Selected Topics in Imperative Languages

Overview

Parameter-Passing
  By value
  By reference
  By value-result
  By name
Notion of Scope
  Static
  Dynamic
Procedure Activation Records
  Static links
  Displays
(Time permitting) What makes a good language?
  A look at some languages

Reading: Chapter 5 in Sethi

Imperative Languages - Motivation and Overview

- In these languages, we specify *how* to solve the problem, giving step by step instructions.
- Notions of modularity, abstraction, information-hiding.
- Based on John van Neumann's notion of computing - program is to modify some memory.
- Central are ideas of state, of variables, of assignment.

We will look at some of the most important notions in imperative languages.

Questions of Interest

1. When is there a mapping between variable name and its declarations, i.e., which declaration applies to a given occurrence of `x`?

2. When is there a mapping between declaration and storage location, i.e., which location does the name in declaration denote?

3. When is there a mapping from locations to values? Does occurrence of a variable deal with value or location? E.g., `x := x + 1` changes value, not location.

Procedures and Functions

- *Procedures* do some action.
- *Functions* return some value.

Pure functions have no side effects (as in Scheme or ML):

```
function square(x:int) :  int;
begin
  return x*x;
end
```

Pure procedures do not return anything:

```
procedure write(x:int);
begin
  write ("I am about to write a variable ");
  write (x);
end;
```

Parameter-Passing Methods

Formal parameters in procedure definition are just place holders, replaced by actuals during procedure call.

Parameter passing - matching of actuals with formals when procedure call occurs. But... what does `A[i]` mean: a name? a value? an address?

Possible interpretations:
- call by value: pass value of `A[i]`
- call by reference: pass location of `A[i]`
- call by name: pass string "A[i]"
- call by value-result: copy values of actuals into formals on entrance, copy values of formals into actuals on exit.

Call by Value

Evaluate the expression and then pass its value to the procedure.

Example:

```
procedure swap(x,y :  T);
var z :  T;
begin
  z := x; x := y; y := z;
end;
```

When we call this procedure:

```
a := 3;
b := 4;
swap(a,b);
```

the following occurs:

```
x := a;
y := b;
z := x; x := y; y := z;
```

So, values of `a` and `b` do not change.

Call by Value (Cont'd)

Expressions (e.g., all parameters that do not have addresses, *lvalues*) are always passed by value.

Disadvantages:
- changes made to the parameter during the procedure call do not get reflected when the procedure returns.
- have to copy the object (OK for small objects, bad for big).

So, what do we do if we want to modify parameters? Use pointers (as in C), return some parameters, or use call by reference (in C++, Turing, Pascal)

Call by Reference

A formal parameter becomes a synonym for the location of the actual parameter. In contemporary languages, all parameters passed by reference have to have a location. In early Fortran, could swap constants which was very confusing.

In Pascal, call by reference is marked by keyword `var`:

```
procedure swap (var x,y :  integer);
var z :  integer;
begin
  z := x; x := y; y := z;
end;
```

So, in the call

```
i := 2;
A[i] := 3;
swap(i, A[i]);
```

values of `i` and `A[i]` do get swapped.

Call by Reference (Cont'd)

The language implementation does the following: make location of `x` same as that of `i`, make location of `y` same as `A[i]` and then call the procedure.

Advantages:
- do not have to make a copy of the object
- do not have to dereference pointers

Disadvantages:
- can incidently modify a parameter.

Best if the language allows both call by value and by reference.

Call by Value-Result

Another name for this is *copy-in/copy-out*.
1. Copy-in phase. Compute values and locations of actual parameters. Values are assigned to the corresponding formals, and locations are stored.
2. Copy-out phase. After procedure body is executed, the final values are copied back to computed locations.

Call by value is used in Ada which has either value-result or reference parameters. Thus, all formals have to be specified as in, out and in-out parameters.

Call by value works the same as by reference unless aliasing is used, i.e., several names for the same location. Can change value of variable directly using assignment or through a formal.

Call by Value-Result (Cont'd)

Typical example:

```
procedure f (x,y : integer);
begin
  x := x+1;
  b := x;
  y := y-1;
  write (x,y);
end;
begin (* main *)
  a = 1; b := 2;
  f (a,b);
  write (a,b);
end;
```

Output by reference:

Output by value-result:

Call by Name

Pass a name string, resolve it only when it is necessary. In functional languages, called "lazy" evaluation.
Example:

```
procedure print (a : integer);
begin
  i := 3; write (a);
end;
begin (* main *)
  A[1] := 1;
  A[2] := 2;
  A[3] := 3;
  i := 2;
  print(A[i]);
end;
```

This prints "2" under first three mechanisms.
However, by name this prints ...

Notion of Scope

Scope rules determine which declaration of a variable (or a function) applies to a name. Two main types of scope - lexical (static) and dynamic. Scopes can be nested.

Static scoping - names are resolved during compile time. Variables come from declaration scope. (Scheme, ML, C, C++). General rule - consistent renaming of formal parameters should not change the function behavior (remember α-substitution in λ Calculus?)

- *Dynamic* scoping - names are resolved at run-time and come from calling scope. (Lisp)

Notion of Scope(Cont'd)

Example:

```
procedure L
  var n:char;
  procedure W
    begin writeln(n); end
  procedure D
    var n: char
    begin
      n := 'D'; W;
    end;
  begin
    n := 'L';
    W;  D;
  end;
```

Under static scoping:
Under dynamic scoping:

Interesting Case - Static Scoping with Macros

Macro processor does the following:
1. Actual parameters are textually substituted for the formals.
2. The resulting procedure body is textually substituted for the call.

Example:

```
var n :char = 'A'
#define WRITE {writeln(n);}
procedure D;
begin
  var n:char = 'D';
  WRITE;
end;
```

This prints `'D'`. So, we get dynamic scoping!

Call by Name

Want lexical scoping. Rename locals:
1. Actual parameters are substituted for formals. Locals are renamed if there are conflicts.
2. Resulting procedure body is substituted for the call. Rename locals at point of call if there are conflicts between nonlocals in proc. and locals at point of call.

```
procedure P(x)
begin
  i :  integer;
  i := i+n; x := x+n;
end;
begin (* main *)
  n, i :  integer;
  P(A[i]);
end;
```

Here, replace `i` by `j` in body of procedure, replace `n` by `m` in calling procedure.

Interesting Case - Static Scoping with Call by Name

```
program P(output);
var A : array[1..4] of integer;
    I,J : integer;
procedure P(X,Y,Z:integer);
  begin
    I := I+1; J := J+2; X := X*Z;
    Y := 0;
  end;
procedure Q;
  var J,K : integer;
  begin
    J := 2; K := 3;
    R(A[I], A[J], J+K);
  end;
begin (* main *)
  for J := 1 to 4 do A[J] := 3;
  J := 1; I := 2;
  Q;
  writeln (A[1], A[2], A[3], A[4]);
end;
```

What does this program print?

Activation Records

When a procedure is called, compiler needs to create storage for local variables. one for each procedure activation. This structure is called *activation* record and is stored on computer stack. In sequential languages, control can be in at most one procedure at a time. So, when a procedure is called, put its activation record on a stack and give control to it. When it is done - pop it off the stack and return control to the calling procedure.

Stack can be used if storage for locals is not needed once an activation ends (i.e., variables are not `static`).

Why Stack? Why not Table?

Q: Why put local variables on a stack rather than in a table?

A: Recursion - unknown number of invocations:

Example:

```
function fact (n :  integer) :  integer;
begin
  if n = 0 then fact := 1
  else fact := n * fact(n-1);
end;
```


C vs Pascal

In C - at most two scopes - local and global. No need to store static links.

In Pascal - scopes can be nested. Thus, a procedure might have access to local variables of many other procedures. Need to store a link to lexically enclosing environment.

Structure of Activation Records

Activation record consists of:
- Dynamic link: points to previous activation record (optional)
- Return address: instruction to execute when procedure terminates
- Static link (sl) or displays - Local storage: formal parameters, local variables, procedures, etc.

Elaborator is used to identify a control thread:
- ep (environment pointer) - points to top activation record
- ip (instruction pointer) - points to instruction currently being executed

Activation Records and Scope (Pascal)

To find declaration of variables in dynamic scope:
- start with top activation record
- follow dynamic links to calling environments

To find declaration of variables in static scope:
- start with top activation record
- follow static links to lexically enclosing environments.

Simulation with Static Links

Initially, stack is empty, elaborator is null.

Procedure entry:
- dynamic link := elaborator's ep
- return address := instruction following procedure call statement
- static link := procedure's ep.
- local variables are undefined
- local procedure declarations are presented by (ep, ip) pairs. Ep points to the new activation record, and ip - to the first executable instruction of the procedure
- formal parameters are treated as initialized local variables

Example

```
procedure L
  var n:char;
  procedure W
    begin
      writeln(n)
    end;
  procedure D
    var n: char
    begin
      n := 'D'; W;
    end;
  begin
    n := 'L';
    W;  D;
  end;
```

Simulation with Static Links (cont'd)

Update elaborator:
  ip := procedure's ip
  ep := pointer to new activation record

Procedure exit:
  ip := return address
  remove activation record from stack
  ep := new top of the stack

Example - show stack at highest point

```
1    program P;
2    var
3      X : integer;
4
5        procedure Q (var C:integer);
6
7            procedure R (var A:integer);
8            var
9               B: integer;
10           begin
11              B := A;
12           end;
13
14       begin
15          X := 3;
16          R(C);
17       end;
18
19   begin
20     X := 1;
21     Q(X);
22   end.
```


Solution using Static Links

```
     Static Storage               Dynamic Storage
     Name   Value  New New        Name       Value  New

ADR1+------+-------------+   ADR2+------------+-------+
ADRX| X    | ?  1 3      |       | Q.sl       | ADR1  |
    | Q.ep | ADR1        |       | Q.return   | 22    |
    | Q.ip | 11          |       | Q.C        | ADRX  |
    +------+-------------+       | Q.R.ep     | ADR2  |
                                 | Q.R.ip     | 15    |
                             ADR3+------------+-------+
                                 | Q.R.sl     | ADR2  |
                                 | Q.R.return | 17    |
                                 | Q.R.A      | ADRX  |
                                 | Q.B        | ?  3  |
                                 +------------+-------+
```


Passing parameters

- Space in activation record is allocated when the record is created
- When we have call by value, then copy the values of actual parameters
- When we have call by reference, then copy over address of that variable
- When we have call by value-result, then
  - Copy values of actuals into formals when activation record is created
  - Copy values of formals back to actuals before the activation record is destroyed

Solution Using Static Links

```
     Static Storage               Dynamic Storage

     Name    Value   New New        Name       Value   New

ADR1+------+-------------+   ADR2+------------+-------+
    | X    | ?   1  0  1 |       | R.sl       | ADR1  |
    | Q.ep | ADR1        |       | R.return   | 27    |
    | Q.ip | 11          |       | R.Y        | ?   1 |
    | R.ep | ADR1        |       | R.S.ep     | ADR2  |
    | R.ip | 21          |       | R.S.ip     | 18    |
    +------+-------------+   ADR3+------------+-------+
                                 | R.S.sl     | ADR2  |
                                 | R.S.return | 23    |
                             ADR4+------------+-------+
                                 | Q.sl       | ADR1  |
                                 | Q.return   | 19    |
                                 | Q.D.ep     | ADR2  |
                                 | Q.D.ip     | 18    |
                                 | Q.Y        | ?   0 |
                                 | Q.T.ep     | ADR4  |
                                 | Q.T.ip     | 7     |
                             ADR5+------------+-------+
                                 | Q.T.sl     | ADR4  |
                                 | Q.T.return | 13    |
                             ADR6+------------+-------+
                                 | R.S.sl     | ADR2  |
                                 | R.S.return | 9     |
                                 +------------+-------+
```


Example

```
program P;
  var C : integer;
  procedure R (var A : integer);
     procedure Q (B : integer);
     begin
       B := B + 1;
     end
  begin
    A := A + 1;
    Q(A);
  end
begin
  C := 3;
  R(C);
end.
```

```
     Static Storage               Dynamic Storage
     Name   Value  New         Name       Value  New
ADR1+------+---------+   ADR2+------------+-------+
ADRC| C    | ? 3 4   |       | R.d1       | ADR1  |
    | R.ep | ADR1    |       | R.return   | 15    |
    | R.ip | 12      |       | R.A        | ADRC  |
    +------+---------+       | R.Q.ep     | ADR2  |
                             | R.Q.ip     | 9     |
                         ADR3+------------+-------+
                             | R.Q.d1     | ADR1  |
                             | R.Q.d2     | ADR2  |
                             | R.Q.return | 11    |
                             | R.Q.B      | 4 5   |
                             +------------+-------+
```


Displays

Why use them? Optimization technique to access nonlocal variables faster, i.e., do not need to follow a bunch of static links on the stack. Rather, keep array of such access links, indexed by lexical nesting depth. Array element d[*i*] is maintained so that it points to the most recent activation of the block at nesting depth *i*.

General technique:
Caller saves the entire display up to its own nesting depth. It is restored when procedure returns.

Advantage: Fast search for non-local variables.

Disadvantage: More overhead in setting up activation record.

Another Example

```
program P1(output);
   var X: integer;
   procedure Q (procedure D);
      var Y: integer;
      procedure T;
         begin
           X := Y;
           D;
         end
      begin
        Y := 0;
        T
      end
  procedure R;
    var Y: integer;
    procedure S;
       begin
          if X = Y then Q(S) else X := Y
       end;
    begin
       Y := 1;
       S
    end;
begin
  X := 1;
  R;
end.
```

Simulation with Displays

Initially, stack is empty, elaborator is null.

Procedure entry:
- dynamic link := elaborator's ep
- return address := instruction following procedure call statement
- For a procedure declared at level n, store n entries from display. Assign d[n+1] to point to new activation record
- local variables are undefined
- local procedure declarations are presented by (ep, ip) pairs. Ep points to the new activation record, and ip - to the first executable instruction of the procedure
- formal parameters are treated as initialized local variables

Simulation with Displays (cont'd)

Update elaborator:
- ip := procedure's ip
- ep := pointer to new activation record

Procedure exit:
- ip := return address
- restore display
- remove activation record from stack
- ep := new top of the stack

Solution using Displays

```
     Static Storage                 Dynamic Storage
     Name   Value  New New          Name       Value  New

ADR1+------+-------------+     ADR2+------------+-------+
    | X    | ?   1  0  1 |         | R.d1       | ADR1  |
    | Q.ep | ADR1        |         | R.return   | 27    |
    | Q.ip | 11          |         | R.Y        | ?   1 |
    | R.ep | ADR1        |         | R.S.ep     | ADR2  |
    | R.ip | 21          |         | R.S.ip     | 18    |
    +------+-------------+     ADR3+------------+-------+
                                   | R.S.d1     | ADR1  |
                                   | R.S.d2     | ADR2  |
                                   | R.S.return | 23    |
                               ADR4+------------+-------+
                                   | Q.d1       | ADR1  |
                                   | Q.return   | 19    |
                                   | Q.D.ep     | ADR2  |
                                   | Q.D.ip     | 18    |
                                   | Q.Y        | ?   0 |
                                   | Q.T.ep     | ADR4  |
                                   | Q.T.ip     | 7     |
                               ADR5+------------+-------+
                                   | Q.T.d1     | ADR1  |
                                   | Q.T.d2     | ADR4  |
                                   | Q.T.return | 13    |
                               ADR6+------------+-------+
                                   | R.S.d1     | ADR1  |
                                   | R.S.d2     | ADR2  |
                                   | R.S.return | 9     |
                                   +------------+-------+
```


Activation Records, a few remaining points

- Display fields - point to all scopes in which definitions of non-local variables can be found

- To construct displays for a new activation record:
  - Go to activation record of lexically-enclosing scope (pointed to by ep)
  - Copy all display fields from there
  - Make another display field which points to the same place as ep

Review

- Procedure activations - how a language implementation stores information about procedures when
  - recursion,
  - nested procedures,
  - passing procedures as parameters

  are allowed.

- Static links/displays can be used to implement static scoping.

ood for slide eater

ood for slide eater

ood for slide eater