Procedure Activations

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

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
```

Sample Activation Trees

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 (ie, 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.

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

Scope Example

```
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.
  (Ie, 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

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

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

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