CSC 324: Principles of Programming Languages

Procedural Language Design Issues

Readings:
Sebesta 5th & 6th ed.: 5.3, 5.4, 5.8–5.10; 9.1–9.5, 9.11; 10.1–10.5

©Suzanne Stevenson 2001

Procedural Language Design Issues

Procedures: A Control Abstraction

- A block of code that can be called (imperative)
- A lambda expression (functional)
- A horn clause (logic programming)

Procedures modularize program structure

Components of a Procedure

1. Name
2. Formal parameters, optionally with types
   - parameter (formal parameter)
     Local variable whose value is received from caller
   - argument (actual parameter)
     The info passed from caller to callee
3. Body, which is a syntactic construct in the language:
   - Block, i.e., declarations and statements
   - Expression
   - Conjunction of terms
4. Optional result, optionally with a type

Procedure Implementation Issues

The general notion of a procedure leaves a number of points unspecified:

- How to pass parameters when the procedure is called
- How to maintain local state and control information
- How to access non-local names within a procedure body
Parameter Passing

Matching arguments with parameters:

1. Positional association:
   - Arguments are associated with parameters left to right

2. Keyword association:
   - Arguments are given tags, eg:
     procedure plot (x,y: real; penup: boolean)
     ...
     plot(0.0, 0.0, penup=>true)
     plot(penup=>true, x=>0.0, y=>0.0)

Passing Modes

How to treat arguments
(pass-by-x/call-by-x):

1. Pass by value
   (Java, C, C++, Pascal, Ada, Scheme, Algol68)

2. Pass by result
   (Ada)

3. Pass by value-result
   (some Fortrans, Ada)

4. Pass by reference
   (Java objects, C++ with &, some Fortrans, Pascal with
   var, COBOL)

5. Pass by name
   (Algol 60)

Parameter Passing

3. Optional arguments:
   - E.g., C printf("")
   - Extra arguments are packaged into some structure
   - Passed to special parameter

Example for Passing Modes

```{ c : array[1..10] of integer;
  m,n integer;
  procedure r (i : integer ) begin
    i := i + 1;
    j := j + 2
    end r;
...
  m := 2;
  n := 3;
  r(m,n); // call 1
  write m, n; // print 1

  m := 2;
  c[1] := 1;
  c[2] := 4;
  c[3] := 8;
  r(m,c[m]); // call 2
  write m,c[1],c[2],c[3]; // print 2
}```
Pass by Value

- Initial values of parameters copied from current values of arguments
- Final values of parameters are "lost" at return time (like local variables).
- Example:
  
  at call 1: \( i = 2 \) \( j = 3 \)
  
  print 1:
  
  at call 2: \( i = 2 \) \( j = 4 \)
  
  print 2:

- Benefit: Arguments protected from changes in procedure.
- Problem: Requires copying of values: costs time and space, especially for large aggregates.

Pass by Result

- No initial values of parameters
- Final values of parameters are copied back to arguments
- Example: does not work, as written

\[ \Rightarrow \text{For output values only. Used to indicate that a parameter is intended solely for returning a result.} \]

Pass by Result (Example)

Suppose proc \( r \) initializes \( i \) and \( j \) to 0:

- call 1:
  - final values of \( i \) and \( j \):
    - \( m \) and \( n \) are set to:
  
- print 1:
  
- call 2: more problematic
  - final values of \( i \) and \( j \):
    - which element of \( c \) is modified, \( c[1] \) or \( c[2] \)?
  
- print 2:
  - If \( c[1] \) is modified:
    - If \( c[2] \) is modified:

Problems with Pass by Result

- Requires copying of values: costs time and space, especially for large aggregates. (Cf. Call by value.)

- What if the argument is not a variable?
  E.g., \( r(1, 2) \);

- What if a variable is used twice in the argument list?
  E.g., \( r(m, m) \);

- What about calculations to determine locations of arguments?
  E.g., which \( c[m] \)?
Pass by Value-Result

- Initial values of parameters copied from current values of arguments
- Final values of parameters copied back to arguments

⇒ Combines functionality of pass by value and pass by result for same parameter.

Further Specifying Pass by Result

With pass by result or pass by value-result, order of assignments and address computations is important.

Options:

1. Perform return address computations at call time:
   On second return:
   m set to 3; c[2] set to 6
   print 2:

Further Specifying Pass by Result (cont’d)

2. Perform return address computations at return time:
   (a) Before any assignments:
       On second return: same as above, but might not be if procedure has side-effects
   (b) Just before that assignment, in order:
       On second return:
       m set to 3; c[3] set to 6
       print 2:
Pass by Reference

• Formal parameters are pointers to the actual parameters (arguments).

• Address computations are performed at procedure call.

• Changes to the formal parameters are thus changes to the actual parameters.

Pass by Reference

• Benefit: No copying for variables

• Problem: allow redefinition of expressions and constants?

• Problem: Leads to aliasing
  
  – two or more visible names for same location

  – can cause side effects not visible from code itself

Pass by Reference (Example)

• call 1:
  
  – initial: \( i = j \)
  
  – final: \( i = j \)
  
  – return: \( m, n \) are:

• print 1:

• call 2:

  – initial: \( i = j \)

  – final: \( i = j \)

  – return: \( m, c[2] \) are:

• print 2:

Aliasing

```python
{ y : integer ;

procedure p ( x : integer ) begin
  x := x + 1;
  x := x + y
end p;
...

y := 2;
p(y);
write y
}
```
Aliasing

Pass by Reference:

- The identifiers x and y refer to the same location in call of p.
- Result of "write y"?

Pass by Value-Result:

- The identifiers x and y refer to different locations in call of p.
- Result of "write y"?

Pass by Name (Example)

- Example:
  - call 1: m, n set to:
  - print 1:
  - call 2: m, c[m] set to:
  - print 2:

  - Benefit: same as pass by reference

  - Problems: Inefficient, requires a thunk:
    - essentially a little program is passed that represents the argument
    - evaluates argument in caller's environment

More Aliasing

```cpp
{ i, j, k : integer ;
  procedure q ( a, b : integer ) begin
    a := i * b;
    b := i * b;
  end q;
  ...
  i := 2; j := 3; k := 4;
  q(i,j);
  q(k,k);
}
```
Procedure Activations

Summary of Parameter Passing Modes

- Pass by value
- Pass by result
- Pass by value-result
- Pass by reference
- Pass by name

Lifetime of procedure:
- Begins when control enters activation (call)
- Ends when control returns from activation

Activation Tree:
- Shows flow of control from one activation to another
- Root: Main program
- Edges: Call from one procedure to another (read left to right)
- Leaves: Procedures that call no other procedures

Sample Activation Trees

Example

main

procedure P
begin
    procedure S begin ... end S;
    if random(1) < 1 then P()
    else { S(); Q() }
end P;
procedure Q begin ... end Q;
P;
Q;
P;
end
Activation Trees and Stack Frames

Running a program corresponds to a traversal of (one of) its activation tree(s).

We can represent the traversal of the tree using a stack.

Each item on the stack is called a frame.

⇒ The stack of frames not only maintains the call sequence info, but also keeps track of the local and non-local environment for each procedure.

Content of Stack Frames

• Run-time stack contains frames for main program and each active procedure.

• Each stack frame includes:
  1. Pointer to stack frame of caller (Control Link)
  2. Return address (within caller)
  3. Mechanism to find non-local variables (Access Link)
  4. Storage for parameters
  5. Storage for local variables
  6. Storage for temporary and final values

• In a language with first-class functions, this is more complex.

Procedure Activation and Run-time Stack

On a call:

1. Set up stack frame on top of run-time stack (current context)

2. Do the real work of the procedure body

3. Release stack frame and restore caller's context (as new top of stack)

Run-time stack establishes a context for a procedure invocation

Context of Procedures

Two contexts:

• static placement in source code (same for each invocation)

• dynamic run-time stack context (different for each invocation)

Name Resolution: Given the use of a name (variable or procedure name), which instance of the entity with that name is referred to?

⇒ Both static and dynamic contexts play a role in this determination.
Scope

Each use of a name must be associated with a single entity at run-time (i.e., an offset within a stack frame).

The **scope** of a declaration of a name is the part of the program in which a use of that name refers to that declaration.

The design of a language includes **scope rules** for resolving the mapping from the use of each name to its appropriate declaration.

Some Terminology

A name is:
- **visible** to a piece of code if its scope includes that piece of code.
- **local** to a piece of code (block/procedure/main program) if its declaration is within that piece of code.
- **non-local** to a piece of code if it is visible, but its declaration is not within that piece of code.

A declaration of a name is **hidden** if another declaration supersedes it in scope.

Scope Rules

Two choices:

1. Use static context: **lexical scope**
2. Use dynamic context: **dynamic scope**

For local names, these are the same.

⇒ Harder for non-local names, and not necessarily the same for both types of scope.

Scope Example

```plaintext
program L;
var n: char;  {n declared in L}

procedure W;
begin
  write(n);  {n referenced in W}
end;

procedure D;
  var n: char;  {n declared in D}
  begin
    n := 'D';  {n referenced in D}
    W
  end;

begin
  n := 'L';  {n referenced in L}
  W;
  D
end.
```
Lexical Scope

- Names are associated with declarations at compile time
- Find the smallest block syntactically enclosing the reference and containing a declaration of the name
- Example:
  - The reference to \( n \) in \( W \) is associated with the declaration of \( n \) in \( L \)
  - The output is?

Benefit: Easy to determine the right declaration for a name from the text of the program.

Dynamic Scope

- Names are associated with declarations at run time
- Find the most recent, currently active run-time stack frame containing a declaration of the name
- Example:
  - The reference to \( n \) in \( W \) is associated with two different declarations at two different times
  - The output is?

Dynamic Scope: Pros and Cons

Benefit: reduces need for parameters.

Problems:
- hard to understand behavior from the text alone.
- renaming variables can have unexpected results.
- no protection of one's local variables from a called procedure.
  (i.e., if \( A \) calls \( B \), \( B \) can modify \( A \)'s local variables.)
- can be slower to execute.

NOTE: Most languages use lexical scope, although early interpreted languages used dynamic scope because of the flexibility and ease of implementation.

Scoping and the Run-time Stack

Access link shows where to look for non-local names.

Static Scope:

Access link points to stack frame of the lexically enclosing procedure
(total no. links to follow determined at compile time)

Dynamic Scope:

Access link points to stack frame of caller
Nested Procedures and Static Scope

program
    a, b, c : integer;       // 1
procedure r
    a : integer;           // 5
        ... a ... b ... c
end r;                   // 6
procedure p
    c : integer;           // 3
    procedure s
        d, e : integer    // 8
            ... a ... b ... c ...
    r;                      // 9
end s;
    r;                     // 4
    s;                     // 7
end p;
    p;                     // 2
end

Nesting Depth

Nesting depth of a procedure is how many lexical levels deep it is.

- Main program has nesting depth 1.
- Body of \( p \) has nesting depth 2.
- Body of \( s \) has nesting depth 3.

Note: Declarations of \( p \) and \( r \) have nesting depth 1, but declarations and statements within \( p \) and \( r \) have nesting depth 2.

Nesting Depth and Access Links

procedure v
    ...
    begin /* v */
    ...
    ...u...; /* use of u */
    ...
end; /* v */

To determine the access link for name \( u \), follow \( n - m \) access links from proc \( v \) in which \( u \) is used, where \( n \) is the nesting depth of the body of \( v \) and \( m \) is the nesting depth of the declaration of \( u \).

Run-Time Stack Trace

Trace through above program, showing snapshot of run-time stack at points 1, 3, 5, 8, 5 (again).
Dynamic Scope Example

program
  a : integer;
  procedure z
    a : integer; ...
    a := 1;
    y;
    output a;
  end z;
  procedure w
    a : integer; ...
    a := 2;
    y;
    output a;
  end w;
  procedure y ...
    a := 0;
  end y;
  a := 5;
  z;
  w;
  output a;
end

Optimizing Variable Access

Problem: Accessing non-local names requires following links up the access link chain.

Solution for lexical scoping only:
Maintain a vector of currently-active static-chain frames.

- Called the display
- Pioneered in Algol60
- Makes addresses directly accessible

Display in Static Example

For example, during execution of proc s:

D[1]: Pointer to stack frame for main pgm
D[2]: Pointer to stack frame for procedure p
D[3]: Pointer to stack frame for procedure s

- Address of d is D[3]+Offset+0
- Address of e is D[3]+Offset+1
- Address of c is D[2]+Offset+0
- Address of a is D[1]+Offset+0
- Address of b is D[1]+Offset+1

Using a Display

- If a procedure is at nesting depth $n$, it may have to follow $n-1$ static links to find variable addresses
- Display is an array of pointers to stack frames
- A variable is stored at an offset in the frame pointed to by the i’th display element, where i is the nesting level of procedure where variable was declared
- Display must be maintained along with run-time stack
Maintaining the Display

Summary:
Procedural Language Design Issues

- Components of a procedure
  - name
  - parameters
  - body
  - optional result

- Parameter passing
  - pass by value
  - pass by result
  - pass by value-result
  - pass by reference
  - pass by name

- Aliasing through parameter passing
- Procedure Activations
- Stack frames
- Lexical scope
- Dynamic scope
- Implementing scope with stack frames
- Displays