5
Procedural Abstraction

- Proper procedures and function procedures.
- Parameters and arguments.
- Implementation notes.

Abstraction

- In programming, abstraction means the distinction between what a program unit does and how it does it.
- This supports a separation of concerns between the application programmer (whose uses the program unit) and the implementor (who codes the program unit).
- This separation of concerns is essential in large-scale program development.
- Program units include:
  - procedures (here)
  - abstract types and classes (Chapter 6).

2

Abstraction (2)

- Abstraction is only helpful when it maps to a meaningful domain (mathematics, models)
- Independence of machine, OS, platform are essential for abstraction
- Commerce often stimulates this abstraction, sometimes hinders it (when defending a monopoly)

Function procedures (1)

- A function procedure (or just function) embodies an expression to be evaluated.
- When called, the function procedure yields a result value.
- Separation of concerns:
  - The application programmer is concerned only with the function procedure's result (its observable behaviour).
  - The implementer is concerned only with how the function procedure works (its algorithm).

Function procedures (2)

- A Haskell function definition has the form:

```
I (FPs) =
E
```

formal parameters

expression

- The function body is an expression E.
- A call to the function simply evaluates E to determine the result.

5

Example: Haskell function procedure

- Function definition:

```
power (x: Float, n: Int) =
if n = 0
then 1.0
else x * power(x, n-1)
```

- Function call:

```
power(y, 3)
```

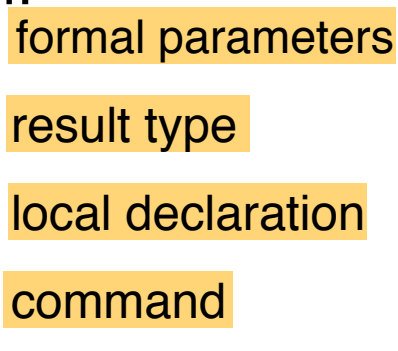
where `y :: Float`

6

Function procedures (3)

- An Ada function definition has the form:

```
function I (FPs) return T is
D
begin
C
end;
```



- The function body "D **begin** C **end;**" is syntactically a block command.

- A call to the function executes the block command. The latter must execute "**return** E**;**", which evaluates E to determine the result.

- Thus the function body acts like an expression!

Example: Ada function procedure (1)

- Function definition:

```
function power (x: Float, n: Integer)
   return Float is
p: Float := 1.0;
begin
for i in 1 .. n loop
   p := p * x;
end loop;
return p;
end;
```

- Function call:

```
power(y, 3)
```

where `y: Float`

8

Example: Ada function procedure (2)

- Recursive function definition:

```
function power (x: Float, n: Integer)
   return Float is
begin
if n = 0 then
   return 1.0;
else
   return x * power(x, n-1);
end if;
end;
```

Function procedures (4)

- A C function definition has the form:

```
T I (FPs)
B
```

result type

formal parameters

block command

  - The function body is a block command.
  - Pros and cons of C/Ada style of function definition:

+ The full expressive power of commands is available in the function body.

– Executing a block command is a roundabout way to compute a result.

– Side effects are possible.

– There's no guarantee that "**`return`** E`;`" will actually be executed.

note: Java does a static check for this

10

Proper procedures (1)

- A proper procedure (or just procedure) embodies a command to be executed.
- When called, the proper procedure updates variables.
- Separation of concerns:
  - The application programmer is concerned only with the updates effected by the proper procedure (its observable behaviour).
  - The implementer is concerned only with how the proper procedure works (its algorithm).

Proper procedures (2)

- Ada procedure definitions have the form:

```
procedure I (FPs) is
D
begin
C
end;
```

  - The procedure body "D **`begin`** C **`end;`**" is a block command.
  - A call to the procedure executes the block command.

Example: Ada proper procedure

- Procedure definition:

```
procedure sort (a: in out Int_Array) is
p: Integer range a'first .. a'last;
begin
if a'first < a'last then
   partition(a, p);
   sort(a(a'first..p-1));
   sort(a(p+1..a'last));
end if;
end;
```

- Procedure call:

```
sort(nums);
```

where `nums: Int_Array`

13

The Abstraction Principle (1)

- In summary:
  - A function procedure abstracts over an expression. That is to say, a function procedure has a body that is an expression (in effect), and a function call is an expression that will yield a value by evaluating the function procedure's body.
  - A proper procedure abstracts over a command. That is to say, a proper procedure has a body that is a command, and a proper procedure call is a command that will update variables by executing the proper procedure's body.
- There is a close analogy.
- We can extend this analogy to other types of procedures.

14

The Abstraction Principle (2)

- PL designers should bear in mind the Abstraction Principle:

  > It is possible to design procedures that abstract over any syntactic category, provided only that the constructs in that syntactic category specify some kind of computation.

- For instance, a variable access yields a reference to a variable. So a PL designer could consider:
  - A selector procedure abstracts over a variable access. That is to say, a selector procedure has a body that is a variable access (in effect), and a selector call is a variable access that will yield a variable by evaluating the selector procedure's body.

Example: C++ selector procedure (1)

- Suppose that values of type `Queue` are queues of integers.
- Function definition:

```
int first (Queue q) {
...  // return the first element of q
}
```

- Function calls:

```
int i = first(qA);
first(qA) = 0;
```

Illegal, since `first` returns an **`int`** value, not an **`int`** variable.

16

Example: C++ selector procedure (2)

- Selector definition in outline:

```
int& first (Queue q) {
...  // return a reference to the first element of q
}
```

- Selector calls:

```
int i = first(qA);
first(qA) = 0;
```

Now legal, since `first` returns a reference to an **`int`** variable.

17

Example: C++ selector procedure (3)

- Possible selector definition in full:

```
struct Queue {
   int elems[100];
   int front, rear, length;
}

int& first (Queue q) {
return q.elems[q.front];
}
```

- Now the selector call `first(qA)` yields a reference to the variable `qA.elems[qA.front]`.

18

Parameters and arguments (1)

- An argument is a value or other entity that is passed to a procedure.
- An actual parameter is an expression (or other construct) that yields an argument.
- A formal parameter is an identifier through which a procedure can access an argument.
- What may be passed as arguments?
  - first-class values (in all PLs)
  - either variables or pointers to variables (in many PLs)
  - either procedures or pointers to procedures (in some PLs).

Parameters and arguments (2)

- Each formal parameter is associated with the correspond-ing argument. The exact nature of this association is called a parameter mechanism.
- Different PLs support a bewildering variety of parameter mechanisms: value, result, value-result, constant, variable, procedural, and functional parameters.
- These can all be understood in terms of two underlying concepts:
  - A copy parameter mechanism binds the formal parameter to a local variable that contains a copy of the argument's value.
  - A reference parameter mechanism binds the formal parameter directly to the argument itself.

Copy parameter mechanisms (1)

- A copy parameter mechanism allows for a value to be copied into and/or out of a procedure:
  - The formal parameter denotes a local variable of the procedure.
  - A value is copied into the local variable on calling the procedure, and/or copied out of the local variable (to an argument variable) on return.
- Three possible copy parameter mechanisms:
  - copy-in parameter
  - copy-out parameter
  - copy-in-copy-out parameter.

Copy parameter mechanisms (2)

- Copy-in parameter (or value parameter)
  - The argument is a value.
  - On call, a local variable is created and initialized with the argument value.
  - On return, that local variable is destroyed.
- Copy-out parameter (or result parameter)
  - The argument is a variable.
  - On call, a local variable is created but not initialized.
  - On return, that local variable is destroyed and its final value is assigned to the argument variable.

Copy parameter mechanisms (3)

- Copy-in-copy-out parameter (or value-result parameter)
  - The argument is a variable.
  - On call, a local variable is created and initialized with the argument variable's current value.
  - On return, that local variable is destroyed and its final value is assigned back to the argument variable.

Example: Ada copy parameters (1)

- Procedure:

Assume:

copy-in

copy-out

```
type Vector is array (1 .. n) of Float;

procedure add (
        v, w: in Vector;
        sum: out Vector) is
begin
for i in 1 .. n loop
   sum(i) := v(i) + w(i);
end loop;
end;

...
add(a, b, c);
```

Local variables `v`, `w` are initialized to

Local variable `sum` is created but not initialized.

Final value of `sum` is assigned to `c`.

24

Example: Ada copy parameters (2)

- Procedure:

Assume:

copy-in-copy-out

```
procedure normalize (
      u: in out Vector) is
s: Float := 0.0;
begin
for i in 1 .. n loop
   s := s + u(i)**2;
end loop;
s := sqrt(s);
for i in 1 .. n loop
   u(i) := u(i)/s;
end loop;
end;

...
normalize(c);
```

Local variable `u` is initialized to value of `c`.

Final value of `u` is assigned to `c`.

25

Copy parameter mechanisms (3)

- Summary:

Parameter mechanism	Argument	On call	On return
Copy-in	value	FP := argument	—
Copy-out	variable	—	argument variable :=
Copy-in-copy-out	variable	FP := argument variable's value	argument variable := FP

- Pros and cons:

+ pleasingly symmetrical
– unsuitable for types that lack assignment (e.g., Ada limited types)
– copying of large composite values is inefficient.

26

Reference parameter mechanisms (1)

- A reference parameter mechanism allows for the formal parameter FP to be bound directly to the argument itself.
- Reference parameter mechanisms appear under several guises:
  - constant parameters
  - variable parameters
  - procedural parameters.

Reference parameter mechanisms (2)

- Constant parameter
  - The argument is a value.
  - On call, FP is bound to that value.
- Variable parameter
  - The argument is a variable.
  - On call, FP is bound to that variable.
- Procedural parameter
  - The argument is a procedure.
  - On call, FP is bound to that procedure

Thus any inspection of FP is actually an indirect inspection of the argument value.

Thus any inspection (or updating) of FP is actually an indirect inspection (or updating) of the argument variable.

Thus any call to FP is actually an indirect call to the argument procedure.

Example: Ada reference parameters (1)

- Procedure:

Assume:

const. params.

var. param.

```
type Vector is array (1 .. n) of Float;

procedure add (
        v, w: in Vector;
        sum: out Vector) is
begin
for i in 1 .. n loop
   sum(i) := v(i) + w(i);
end loop;
end;

...
add(a, b, c);
```

v, w are bound to values of a,

sum is bound to variable c.

Example: Ada reference parameters (2)

- Procedure:

Assume:

var. param.

```
procedure normalize (
      u: in out Vector) is
s: Float := 0.0;
begin
for i in 1 .. n loop
   s := s + u(i)**2;
end loop;
s := sqrt(s);
for i in 1 .. n loop
   u(i) := u(i)/s;
end loop;
end;

...
normalize(c);
```

u is bound to variable c.

30

Reference parameter mechanisms (3)

- Summary:

Parameter mechanism	Argument	On call	On return
Constant	value	bind FP to argument value	—
Variable	variable	bind FP to argument variable	—
Procedural	procedure	bind FP to argument procedure	—

- Pros and cons:

+ simple and uniform semantics
+ suitable for all types of values (not just first-class values)
– indirect reference to primitive values is rather inefficient
– variable parameters can cause aliasing.

C and Java parameter mechanisms

- C supports only the copy-in parameter mechanism. However, we can achieve the effect of a variable parameter by passing a pointer:
  - If a C function has a parameter of type `T*`, the corresponding argument must be a pointer to a variable of type T. The function can then inspect or update that variable.
- Java supports the copy-in parameter mechanism for primitive types, and the reference parameter mechanism for object types:
  - Parameters of type **`int`**, **`float`**, etc., are passed by copy.
  - Parameters of type `T[]`, `String`, `List`, etc., are passed by reference.

Ada parameter mechanisms

- Ada mandates copy mechanisms for primitive types, but allows the compiler to choose between copy and reference mechanisms for composite types:
  - Parameters of type `Integer`, `Float`, etc., are passed by copy.
  - Parameters of type `Vector` may be passed either by copy or by reference. Either way, procedure calls such as '`add(a, b, c);`' will usually have the same effect. (But aliasing might make a difference.)
- Why leave the compiler to choose? Reference mechanisms are usually more efficient, but:
  - Copy mechanisms may be more efficient for small composite types.
  - Copy mechanisms are more efficient for remote procedure calls.

Aliasing

- Aliasing occurs when two or more formal parameters are bound to the same argument variable.
- Aliasing sometimes has unexpected effects.
- In general, aliasing cannot be detected by the compiler. The onus* is on programmers to avoid harmful aliasing.

* Dutch: last; Cicero: obiit anus abiit onus

34

Example: Ada aliasing

- Procedure:

```
type Position is range 1 .. 9;
type Board is array (Position) of
Character;

procedure move (
   mark: in Character;
   pos: in Position;
   old: in Board;
   new: out Board) is
begin
new := old;
new(pos) := mark;
display(old, new);
end;

...
move('X', 9, bd, bd);
```

Assume:

var. param.

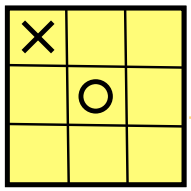
var. param.

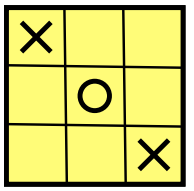
`old` is bound to `bd`.

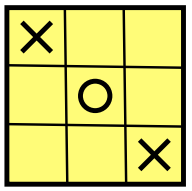
`new` is bound to `bd`.

What is displayed?

bd







Implementation of procedure calls

- When a procedure is called, an activation frame is pushed on to the stack. The activation frame contains enough space for all of the procedure's local variables, together with space for the return address and other housekeeping data. When the procedure returns, the activation frame is popped off the stack.
- At any given time, the stack contains activation frames for all currently-active procedures, in the order in which they were called.
- Recursive procedures require no special treatment.

Implementation of parameter mechanisms

- Arguments are passed by being deposited either in registers or in the procedure's activation frame.
- Each copy parameter is implemented by creating (and later destroying) a variable local to the procedure:
  - Copy-in parameter: On call, pass in the argument value and use it to initialize the local variable.
  - Copy-out parameter: On return, pass out the local variable's final value and use it to update the argument variable.
  - Copy-in-copy-out parameter: On call, pass in the argument variable's value. On return, pass out the updated value.
- A reference parameter is implemented by passing the argument's address. The procedure accesses the argument by indirect addressing.

37

Correspondence Principle

- *For each form of declaration there exists a corresponding parameter mechanism*
  - constant definition -- constant parameter
  - variable renaming definition -- variable parameter
  - variable declaration -- copy-in parameter
- Benefit: a procedure depending on X can be parameterized w.r.t. X

Relieving dependencies is the road to programmer's (software engineer's?) wisdom.