Solutions to Exercises 5

5.1.1 We could redesign the syntax of C or C++ to distinguish between a function definition and a proper procedure definition as follows:

function definition	proper procedure definition
T I `(`*FPs*`)` *B*	**`proc`** *I* `(`*FPs*`)` *B*

and distinguish between a function call (an *expression*) and a proper procedure call (a *command*) as follows:

function call	proper procedure call
I`(`*APs*`)`	*I*`(`*APs*`);`

* **5.1.2** We could redesign the syntax of ADA's function definitions as follows:

`function` *I* `(`*FPs*`)` **`return`** *T* **`is`** *E*

where the function body is the expression *E*, which would be evaluated whenever the function is called.

This change, in isolation, would reduce the expressive power of functions because the function body could neither declare local variables nor contain loops. To compensate we would have to add block expressions, command expressions, and/or iterative expressions to the language.

5.2.1 An example of a parameterless procedure that does not perform exactly the same computation every time it is called would be one that displays the time of day. In general, a parameterless procedure will behave like this if it updates a global variable, or accesses a global variable that is updated by some other means (the system clock in this example).

5.2.3 Parameter mechanisms in the ADA procedure `multiply`:

(a) If the copy-in-copy-out parameter mechanism is used, the procedure calls would print 6, 3, 4, 2.

(b) If the variable parameter mechanism were used instead, the same procedure calls would print 6, 3, 4, 4. The reason for the difference is that the second procedure call '`multiply(i, i);`" causes both formal parameters `m` and `n` to be aliases of the variable `i`, so the assignment to m simultaneously updates `m`, `n`, and `i`.

5.2.4 The procedure call "`add(a, b, c);`" works as follows:

- Copy parameter mechanisms: all components of `a` and `b` are copied into `v` and `w`, `sum` is created but not initialized, the components of `sum` are updated by adding the respective components of `v` and `w`, then all components of `sum` are copied into `c`. The net effect is to make `c` contain the vector sum of `a` and `b`.
- Reference parameter mechanisms: `v`, `w`, and `sum` are made to refer to `a`, `b`, and `c`, respectively, then the components of `c` are (indirectly) updated by adding the respective components of `a` and `b` (accessed indirectly). The net effect again is to make `c` contain the vector sum of `a` and `b`.

The procedure call '`add(a, b, b);`' works as follows:

- Copy parameter mechanisms: all components of `a` and `b` are copied into `v` and `w`, `sum` is created but not initialized, the components of `sum` are updated by adding the respective components of `v` and `w`, then all components of `sum` are copied into `b`. The net effect is to make `b` contain the vector sum of `a` and `b`.

- Reference parameter mechanisms: `v`, `w`, and `sum` are made to refer to `a`, `b`, and `b`, respectively, then the components of `b` are (indirectly) updated by adding the respective components of `a` and `b` (accessed indirectly). Aliasing of `w`, `sum`, and `b` arises here, but the net effect again is to make `b` contain the vector sum of `a` and `b`.

5.3.1 A procedure *P* has a formal parameter *FP* of type *T*.

(a) Copy-in-copy-out parameter mechanism: The argument must be a variable of type *T*, and its address is passed to *P*. When *P* is called, *FP* is bound to a local variable, and the argument variable's current value is copied into that local variable. Inside *P*, each inspection (update) of *FP* is actually an inspection (update) of that local variable. When *P* returns, that local variable's current value is copied out to the argument variable.

(b) Variable parameter mechanism: The argument must be a variable of type *T*, and its address is passed to *P*. When *P* is called, *FP* is bound to that argument variable. Inside *P*, each inspection (update) of *FP* is actually an indirect inspection (update) of the argument variable. Nothing happens when *P* returns.