6 Parameter Passing

In the last chapter we explored the semantics of language features that are found in most programming languages, though sometimes in restricted forms. In this chapter we explore a number of semantic variations that are commonly found in programming languages. Once more, by modifying interpreters we are able to express semantic alternatives precisely and in a manner that highlights their essential differences.

In this chapter we explore several areas of language design. In section 6.1 we look at two models of arrays and similar data structures. In sections 6.2–6.5 we study different mechanisms for the transmission of parameters to procedures. Along the way, in section 6.4, we look at some of the implications of our choice of expressed and denoted values. Finally, in section 6.6 we explore optional parameters and keyword arguments.

6.1 Adding Arrays

Most programming languages include data structures composed of multiple elements, such as arrays and records. Such structures are called *aggregates*. The introduction of aggregates brings with it new choices in language design.

The primitive aggregate data types of Scheme are vectors, pairs, and strings. Values of these types are typically represented by consecutive memory locations that contain their elements. When an aggregate value is passed to a procedure, what is passed is a pointer to the first memory location of the structure. We call this an *indirect* representation, since the values of aggregate elements are obtained indirectly, by reference to the aggregate's pointer. Assignment to a binding containing an indirect aggregate simply changes the pointer to the aggregate. In a *direct* aggregate representation, such an assignment affects the aggregate elements directly.

To illustrate these ideas, we add arrays to the defined language. To support arrays, we extend the language with special forms for creating array bindings, accessing array elements, and assigning to array elements. We use the following concrete and abstract syntax:

⟨form⟩	::= `definearray` ⟨var⟩ ⟨exp⟩	`definearray (var len-exp)`
⟨exp⟩	::= `letarray` ⟨arraydecls⟩ `in` ⟨exp⟩	`letarray (arraydecls body)`
	| ⟨array-exp⟩ `[`⟨exp⟩`]`	`arrayref (array index)`
	| ⟨array-exp⟩ `[`⟨exp⟩`] :=` ⟨exp⟩	`arrayassign (array index exp)`
⟨array-exp⟩	::= ⟨varref⟩ | `(`⟨exp⟩`)`	
⟨arraydecls⟩	::= ⟨arraydecl⟩ {`;`⟨arraydecl⟩}*	
⟨arraydecl⟩	::= ⟨var⟩ `[`⟨exp⟩`]`	`decl (var exp)`

An array is a sequence of cells containing expressed values. Expressed values include arrays and denoted values are still simply cells containing expressed values as in section 5.5.

$$\begin{aligned} \text{Array} &= \{\text{Cell(Expressed Value)}\}^* \\ \text{Expressed Value} &= \text{Number} + \text{Procedure} + \text{Array} \\ \text{Denoted Value} &= \text{Cell (Expressed Value)} \end{aligned}$$

The asterisk is meant to suggest the Kleene star (section 2.1). For our Scheme implementation, arrays are represented as vectors whose first element is the tag `*array*`; see figure 6.1.1.

Array indexing in our examples will be zero based, so `a[0]` refers to the first element of array `a`. For example, with indirect arrays we obtain

```
--> define p = proc (b) b[0] := 3;
--> letarray a[2] in begin a[0] := 1; a[1] := 2; p(a); a[0] end;
3
```

Figure 6.1.2 shows the core of an interpreter for handling arrays. In *eval-exp*, the `letarray`, `arrayref`, and `arrayassign` cases are, of course, new. The procedure *apply-proc* now expects *args* to be a list of denoted values, so it is the responsibility of *eval-rands*, not *apply-proc*, to call *expressed->denoted* (as in exercise 5.5.3.) The procedure *eval-rands* now maps *eval-rand* across the operands. We introduce *eval-rator*, whose definition starts out as *eval-exp*. The other cases are the same as in chapter 5.

Figure 6.1.1 Array ADT

```
(define make-array
  (lambda (length)
    (let ((array (make-vector (+ length 1))))
      (vector-set! array 0 '*array*)
      array)))

(define array?
  (lambda (x)
    (and (vector? x) (eq? (vector-ref x 0) '*array*))))

(define array-ref
  (lambda (array index)
    (vector-ref array (+ index 1))))

(define array-set!
  (lambda (array index value)
    (vector-set! array (+ index 1) value)))

(define array-whole-set!
  (lambda (dest-array source-array)
    (let ((source-len (vector-length source-array)))
      (if (> source-len (vector-length dest-array))
          (error "Array too long for assignment:" source-array)
          (letrec ((loop (lambda (n)
                           (if (< n source-len)
                               (begin
                                 (vector-set! dest-array n (vector-ref source-array n))
                                 (loop (+ n 1)))))))
            (loop 1))))))

(define array-copy
  (lambda (array)
    (let ((new-array (make-array (- (vector-length array) 1))))
      (array-whole-set! new-array array)
      new-array)))
```

Figure 6.1.2 Interpreter for illustrating parameter-passing variations

```
(define eval-exp
  (lambda (exp env)
    (variant-case exp
      (varref (var) (denoted->expressed (apply-env env var)))
      (app (rator rands)
        (apply-proc (eval-rator rator env) (eval-rands rands env)))
      (varassign (var exp)
        (denoted-value-assign! (apply-env env var) (eval-exp exp env)))
      (letarray (arraydecls body)
        (eval-exp body
          (extend-env (map decl->var arraydecls)
            (map (lambda (decl)
                   (do-letarray (eval-exp (decl->exp decl) env)))
                 arraydecls)
            env)))
      (arrayref (array index)
        (array-ref (eval-array-exp array env)
          (eval-exp index env)))
      (arrayassign (array index exp)
        (array-set! (eval-array-exp array env)
          (eval-exp index env)
          (eval-exp exp env)))
      ...)))

(define eval-rator
  (lambda (rator env)
    (eval-exp rator env)))

(define eval-rands
  (lambda (rands env)
    (map (lambda (rand) (eval-rand rand env)) rands)))

(define eval-rand
  (lambda (exp env)
    (expressed->denoted (eval-exp exp env))))

(define apply-proc
  (lambda (proc args)
    (variant-case proc
      (prim-proc (prim-op) (apply-prim-op prim-op (map denoted->expressed args)))
      (closure (formals body env) (eval-exp body (extend-env formals args env)))
      (else (error "Invalid procedure:" proc)))))
```

Figure 6.1.3 Auxiliaries for call-by-value with indirect arrays

```
(define denoted->expressed cell-ref)

(define denoted-value-assign! cell-set!)

(define do-letarray (compose make-cell make-array))

(define eval-array-exp eval-exp)

(define expressed->denoted make-cell)
```

Figure 6.1.2 introduces five new procedures that are not defined there. These are *denoted->expressed*, *denoted-value-assign!*, *do-letarray*, *eval-array-exp*, and *expressed->denoted*. By varying the definitions of these procedures, we can model a variety of parameter-passing mechanisms.

For the indirect representation, the code for these auxiliary procedures is shown in figure 6.1.3. To convert a denoted value to an expressed value we take the contents of the cell, and to assign to a denoted value, we use *cell-set!*. The procedure *do-letarray* first makes an array and then places it in a new cell. Since arrays are expressed values, *eval-array-exp* is simply *eval-exp*. Last, *expressed->denoted* just creates a cell containing the expressed value.

• *Exercise 6.1.1*
Implement this interpreter and run some examples. □

Scheme and C both use the indirect model of arrays. Some other languages, such as Pascal, use the *direct* model, which avoids indirect references to arrays. Arrays are represented directly as denoted values and array elements may not contain other arrays. In terms of our defined language, this means

$$\begin{aligned} \text{Array} &= \{\text{Cell}(\text{Number} + \text{Procedure})\}^* \\ \text{Expressed Value} &= \text{Number} + \text{Procedure} + \text{Array} \\ \text{Denoted Value} &= \text{Cell (Number)} + \text{Cell (Procedure)} + \text{Array} \end{aligned}$$

In the direct model, when an array is passed by value, the formal parameter is bound to a *copy* of the sequence of values in the original array. Then if the procedure performs array assignment, only the local copy of the array is modified. If assignment to a destination containing an array is allowed, the assigned value must be an array and its elements are copied into the

Figure 6.1.4 Auxiliaries for call-by-value with direct arrays

```
(define denoted->expressed
  (lambda (den-val)
    (if (array? den-val) den-val (cell-ref den-val))))

(define denoted-value-assign!
  (lambda (den-val exp-val)
    (cond
      ((not (array? den-val)) (cell-set! den-val exp-val))
      ((array? exp-val) (array-whole-set! den-val exp-val))
      (else (error "Must assign array:" den-val)))))

(define do-letarray make-array)

(define eval-array-exp eval-exp)

(define expressed->denoted
  (lambda (exp-val)
    (if (array? exp-val) (array-copy exp-val) (make-cell exp-val))))
```

destination array. (We arbitrarily require the destination array to be at least as long as the source array.)

Modeling direct arrays in Scheme is somewhat tricky because Scheme uses the indirect model. The simplest way to do this is to model an array with a vector, as in the indirect case. See figure 6.1.4. The procedure *denoted->expressed* now dereferences a cell only when the denoted value is not an array, while *denoted-value-assign!* performs a cell assignment or a whole array assignment as appropriate to the type of denoted value. The procedure *do-letarray* now simply makes an array, rather than making an array and putting it in a cell. The procedure *eval-array-exp* remains as before. Finally, if an expressed value is an array, *expressed->denoted* copies it, otherwise as before it makes a cell containing the value. Finally, we modify the array ADT to prohibit assignment of an array to an array element.

```
(define array-set!
  (lambda (array index value)
    (if (array? value)
        (error "Cannot assign array to array element:" value)
        (vector-set! array (+ index 1) value))))
```

Figure 6.1.5 Example illustrating direct and indirect array models

```
letarray u[3]; v[2]
in begin
     u[0] := 5; u[1] := 6; u[2] := 4; v[0] := 3; v[1] := 8;
     let p = proc (x)
               begin
                 x[1] := 7; x := v; x[1] := 9
               end
     in p(u)
   end
```

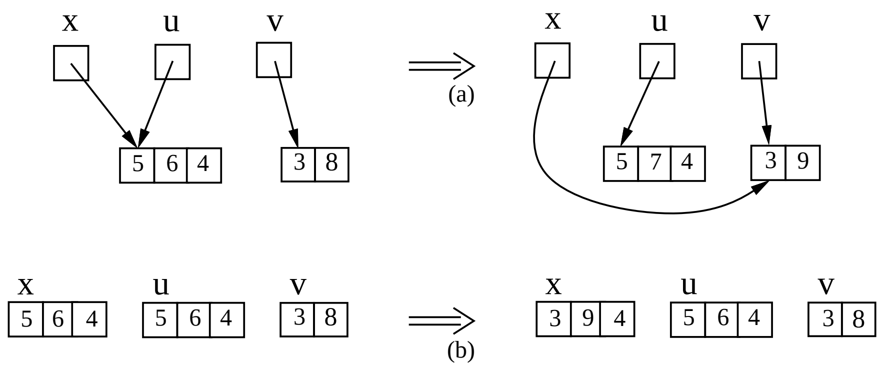

Figure 6.1.6 Effect of figure 6.1.5 using call-by-value

The difference between indirect and direct models of arrays is illustrated by figure 6.1.5. For the indirect version, `x` and `u` are both bound to cells that point at the first location of an array containing 5, 6, and 4 and `v` is bound to a cell that points to the first location of an array containing 3 and 8. Executing the assignment `x[1] := 7` changes the 6 to a 7. Then executing `x := v` changes the `x` cell to point at the same array as `v` does. Finally, executing `x[1] := 9` then changes the 8 to a 9. See figure 6.1.6 (a). For the direct version, `u` is bound to a sequence of cells containing its elements and `x` is bound to a sequence of cells containing copies of the elements of `u`. Executing `x[1] := 7` changes the 6 in `x`'s array to a 7. The assignment `x := v` replaces the contents of `x`'s first two elements by a copy of `v`'s array. Once again, executing `x[1] := 9` changes the 8 to a 9, but this time `v` remains unchanged; see figure 6.1.6 (b).

• *Exercise 6.1.2*
Implement the direct array interpreter and run some examples. □

• *Exercise 6.1.3*
Most languages that support direct arrays forbid assignment of one array to another or arrays to be expressed values (section 6.4). Modify your interpreter accordingly to enforce these restrictions. Among other changes, modify *eval-rand* so that an operand may be a variable referring to an array. □

6.2 Call-by-Reference

When an operand is a variable and the called procedure assigns to the corresponding formal parameter, is this visible to the caller as a change in the binding of its variable? The indirect model of arrays provides such visibility for arrays modified by array assignment, but for ordinary variables, with modification by variable assignment, the answer to this question has been no. If an assignment is made to a procedure's call-by-value parameter, it affects only the binding made when the procedure was called. For example, in

```
let p = proc (x) x := 5
in let a = 3;
       b = 4
   in begin
        p(a);
        p(b);
        +(a, b)
      end
```

on each call to procedure *p*, the variable x is bound to a fresh cell. Therefore the assignment to x does not affect a or *b*, and so the expression yields 7. The effect of an assignment is always restricted to the scope of the assigned variable. This is generally desirable, especially given the difficulties inherent in understanding assignment. It is clear what code needs to be examined to understand the effect of an assignment.

There are times, however, when it is necessary or convenient for a procedure to be capable of assigning a binding passed by its caller, even though the procedure may not be defined within the scope of the associated variable. For example, it may be intended above that a and b both be set to 5, in which case the expression should yield 10.

This may be accomplished by a different parameter-passing technique in which the arguments passed to p are the bindings of a and b, not their corresponding expressed values. This form of parameter passing is known as *call-by-reference.*

Call-by-reference allows us to write some procedures that cannot be written using call-by-value. The classic example is a procedure that swaps the contents of its arguments. For example, using call-by-reference we can write

```
--> define swap =
       proc (x, y)
         let temp = 0
         in begin temp := x; x := y; y := temp end;
--> define c = 3;
--> let b = 4
    in begin
         swap(c, b);
         b
       end;
3
--> c;
4
```

∘ *Exercise 6.2.1*
Why doesn't this work under call-by-value? □

In most languages with call-by-reference, references can be array elements as well as variable bindings. If an array element is passed by reference to a procedure and the procedure assigns to its corresponding formal parameter, the effect is to assign a new value to the array element. Thus *swap* may be used to exchange the values of array elements.

```
--> define b = 2;
--> letarray a[3]
    in begin
         a[1] := 5;
         swap(a[1], b);
         a[1]
       end;
2
--> b;
5
```

In some languages, call-by-reference operands may be expressions other than variable or data structure references (such as applications). In these languages the value of the operand is placed in a new location, and assignments to this location by the called procedure have no effect that is visible to the caller. For example, consider

```
--> define c = 3;
--> define p = proc (x) x := 5;
--> begin
      p(add1(c));
      c
    end;
3
```

Here c refers to a location that initially contains 3. The assignment changes the value of the binding of x from 4 to 5 but does not affect the binding of c. In such cases call-by-value and call-by-reference behave the same way.

More than one call-by-reference parameter may refer to the same location:

```
--> let b = 3;
        p = proc (x, y)
              begin
                x := 4;
                y
              end
    in p(b, b);
4
```

The reason this yields 4 is that x and y both refer to the cell that is the binding of b. This phenomenon is known as *aliasing.* Here x and y are aliases (names) for the same location. Aliasing makes it very difficult to understand programs. Generally, we do not expect an assignment to one variable to change the value of another. Virtually all rules for reasoning formally about programs are invalid in the presence of aliasing.

If references to array elements are passed as arguments, then in general it is impossible to detect aliasing without costly run-time checks. For example, `swap(a[1], a[f(b)])` results in aliasing if and only if `f(b)` yields 1, which may be impossible to predict.

The preceding definition of *swap* happens to work even if its parameters are aliased, but aliasing can provide unpleasant surprises. The following version of swap cleverly avoids the use of a temporary variable by assuming that its arguments are integers.

```
--> define swap2 =
      proc (x, y)
        begin
          x := +(x, y); y := -(x, y); x := -(x, y)
        end;
--> define b = 1;
--> define c = 2;
--> swap2(b, c);
--> b;
2
--> c;
1
--> swap2(b, b);
--> b;
0
```

The first call to *swap2* works correctly. In the second call, however, x and y are aliases for b, so b is assigned $2 + 2 = 4$, and then $4 - 4 = 0$. Clearly *swap2* works only if its arguments are not aliased.

How is call-by-reference to be modeled? The clue is that the denoted values of the caller are the same as the denoted values of the procedure. Thus, if the operand is a variable, we can pass its binding directly to the procedure, rather than copying its contents to a new cell, as we did in call-by-value. So we can obtain a simple version of call-by-reference by changing the production

$$\langle\text{operand}\rangle \ ::= \ \langle\text{exp}\rangle$$

to

$$\langle\text{operand}\rangle \ ::= \ \langle\text{varref}\rangle$$

and changing *eval-rand* to

```
(define eval-rand
  (lambda (rand env)
    (variant-case rand
      (varref (var) (apply-env env var))
      (else (error "Invalid operand:" rand)))))
```

What other forms can operands take? We have seen that when an array reference appears as an operand in call-by-reference, we must pass a reference to the array element, not the value of the array element. In a typical implementation, a reference to an array element is simply a pointer to the element. Since it is not possible in Scheme to obtain a pointer directly to a vector element, we represent array elements by the `ae` record type, which records the array and the index of an element within the array's vector.

```
(define-record ae (array index))
```

So in general, an L-value in the language is either a cell or an array element; in either case the contents of the L-value is an expressed value. We write this as

$$\text{L-value} = \text{Cell (Expressed Value)} + \text{Array Element (Expressed Value)}$$

$$\text{Denoted Value} = \text{L-value}$$

Later we introduce denoted values that are not L-values.

We have explored two possibilities for call-by-reference operands: variables and array references. We could restrict call-by-reference operands to these two forms, but instead we adopt the more common design alternative, suggested earlier in this section, that arbitrary expressions be allowed as call-by-reference operands, with their values passed to the procedure in fresh cells as in call-by-value. Hence we write the grammar for operands as

⟨operand⟩ ::=	⟨varref⟩	
	| ⟨array-exp⟩ `[`⟨exp⟩`]`	`arrayref (array index)`
	| ⟨exp⟩	

To obtain an interpreter for call-by-reference, we start off with the interpreter for call-by-value and modify the auxiliary procedures that deal with operands and denoted values, since those are the ones that change.

If we start with an interpreter for call-by-value with indirect arrays, we get the auxiliary procedures shown in figure 6.2.1. Corresponding to the grammar for operands, *eval-rand* has three cases. If the operand is a variable or an array reference, the corresponding L-value should be passed directly to the procedure; otherwise the expression should be evaluated and copied into a new cell, as for call-by-value.

Since we have changed the set of denoted values (or at least the set of representations of denoted values), we need to change the procedures that deal with denoted values. There are only two of these: *denoted->expressed* and *denoted-value-assign!*. We modify each of them to check what kind of denoted value they are given and to do the right thing in each case. In this

Figure 6.2.1 Auxiliaries for call-by-reference with indirect arrays

```
(define eval-rand
  (lambda (rand env)
    (variant-case rand
      (varref (var) (apply-env env var))
      (arrayref (array index)
        (make-ae (eval-array-exp array env) (eval-exp index env)))
      (else (make-cell (eval-exp rand env))))))

(define denoted->expressed
  (lambda (den-val)
    (cond
      ((cell? den-val) (cell-ref den-val))
      ((ae? den-val) (array-ref (ae->array den-val) (ae->index den-val)))
      (else (error "Can't dereference denoted value:" den-val)))))

(define denoted-value-assign!
  (lambda (den-val val)
    (cond
      ((cell? den-val) (cell-set! den-val val))
      ((ae? den-val) (array-set! (ae->array den-val) (ae->index den-val) val))
      (else (error "Can't assign to denoted value:" den-val)))))
```

interpreter, the error lines in each of these two procedures should never be executed, but they allow for extension later.

The combination of indirect aggregate representation and call-by-reference is used occasionally, as in Modula-2, when passing an aggregate value whose type is "hidden." More commonly, call-by-reference is used in conjunction with direct array representation, as with arrays passed as `var` parameters in Pascal. A procedure that is passed a reference to a directly represented aggregate treats references to the aggregate as if it were represented indirectly. However, assignment to a binding that refers to a direct aggregate results in copying elements of the source aggregate into the destination aggregate, not mutation of the reference to the aggregate.

Figure 6.2.2 shows the effect of executing the program in figure 6.1.5 under both the indirect and direct models. In the indirect model, `x` and `u` are both bound to a cell containing a pointer to the first location in the array containing the sequence 5, 6, and 4. The assignment `x := v` mutates that cell, so that now `u` and `x` both point to the other array. (See figure 6.2.2 (a).) In the

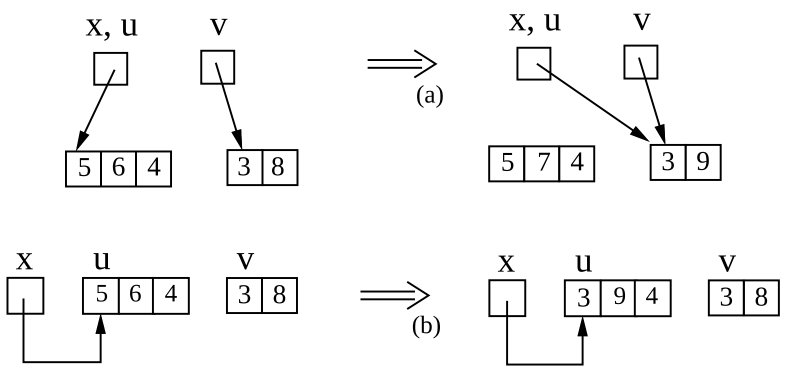

Figure 6.2.2 Effect of figure 6.1.5 using call-by-reference

direct model, u is bound to a sequence of cells containing its elements and x is bound to a cell that refers to the cells of u. For the assignment `x := v`, the contents of the cells of v are copied into the cells of u to which x refers. (See figure 6.2.2 (b).) The final assignment `x[1] := 9` mutates the second cell of the array to which x points.

One advantage of call-by-reference is that it allows a procedure to "return" more than one value by side-effecting some of its parameters. For example, a procedure may take a call-by-reference parameter to which it assigns a status code indicating what sort of exception, if any, it may have encountered. This allows the procedure to return normal results in the usual way without danger of confusion with error codes. There are other ways of returning multiple results using call-by-value (for example, continuation-passing style as in exercise 11.4.4 or variant records as used in exercise 5.5.8).

As we saw in section 6.1, passing structured objects such as arrays by value requires copying the object. For reasons of efficiency structured objects may be passed by reference, even when other arguments are passed by value. Another alternative when call-by-value is used by default is to provide specific declarations that force certain parameters to be passed by reference. If such a declaration is forgotten when passing an array, an inefficient copy may be performed without the programmer's knowledge. Heap-allocated objects, including Scheme's cons cells and vectors, are always accessed via references.

Figure 6.2.3 Program for exercise 6.2.2

```
letarray u[3]; v[2]
in begin
     u[0] := 5; u[1] := 6; u[2] := 4; v[0] := 3; v[1] := 8;
     let p = proc (x, y)
               begin
                 write(y);
                 x[1] := 7;
                 write(y);
                 x := v;
                 x[1] := 9;
                 write(y)
               end
     in begin
          p(u, u[1]);
          write(u[1])
        end
   end
```

• *Exercise 6.2.2*
Using diagrams, as in figure 6.2.2, trace the program in figure 6.2.3 using call-by-reference with indirect arrays and direct arrays. What value is printed in each case? □

• *Exercise 6.2.3*
Implement the interpreter of this section, which uses call-by-reference and indirect arrays, using figure 6.2.1. □

• *Exercise 6.2.4*
Modify the interpreter of this section to support call-by-reference with direct arrays. What does it mean to pass an array element by reference with the direct model? □

6.3 Call-by-Value-Result and Call-by-Result

It is often possible for compilers to allocate memory cells for call-by-value parameters of a procedure in locations that the procedure's code can access directly. This contrasts with call-by-reference parameters, the locations of which are not known until the procedure is called. This usually makes variable references less efficient when call-by-reference is used. The *call-by-value-result* parameter-passing technique combines the variable reference efficiency of call-by-value with the ability, provided by call-by-reference, to return information to the caller through parameters. The trick is to pass a pointer to the caller's cell, as in call-by-reference, and then to copy the contents of the cell in a location local to the called procedure. The called procedure then uses the local cell just as it would with call-by-value. Then, just before the procedure returns, the contents of the local cell is copied back into the caller's cell.

Call-by-value-result achieves more efficient variable reference than call-by-reference at the expense of less efficient procedure call and return. In this respect it is superior only if variables are referenced many times in a typical call, as in a loop. Even in such cases, a compiler may be able to eliminate the indirection of call-by-reference using optimization techniques that are beyond the scope of this book.

There may, however, be other reasons to prefer call-by-value-result over call-by-reference. One is that call-by-value-result does not suffer from the aliasing problems of call-by-reference. For example, *swap2* is correct if parameters are passed by value-result. Given the difficulties that aliasing poses for formal proof and other approaches to reasoning carefully about programs, call-by-value-result may be preferred when attempting to write highly reliable programs.

In the absence of aliasing, call-by-value-result has the same effect as call-by-reference. Therefore, some language specifications allow the use of either call-by-reference or call-by-value-result. The compiler is then free to choose for each parameter the passing technique that is optimum under the circumstances of the call. The danger is that if aliasing does occur, the results are unpredictable. This may have serious consequences. For example, though a program may at times alias two arguments, it might still work correctly because the compiler happens to choose call-by-value-result for one or both of the arguments. If the program is then compiled using a different compiler, or even using the same compiler after the program has been changed in ways that affect optimization of the call, call-by-reference may be used for both arguments. This exposes an error in a program that may have performed correctly for some time. To make matters worse, the error may not actually cause damage for a long time, since the arguments may be array elements that are aliased only under run time conditions that rarely occur.

Call-by-reference and call-by-value-result allow information to be passed both to and from a called procedure. The *call-by-result* parameter-passing technique is appropriate when a parameter is used only to pass information from a procedure to its caller. It is simply a variation on call-by-value-result in which the local L-value of a parameter is uninitialized. It may be used, for example, to return an error code to the caller.

Call-by-value, call-by-value-result, and call-by-result are sometimes known collectively as *call-by-copy,* since they all involve making a copy of a variable binding when the operand is a variable reference.

• *Exercise 6.3.1*
Modify the parameter-passing of the call-by-reference with indirect arrays interpreter so that parameters are passed by value-result. □

• *Exercise 6.3.2*
Write a program (without using *swap2*) in which call-by-value, call-by-reference, and call-by-value-result all yield different results. □

6.4 Expressed or Denoted Values?

The languages we have developed follow the tradition of Scheme and similar languages in that they have a rich set of expressed values. This is important, because the primary way in which computed information is returned is as the value of a procedure: an expressed value.

In traditional imperative languages, however, expressions often occur on the right-hand side of assignment statements, so the set of expressed values must correspond to the set of values that are *storable* in a memory location. Many languages were designed with the intention that they be implemented using a stack-based run-time architecture. As we shall see in section 10.4, when such an architecture is used, storing procedures or references to arrays may result in dangling pointer problems. Also, storing arrays directly involves copying. To avoid these problems, the set of expressed values is often restricted, even when the set of denoted values is quite rich. In Pascal, for example, the expressed values are scalars, such as integers and characters, but the denoted values include arrays and procedures. We shall now see how this change in perspective may be reflected by interpreters.

The next language has the simplest possible set of expressed values:

$$\text{Expressed Value} = \text{Number}$$

but we make the set of denoted values much richer:

$$\text{Denoted Value} = \text{L-value} + \text{Array} + \text{Procedure}$$

So arrays and procedures are denoted values but not expressed values. This does not cause much problem for arrays, since arrays are already introduced with `letarray`, but procedures are more complicated. The fact that procedures are expressed values is built into our syntax and interpreters. Procedures are created as expressed values by `proc`, and retrieved as expressed values when *`eval-exp`* evaluates the operator portion of an application.

Thus our first task is to change the language's syntax. Since a procedure (or array) can exist only as the binding of a variable, we make ⟨operator⟩ (or ⟨array-exp⟩) be a variable. In addition, we need to introduce a form, like `let`, that allows us to introduce variables, without using procedure application. To do this, we delete the productions

⟨exp⟩ ::=	`proc` ⟨varlist⟩ ⟨exp⟩	`proc (formals body)`
⟨operator⟩ ::=	⟨varref⟩ | `(`⟨exp⟩`)`	
⟨array-exp⟩ ::=	⟨varref⟩ | `(`⟨exp⟩`)`	

and replace them by the productions

⟨exp⟩ ::=	`letproc` ⟨procdecls⟩ `in` ⟨exp⟩	`letproc (procdecls body)`
	| `local` ⟨decls⟩ `in` ⟨exp⟩	`local (decls body)`
⟨operator⟩ ::=	⟨varref⟩	
⟨array-exp⟩ ::=	⟨varref⟩	

In order to have something interesting to pass to these procedures, we use call-by-reference. Our starting point is the call-by-reference interpreter of section 6.2, shown in figure 6.1.2 and figure 6.2.1. Figure 6.4.1 shows the resulting interpreter. There are only two changes in the code of figure 6.4.1. First, we replace *`eval-rator`*, which was *`eval-exp`*, with code that uses *`apply-env`*, since the `rator` is known to be a variable reference.

The second change is the new construction `letproc`. The code for `letproc` builds a list of closures, and then evaluates the body in a new environment

Figure 6.4.1 Interpreter with denoted procedures and arrays

```
(define eval-exp
  (lambda (exp env)
    (variant-case exp
      (letproc (procdecls body)
        (let ((vars (map procdecl->var procdecls))
              (closures (map (lambda (decl)
                                (make-closure
                                  (procdecl->formals decl)
                                  (procdecl->body decl)
                                  env))
                              procdecls)))
          (let ((new-env (extend-env vars closures env)))
            (eval-exp body new-env))))
      (local (decls body)
        (let ((vars (map decl->var decls))
              (exps (map decl->exp decls)))
          (let ((new-env (extend-env vars
                           (map (lambda (exp)
                                  (make-cell (eval-exp exp env)))
                                exps)
                           env)))
            (eval-exp body new-env))))
      ...)))

(define eval-rator
  (lambda (rator env)
    (let ((den-val (apply-env env (varref->var rator))))
      (if (closure? den-val)
          den-val
          (denoted->expressed den-val)))))
```

in which each of the names is bound to the corresponding closure. See figure 6.4.1.

Since the set of denoted values has changed, we also need to review the auxiliary procedures introduced in section 6.1, which manipulate denoted values. The results are shown in figure 6.4.2. The procedures *denoted->expressed* and *denoted-value-assign!* can be the same as for the call-by-reference case, since they already contain checks to make sure that only legal L-values are dereferenced or mutated. Similarly, *expressed->denoted* and *eval-rand*

Figure 6.4.2 Procedures for call-by-reference with denoted indirect arrays

```
(define denoted->expressed
  (lambda (den-val)
    (cond
      ((cell? den-val) (cell-ref den-val))
      ((ae? den-val) (array-ref (ae->array den-val) (ae->index den-val)))
      (else (error "Can't dereference denoted value:" den-val)))))

(define denoted-value-assign!
  (lambda (den-val val)
    (cond
      ((cell? den-val) (cell-set! den-val val))
      ((ae? den-val) (array-set! (ae->array den-val) (ae->index den-val) val))
      (else (error "Can't assign to denoted value:" den-val)))))

(define do-letarray make-array)

(define eval-array-exp
  (lambda (array-exp env)
    (apply-env env (varref->var array-exp))))
```

remain unchanged. In `letarray`, the variables are bound directly to the arrays, rather than to cells pointing to the arrays, so *do-letarray* need not call *make-cell*. Since array expressions must be variables, *eval-array-exp* can be simplified to use *apply-env*. This completes the interpreter.

Typically, languages that use a stack as their basic run-time structure (see section 10.4) allocate their environments on the stack, so denoted values are restricted to values that are storable on the stack. These typically include single-word quantities, such as small integers and pointers, but may also include quantities that take up several words on the stack. In Pascal, for example, even arrays are allocated on the stack, and are copied when they are passed by value, an expensive operation. Denoted values typically include cells, since these are represented by pointers and pointers are easily stored on the stack. In section 10.4, we study the issue of stack allocation and the representation of values on the stack in more detail.

Imperative languages typically equate expressed values with those values that are storable in a memory cell. Therefore the set of expressed values is restricted to those that fit into a single machine word, such as small integers and pointers. Languages vary in the specification of what values are storable, particularly when those values are pointers. In Pascal, for example, pointers refer to heap-allocated data structures, similar to cons cells in Scheme. C, on the other hand, allows pointers to almost any type of value, including procedures.

Scheme has a rich set of expressed values, as do a number of other functional and almost-functional languages, including ML. Since Scheme uses an indirect model of aggregates, most values are represented as pointers. Cells (L-values) are not expressed values in Scheme. This is reflected in our need to define a `cell` ADT, and in our simulation of array element pointers. Scheme's denoted values are just cells. ML, on the other hand, does not support variable assignment. Instead, assignment is accomplished by explicit operations on cells, just as we have used the cell ADT in our interpreters. Expressed values then include cells. Since there is no variable assignment, there is no reason to require denoted values to differ from expressed values, and therefore denoted and expressed values are the same in ML.

We have only touched upon the study of the value structures of programming languages. Studying the denoted and expressed values of a language provides deep insight into its structure. A language's value structure determines a great deal of its expressive power and affects its efficiency and implementation strategies. For example, when procedures are not expressed values, many functional programming techniques (such as currying) are not possible. On the other hand, when procedures are expressed values, they must at times be heap allocated, which is generally less efficient than stack allocation. Clever compilers are, however, often able to avoid apparent inefficiencies. If, for example, a Scheme procedure is bound directly using `let` and every reference to it is in the operator position of an application, then it could be stack allocated.

• *Exercise 6.4.1*

Implement the interpreter of this section and test it with several programs. □

∘ *Exercise 6.4.2*

The interpreter of this section, like our earlier ones, assumes that the initial environment consists of cells containing primitive procedures. These cells can thus be modified by variable assignment. Modify the interpreter to prevent this by having the initial environment contain procedures rather than cells. □

• *Exercise 6.4.3*
Modify this interpreter to use call-by-value instead of call-by-reference. □

∘ *Exercise 6.4.4*
Modify this interpreter to use a direct, rather than indirect representation of arrays. □

∘ *Exercise 6.4.5*
Modify this interpreter so that `letproc` defines its procedures recursively, as `letrecproc` does. □

∘ *Exercise 6.4.6*
Several times during the course of this chapter we have presented a grammar rule that weakens the expressiveness of the language. For example, ⟨array-exp⟩ was any ⟨varref⟩ or (⟨exp⟩) and now it can be only a ⟨varref⟩. Modify the parser or the interpreter to support these restrictions. □

6.5 Call-by-Name and Call-by-Need

In section 4.3, we saw that lambda calculus expressions may be evaluated using β-reduction. In this model, procedures are called by performing the body of the procedure after the operands are substituted for each reference to the corresponding variable. This substitution may require renaming of bound variables (α-conversion) to avoid capture of variable references that occur free in the operands. Such rewriting techniques, which require manipulation of program text during evaluation, are heavily used in the theoretical study of programming languages. In general, however, they are too inefficient for practical use.

β-reduction does have one characteristic that is sometimes of practical importance and is not shared by the parameter-passing techniques discussed so far: the possibility of delaying (through normal order reduction) the evaluation of operands until their values are needed. This may avoid unnecessary or even nonterminating computation.

The price paid for avoiding non-termination is that if there are repeated references to the same parameter, then the associated operand is reevaluated with each reference. Another subtlety of this evaluation scheme is that the index subexpression of an array reference is not evaluated until the time of assignment. In the following example, the delay of array index evaluation is critical.

```
--> local p = proc (x)
                begin
                  i := 1;
                  x := 2
                end;
          i = 0
      in letarray a[2]
           in begin
                a[0] := 1;
                p(a[i]);
                writeln(a[0], a[1])
              end;
1 2
```

Delayed evaluation and assignment do not combine well, since it is hard to understand the effect of assignment if it is unclear when the assignment will happen. Nevertheless, this combination is supported by some languages, notably Algol 60, and there are times when it is useful. A classic example is Jensen's device (exercise 6.5.4). Nevertheless, parameter-passing mechanisms that use delayed evaluation are used in some programming languages, so they are worth studying.

To delay operand evaluation without program rewriting, it is tempting simply to pass a reference to some representation such as an abstract syntax tree (or compiled code) of the operand. However, it is necessary to prevent free variables in the operand from being captured by declarations in the called procedure. We have already encountered the variable capture problem when creating procedures, and the solution is the same: close the operand in the calling environment. We do this by forming a data structure that includes both some representation of the text of the operand and also the bindings of all variables that occur free in the operand. Closures created to delay evaluation are called *thunks*. For example, in section 4.5.2, we saw how thunks (created as procedures of no arguments) may be used to delay evaluating a stream's tail.

Parameter passing in which argument evaluation is delayed using thunks that are reevaluated with every variable reference is called *call-by-name*. To develop an implementation for a call-by-name interpreter employing indirect representation of arrays, we first identify the expressed and denoted values. We begin with the language of section 6.1. For this language we had

$$\text{Expressed Value} = \text{Number} + \text{Procedure} + \text{Array}$$

What should the denoted values be? As suggested above, these values should include thunks that encapsulate an operand and its environment. But what kind of value should invocation of such a thunk return? Since a formal parameter may appear on the left-hand side of a variable assignment, the only safe answer is that invocation of a thunk must return an L-value. We write this as

$$\text{Thunk} = () \rightarrow \text{L-value}$$

meaning that a thunk is a procedure of no arguments, which returns an L-value when called. We represent a thunk, containing an operand and an environment, using the following record type:

```
(define-record thunk (exp env))
```

As in the case of call-by-reference, the possible L-values include array elements:

$$\text{L-value} = \text{Cell (Expressed Value)} + \text{Array Element (Expressed Value)}$$

Since these L-values are the same as in section 6.2, we can use most of the same auxiliary procedures.

We would also like to have local variables, either scalars or arrays, in procedure bodies. It is unnecessary and inefficient to use the thunk mechanism for these variables, so we can represent them in the usual way. Thus, the denoted values are either L-values or thunks:

$$\text{Denoted Value} = \text{L-value} + \text{Thunk}$$

We can continue to use *eval-exp* from figure 6.1.2, which we extend with `local` in figure 6.5.1. The call-by-name interpreter contains one big difference: *eval-rands* does not evaluate the operands. Instead, it merely packages them in thunks. For each variable operand, the packaging only occurs if it has been declared by `local`. If the variable is a formal parameter of a procedure, then it must already be bound to a thunk, in which case that thunk is used instead of creating a new one.

A thunk is evaluated when the variable to which it is bound is evaluated. Since the body of a thunk is an operand, *eval-rand* is used to evaluate the thunk body. Since invocation of a thunk returns an L-value, we cannot use *eval-exp* here. Instead, we use a grammar for operands that recognizes the special case of an operand that is a variable or array reference. In this case we can pass the corresponding L-value directly. This is the same situation as in call-by-reference.

Figure 6.5.1 Call-by-name interpreter

```
(define eval-exp
  (lambda (exp env)
    (variant-case exp
      (local (decls body)
        (let ((vars (map decl->var decls))
              (exps (map decl->exp decls)))
          (let ((new-env (extend-env vars
                           (map (lambda (exp)
                                  (make-cell (eval-exp exp env)))
                                exps)
                           env)))
            (eval-exp body new-env))))
      (app (rator rands) ...)
      (varref (var) ...)
      (varassign (var exp) ...)
      (letarray (arraydecls body) ...)
      (arrayref (array index) ...)
      (arrayassign (array index exp) ...)
      ...)))

(define eval-rands
  (lambda (rands env)
    (map (lambda (rand)
           (variant-case rand
             (varref (var)
               (let ((den-val (apply-env env var)))
                 (if (thunk? den-val)
                     den-val
                     (make-thunk rand env))))
             (else (make-thunk rand env))))
         rands)))
```

Figure 6.5.2 Auxiliaries for call-by-name interpreter

```
(define eval-rand
  (lambda (rand env)
    (variant-case rand
      (varref (var) (apply-env env var))
      (arrayref (array index) (make-ae (eval-array-exp array env) (eval-exp index env)))
      (else (make-cell (eval-exp rand env))))))

(define denoted->L-value
  (lambda (den-val)
    (if (thunk? den-val)
        (eval-rand (thunk->exp den-val) (thunk->env den-val))
        den-val)))

(define denoted->expressed
  (lambda (den-val)
    (let ((l-val (denoted->L-value den-val)))
      (cond
        ((cell? l-val) (cell-ref l-val))
        ((ae? l-val) (array-ref (ae->array l-val) (ae->index l-val)))
        (else (error "Can't dereference denoted value:" l-val))))))

(define denoted-value-assign!
  (lambda (den-val exp-val)
    (let ((l-val (denoted->L-value den-val)))
      (cond
        ((cell? l-val) (cell-set! l-val exp-val))
        ((ae? l-val) (array-set! (ae->array l-val) (ae->index l-val) exp-val))
        (else (error "Can't assign to denoted value:" l-val))))))
```

We write the grammar for operands as

⟨operand⟩ ::=	⟨varref⟩	
	| ⟨array-exp⟩ [⟨exp⟩]	`arrayref (array index)`
	| ⟨exp⟩	

and we invoke thunks with a slightly modified version of the call-by-reference *eval-rand*, shown in figure 6.5.2. If the thunk contains a variable or an array reference, the corresponding L-value is returned. If the thunk contains a more complex expression, the best we can do is to evaluate it and put it in a new cell. Because this mechanism tries to return an L-value rather than copying its contents, call-by-name often coincides with call-by-reference.

All that remains is to define the auxiliary procedures used in figure 6.5.1. Since these L-values are the same as in the call-by-reference case, we can do this by adapting them to invoke thunks when necessary; see figure 6.5.2.

The procedure *denoted->expressed* first coerces its denoted value to an L-value by using *denoted->L-value* to invoke the thunk if necessary. It then dereferences the L-value, just as it did in the call-by-reference case. The procedure *denoted-value-assign!* works similarly. Since arrays are expressed but not denoted, we use the version of *do-letarray* that calls *make-cell*. Furthermore, since arrays are expressed values, array expressions may be arbitrary expressions, and *eval-array-exp* must be *eval-exp*. Similarly, *eval-rator* must be *eval-exp*. This completes the procedures that need to be defined for the interpreter.

• *Exercise 6.5.1*

Since call-by-name is so much like call-by-reference, you might expect *swap* to work under call-by-name. One example of an unpleasant surprise provided by the interaction between delayed array index evaluation and assignment is that *swap* may fail. What values are printed by the following expression?

```
letarray a[2]
in local i = 0
   in begin
        a[0] := 1; a[1] := 0;
        swap(i, a[i]);
        writeln(a[0], a[1])
      end
```

□

• *Exercise 6.5.2*

What values are displayed when these two programs terminate?

```
local p = proc (x)
            local a = *(x, x)
            in begin
                 x := 5; write(a)
               end
in p(3)
```

```
local a = 10
in local p = proc (x)
               begin
                 a := 3; a := +(x, 5); a := +(x, 5)
               end
   in begin p(+(a, a)); write(a) end
```

Replace "`local a`" by "`let a`" in both expressions. Now, what values are printed? Remember that `let` is syntactic sugar for procedure application. □

∘ *Exercise 6.5.3*

Write an interpreter that implements procedure calling by substituting the operands for the formal parameters of the procedures. What difficulties do you encounter? □

∘ *Exercise 6.5.4*

Although the combination of assignment and delayed evaluation is problematic, there are times when it is useful. The classic example is a procedure that computes integral approximations. Write a procedure *int* that takes an integrand expression E, an integration parameter x (a variable occurring free in the integrand expression E), lower and upper bounds a and b, and an increment δ, and returns an approximation to the integral using the rectangular rule. That is,

$$\texttt{int}(E\ x\ a\ b\ \delta) = \left(\sum_{\substack{x=a,a+\delta,\ldots,\\ a+((\lceil (b-a)/\delta \rceil)-1)\delta}} \delta \cdot E \right) \approx \int_a^b E\,dx$$

For example, assuming arithmetic operations are defined on floating point numbers (written with decimal points) as well as integers, and a primitive division procedure has been added to the initial environment, we could obtain a rough approximation of $\int_1^5 2x\,dx$ as follows.

```
--> define x = 0;
--> int(*(x, 2), x, 1, 5, quotient(1, 2));
22.0
--> x;
5.0
```

The variable *x* could have been initialized to any value. Assignments to *x* performed by *int* leave it with the value of the upper bound (or a bit more). The trick of assigning to an argument that is the binding of a variable occurring free in another argument is called *Jensen's device*. How should *int* be written using first-class procedures? □

When side effects are not involved, a delayed argument yields the same value each time it is referenced. Repeated evaluation of such arguments is a waste of effort. By saving the value obtained the first time such an argument is needed, it is possible to avoid redundant computation. This is *call-by-need.* A similar technique, memoization, was used in section 4.5.2 to avoid repeated evaluation of a stream's tail.

We can turn the call-by-name interpreter into a call-by-need interpreter by memoizing the result of invoking a thunk, so that later evaluations of the same variable will see the result instead of the thunk. To do this, we must change the set of denoted values slightly:

$$\begin{aligned}\text{Denoted Value} &= \text{L-value} + \text{Memo}\\ \text{Memo} &= \text{Cell (Thunk} + \text{L-value)}\end{aligned}$$

so that a denoted value is either an L-value or a memo, which is a cell containing either a thunk or its resulting L-value. We represent a memo as a record type:

```
(define-record memo (cell))
```

To incorporate this into the interpreter, we change *eval-rands* to produce memos and *denoted->L-value* to do the memoization. The procedure *denoted->L-value* looks to see if its denoted value is a memo. If not, it must be an L-value, so it is returned directly. Otherwise, it looks at the contents of the cell. If the cell contains an L-value, it returns that L-value. If not, the cell contains a thunk, so the thunk is invoked with *eval-rand*, returning an L-value. This L-value is put in the cell, and then returned; see figure 6.5.3.

◦ *Exercise 6.5.5*

What do call-by-name and call-by-need print for this program?

```
local a = 10;
      p = proc (x) +(x, x)
in p(+(begin write(a); a end, a))
```

□

Figure 6.5.3 Auxiliaries for call-by-need interpreter

```
(define eval-rands
  (lambda (rands env)
    (map
      (lambda (rand)
        (variant-case rand
          (varref (var)
            (let ((den-val (apply-env env var)))
              (if (memo? den-val)
                  den-val
                  (make-memo (make-cell
                               (make-thunk rand env))))))
          (else
            (make-memo (make-cell
                         (make-thunk rand env))))))
      rands)))

(define denoted->L-value
  (lambda (den-val)
    (if (memo? den-val)
        (let ((cell (memo->cell den-val)))
          (let ((contents (cell-ref cell)))
            (if (thunk? contents)
                (let ((l-val (eval-rand
                               (thunk->exp contents)
                               (thunk->env contents))))
                  (cell-set! cell l-val)
                  l-val)
                contents)))
        den-val)))
```

6.6 Optional and Keyword Arguments

In most languages, procedures usually require a fixed number of arguments, and an exception is raised if they are called with the wrong number. If a given argument is the same in most calls to a procedure, it is convenient to omit this argument from these calls. This is possible if there is provision for *optional* arguments. If an optional argument is omitted from a call, the corresponding formal parameter is associated with a *default* value that is specified by the called procedure. For example, it is usually possible to supply a port to an output procedure so that output may be directed to a chosen file. In most cases, however, there is a "standard output port" to which most output is directed, such as the terminal in an interactive system. Thus the port argument of an output procedure is often optional and defaults to the standard output port.

Procedures may have several optional arguments. Usually optional parameters must follow required parameters in formal parameter lists. If there are n required parameters and m optional parameters, the procedure may be called with i arguments, where $n \leq i \leq n+m$, and the last $n+m-i$ parameters assume their default values. Arguments that it would most often be convenient to omit should be placed last, since omission of an argument requires omission of all the following arguments. Scheme input and output procedures take optional port arguments in this way.

Scheme procedures with optional arguments may be created using a syntax in which an improper list is used to specify the formal parameters.

```
(lambda (var1 ... varn . opt) body)
```

When a procedure created with this syntax is invoked with $m \geq n$ arguments, *opt* is bound to a (possibly empty) list of arguments $n+1$ through m. This generalizes the Scheme expression of the form `(lambda` *var body*`)` discussed in section 1.3, which may be used to create procedures in which all arguments are optional.

The Scheme optional-argument form does not allow the specification of default values for optional arguments. We can characterize this feature by a translation into Scheme as follows:

```
(lambda-opt (var1 ... varn (opt1 exp1) ... (optk expk))
  body)

⇒ (lambda (var1 ... varn . opts)
    (let ((len-opts (length opts)))
      (let ((opt1 (if (< len-opts 1) exp1 (list-ref opts 0)))
            ...
            (optk (if (< len-opts k) expk (list-ref opts k-1))))
        body)))
```

This translation highlights some design decisions about the default values. The expressions exp_i appear inside the scope of the var_j, but they could have been placed outside that scope. Furthermore, the exp_i are evaluated each time the procedure is called. Even if the exp_i were not in the scope of the var_j, we might get different values because of side-effects.

• *Exercise 6.6.1*
Consider the concrete and abstract syntax of ⟨varlist⟩.

⟨varlist⟩ ::=	`( )`	
	| `(`⟨vars⟩ {`,`⟨keydecl⟩}*`)`	
	| `(`⟨keydecls⟩`)`	
⟨keydecls⟩ ::=	⟨keydecl⟩ {`,`⟨keydecl⟩}*	
⟨keydecl⟩ ::=	`:`⟨var⟩ `=` ⟨exp⟩	`keydecl (var exp)`

Then modify the interpreter of figure 5.5.1 to support optional arguments. When a procedure with optional formals is called, the expressions associated with optional arguments are evaluated in the scope outside the procedure to obtain their default values before evaluating the procedure body. It is a run-time error to omit an argument that has not been given a default value.

```
--> (proc (x, :a = 1) +(x, a))(100);
101
--> let y = 3
    in (proc (x, :a = 1, :b = +(2, y))
          +(x, +(a, b)))
       (100, 10);
115
```

□

∘ *Exercise 6.6.2*
The default values may appear inside or outside the scope of the non-optional formals var_j, and they may be evaluated at procedure creation time or at procedure call time. This gives two design decisions, for a total of four possible designs. Are all of them sensible? Give examples to show how the same program would give different answers in each possible design. Modify the equation for `lambda-opt` to express each design. □

In most languages, formal parameters are associated with arguments by *position*: the value of the n^{th} operand of an application is associated with the n^{th} variable in the procedure's formal parameter list. It is also possible to make this association by pairing operands with *keywords* that name the corresponding formal parameter. It then does not matter in what order these operands appear. In Lisp dialects that support keyword parameters, these keywords are typically indicated with a colon followed by the associated variable name. For example, a procedure ***make-window*** with keyword parameters named `height` and `width` might be called with

```
make-window(:height = 1, :width = +(2, 3)),
```

which is the same as

```
make-window(:width = +(2, 3), :height = 1).
```

Some languages use only keyword operands. In others, both positional and keyword operands may be used in the same call, with the positional operands occurring first. With keyword operands, the programmer must remember the name but not the operand's position, whereas for positional operands it is the position, not the name, that matters.

Keyword operands add significantly to the visual complexity of applications, unless they allow the number of operands to be significantly reduced. This is the case when they are used in combination with optional operands. Then any collection of optional operands may be used in a call. An optional operand identified by a keyword is especially useful when the behavior of a procedure is determined by a large number of parameters whose default values are suitable most of the time. There may be many optional parameters that a programmer does not even know exist, but a rarely needed parameter may be specified using the appropriate keyword. Operating system commands often use some form of keyword parameter mechanism.

∘ *Exercise 6.6.3*

Modify the interpreter of exercise 6.6.1 to support keyword parameters by extending the concrete and abstract syntax of ⟨operands⟩.

$$
\begin{aligned}
\langle\text{operands}\rangle &::= \texttt{( )} \\
&\mid\ \texttt{(}\langle\text{exps}\rangle\ \{\texttt{,}\langle\text{keydecl}\rangle\}^*\texttt{)} \\
&\mid\ \texttt{(}\langle\text{keydecls}\rangle\texttt{)} \\
\langle\text{exps}\rangle &::= \langle\text{exp}\rangle\ \{\texttt{,}\langle\text{exp}\rangle\}^*
\end{aligned}
$$

For example,

```
--> (proc (x, y) -(x, y)) (:y = 3, :x = 2);
-1
--> (proc (x, :a = 1, :b = +(2, 3))
       +(x, +(a, b)))
    (100, :b = 2);
103
```

□

6.7 Summary

When an assignment is made to a binding that contains an aggregate, if the aggregate is represented indirectly, the pointer to the aggregate is replaced with another value. If the aggregate is represented directly, the assignment is directly to the elements of the aggregate.

Call-by-reference allows a procedure to return information to its caller by assigning new values to its parameters. This is made possible by parameters that are references to variable bindings or data structure elements belonging to the caller. With call-by-reference it is possible for two parameters to refer to the same location. This phenomenon, known as aliasing, makes it difficult to understand programs. Call-by-value-result also allows information to be returned to the caller through variables but avoids the aliasing problem.

Analysis of the expressed and denoted values of our languages play an important role in the design of all these alternatives. Some languages are rich in expressed values but poor in denoted values, and others have the opposite mix.

Call-by-name delays evaluation of arguments until their values are needed, as with leftmost evaluation in the lambda calculus. This is achieved by passing thunks that close arguments over the calling environment. Call-by-name in combination with assignment is dangerous, since it may be difficult to predict just when assignments will take place. Call-by-need is a memoized variation on call-by-name. It avoids the inefficiency of repeatedly evaluating the same argument, but in combination with assignment it may not yield the same results as call-by-name.

Optional arguments and keyword operands are useful when a procedure has many parameters. With keyword operands the programmer need not remember the order of the arguments, and optional arguments may be omitted when a default value supplied by the procedure is appropriate.