
 & \{ lb = (\text{proc } \{ \$ X \ Z \} \text{ if } X \geq Y \text{ then } Z=X \text{ else } Z=Y \text{ end end}, \{ Y \rightarrow y \}), \\
 & y' = 10, y = 5, a = 3, c \} )
 \end{aligned}$$

The store still has the binding  $y' = 10$ . But  $y'$  is not referenced by the semantic stack, so this binding makes no difference to the execution.

### 2.4.6 Last call optimization

Consider a recursive procedure with just one recursive call which happens to be the last call in the procedure body. We call such a procedure *tail-recursive*. Our abstract machine executes a tail-recursive procedure with a constant stack size. This is because our abstract machine does *last call optimization*. This is sometimes called *tail recursion optimization*, but the latter terminology is less precise since the optimization works for any last call, not just tail-recursive calls (see Exercises). Consider the following procedure:

```

proc {Loop10 I}
  if I==10 then skip
  else
    {Browse I}
    {Loop10 I+1}
  end
end

```

Calling  $\{\text{Loop10 } 0\}$  displays successive integers from 0 up to 9. Let us see how this procedure executes.

- The initial execution state is:

$$\begin{aligned}
 & ( [ ( \{ \text{Loop10 } 0 \}, E_0 ) ], \\
 & \sigma )
 \end{aligned}$$

where  $E_0$  is the environment at the call.

- After executing the **if** statement, this becomes:

$$\begin{aligned}
 & ( [ ( \{ \text{Browse } I \}, \{ I \rightarrow i_0 \} ) ( \{ \text{Loop10 } I+1 \}, \{ I \rightarrow i_0 \} ) ], \\
 & \{ i_0 = 0 \} \cup \sigma )
 \end{aligned}$$

- After executing the `Browse`, we get to the first recursive call:

$$(\{ \text{Loop10 } I+1 \}, \{ I \rightarrow i_0 \}), \\ \{ i_0 = 0 \} \cup \sigma)$$

- After executing the `if` statement in the recursive call, this becomes:

$$(\{ \text{Browse } I \}, \{ I \rightarrow i_1 \}) (\{ \text{Loop10 } I+1 \}, \{ I \rightarrow i_1 \}), \\ \{ i_0 = 0, i_1 = 1 \} \cup \sigma)$$

- After executing the `Browse` again, we get to the second recursive call:

$$(\{ \text{Loop10 } I+1 \}, \{ I \rightarrow i_1 \}), \\ \{ i_0 = 0, i_1 = 1 \} \cup \sigma)$$

It is clear that the stack at the  $k$ th recursive call is always of the form:

$$(\{ \text{Loop10 } I+1 \}, \{ I \rightarrow i_{k-1} \})$$

There is just one semantic statement and its environment is of constant size. This is the last call optimization. This shows the efficient way to program loops in the declarative model: the loop should be invoked through a last call.

### 2.4.7 Active memory and memory management

In the `Loop10` example, the semantic stack and the store have very different behaviors. The semantic stack is bounded by a constant size. On the other hand, the store grows bigger at each call. At the  $k$ th recursive call, the store has the form:

$$\{ i_0 = 0, i_1 = 1, \dots, i_{k-1} = k - 1 \} \cup \sigma$$

Let us see why this growth is not a problem in practice. Look carefully at the semantic stack. The variables  $\{ i_0, i_1, \dots, i_{k-2} \}$  are not needed for executing this call. The only variable needed is  $i_{k-1}$ . Removing the not-needed variables gives a smaller store:

$$\{ i_{k-1} = k - 1 \} \cup \sigma$$

Executing with this smaller store gives exactly the same results as before!

From the semantics it follows that a running program needs only the information in the semantic stack and in the part of the store reachable from the semantic stack. A partial value is *reachable* if it is referenced by a statement on the semantic stack or by another reachable partial value. The semantic stack and the reachable part of the store are together called the *active memory*. The rest of the store can safely be reclaimed, i.e., the memory it uses can be reused for other purposes. Since the active memory size of the `Loop10` example is bounded by a small constant, it can loop indefinitely without exhausting system memory.

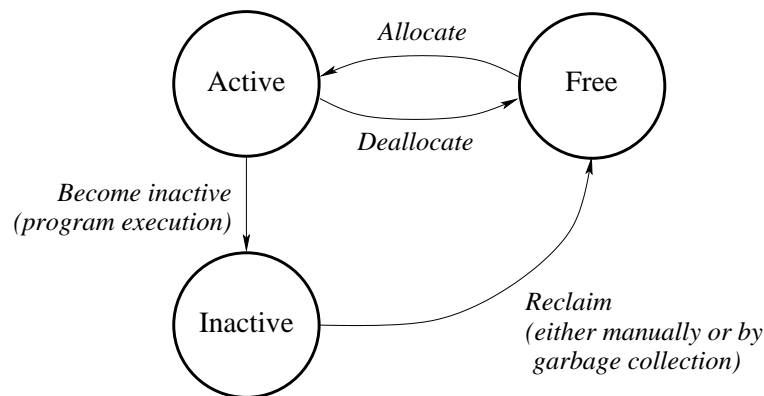


Figure 2.18: Lifecycle of a memory block

### Memory use cycle

Memory consists of a sequence of *words*. This sequence is divided up into *blocks*, where a block consists of a sequence of one or more words used to store a language entity or part of a language entity. Blocks are the basic unit of memory allocation. Figure 2.18 shows the lifecycle of a memory block. Each block of memory continuously cycles through three states: *active*, *inactive*, and *free*. *Memory management* is the task of making sure that memory circulates correctly along this cycle. A running program that needs a block of memory will allocate it from a pool of *free* memory blocks. During its execution, a running program may no longer need some of the memory it allocated:

- If it can determine this directly, then it deallocates this memory. This makes it immediately become free again. This is what happens with the semantic stack in the `Loop10` example.
- If it cannot determine this directly, then the memory becomes *inactive*. It is simply no longer reachable by the running program. This is what happens with the store in the `Loop10` example.

Usually, memory used for managing control flow (the semantic stack) can be deallocated and memory used for data structures (the store) becomes inactive.

Inactive memory must eventually be reclaimed, i.e., the system must recognize that it is inactive and put it back in the pool of free memory. Otherwise, the system has a *memory leak* and will soon run out of memory. Reclaiming inactive memory is the hardest part of memory management, because recognizing that memory is unreachable is a global condition. It depends on the whole execution state of the running program. Low-level languages like C or C++ often leave reclaiming to the programmer, which is a major source of program errors. There are two kinds of program error that can occur:

- *Dangling reference*. This happens when a block is reclaimed even though it is still reachable. The system will eventually reuse this block. This means

that data structures will be corrupted in unpredictable ways, causing the program to crash. This error is especially pernicious since the effect (the crash) is usually very far away from the cause (the incorrect reclaiming). This makes dangling references hard to debug.

- *Memory leak.* This happens when an unreachable block is considered as still reachable, and so is not reclaimed. The effect is that active memory size keeps growing indefinitely until eventually the system's memory resources are exhausted. Memory leaks are less dangerous than dangling references because programs can continue running for some time before the error forces them to stop. Long-lived programs, such as operating systems and servers, must not have any memory leaks.

### Garbage collection

Many high-level languages, such as Erlang, Haskell, Java, Lisp, Prolog, Smalltalk, and so forth, do *automatic* reclaiming. That is, reclaiming is done by the system independently of the running program. This completely eliminates dangling references and greatly reduces memory leaks. This relieves the programmer of most of the difficulties of manual memory management. Automatic reclaiming is called *garbage collection*. Garbage collection is a well-known technique that has been used for a long time. It was used in the 1960's for early Lisp systems. Until the 1990's, mainstream languages did not use it because it was incorrectly judged as being too inefficient. It has finally become acceptable in mainstream programming because of the popularity of the Java language.

A typical garbage collector has two phases. In the first phase, it determines what the active memory is. It does this finding all data structures that are reachable starting from an initial set of pointers called the *root set*. The root set is the set of pointers that are always needed by the program. In the abstract machine defined so far, the root set is simply the semantic stack. In general, the root set includes all pointers in ready threads and all pointers in operating system data structures. We will see this when we extend the machine to implement the new concepts introduced in later chapters. The root set also includes some pointers related to distributed programming (namely references from remote sites; see Chapter 11).

In the second phase, the garbage collector compacts the memory. That is, it collects all the active memory blocks into one contiguous block (a block without holes) and the free memory blocks into one contiguous block.

Modern garbage collection algorithms are efficient enough that most applications can use them with only small memory and time penalties [95]. The most widely-used garbage collectors run in a "batch" mode, i.e., they are dormant most of the time and run only when the total amount of active and inactive memory reaches a predefined threshold. While the garbage collector runs, the program does not fulfill its task. This is perceived as an occasional pause in program execution. Usually this pause is small enough not to be disruptive.