Implementing Subprograms

- Call
  - Save registers
  - provide storage for parameters
  - provide storage for locals
  - save execution status of caller
  - provide access to non-locals

- Return
  - copy back parameter values (if necessary)
  - may deallocate local storage
  - restore access to non-locals
    - (registers)
      - variables with same name as locals
  - return control to caller

Preliminaries

- Need memory for code & data
  - code: static (instruction space)
  - data: changes with each activation => activation record (data space)

- Three ways to allocate storage for data:
  - static
  - stack-based
  - dynamic
Static Allocation (FORTRAN)

- **Simplifying factors**
  - no recursion
  - size of variables known statically
  - access non-local variables with COMMON statement

- **Result**
  - can allocate all memory statically
  - example:
    ```fortran
    main
    ...
    f (...) ...
    proc f (...) ...
    g (...) ...
    proc g (...) ...
    g (...) -- !!!
    ```
  - What about recursion???

Handling Recursion

- procedure activations are LIFO ⇒ use a stack
- for now, two simplifying assumptions:
  - no non-local references
  - size of all variables known statically
Activation Records on a Stack

- **Call**
  - set return addr, dynamic link
  - push actual parameter values
  - allocate space for locals
  - FP := TOS, reset TOS

- **Return:**
  - instruction pointer := mem[FP] (return address)
  - TOS := FP (clear this activation record)
  - FP := mem[FP + 1] (dynamic link)

Example: factorial

```plaintext
program p;
var v : int;
function fac(n : int) : int;
begin
  if n ≤ 1 then
    fac := 1
  else
    fac := n * fac(n - 1);
end;
begin
  v := fac(3);
  print(v)
end.
```

![Diagram of activation records on a stack and an example factorial program](image-url)
Lifting Restrictions

- Allow size of variables to be determined dynamically
  - semi-dynamic -- once size is fixed it remains
    e.g., arrays
  - dynamic
    e.g., variables

- Allow non-local references

Semi-Dynamic Variables

- Size fixed at unit invocation time
  proc p(a, b, c, d: int);
  var x, y: int;
  v1[a..b], v2[c..d]: array of int;
  ...
  p(2, 4, 3, 6)

- Solution
  Allocate space for pointer to each semi-dynamic variable first, then space for actual arrays; offset of pointer known statically.
Dynamic Variables

- Example: flex array (Algol 68) => size of array may change arbitrarily at runtime
  - cannot be stored on stack
    - proc p;
      var x : int;
      y, z : flex array;
  - need a heap
    - less efficient
    - more flexible
  - store pointers to heap in stack

At any point in time you might need more space but next stack space might be in use.

Non-local References

- Need
  - which Activation Record (AR)
  - variable's local offset within AR
    Note: this is all we need for a local reference; we already have the pointer to the AR

- Two cases
  - static scoping -- reference is determined by static layout of program
  - dynamic scoping -- reference is determined by call chain
Static Scoping

- Need access to the static environment => *static link*.

For a given procedure or function f, f’s static link (usually) points to the most recent AR for the procedure or function that statically encloses f.

Exception: where f is passed as a parameter

Let’s standardize on the format of an Activation Record:

<table>
<thead>
<tr>
<th></th>
<th>Dynamic Locals</th>
<th>Static Locals</th>
<th>Parameters</th>
<th>Dynamic Link</th>
<th>Static Link</th>
<th>Return Address</th>
</tr>
</thead>
</table>

Static Links

- Using static links:
  - The AR containing the definition of a variable used in f is always a fixed distance d from f’s AR along the static chain. (set of static links)
  - If f uses a non-local x that is defined in g, 
    \[ d = \text{level}(f) - \text{level}(g) \]  # links that you traverse
  - Know levels and that x was defined in g statically!

- Setting static links:
  - If a calls b, then b’s static link should be set to the AR \((\text{level}(a) - \text{level}(b) + 1)\) links along the static chain starting at a.
  - cases:
    \[
    \begin{align*}
    \text{level}(a) &= \text{level}(b) - 1 \quad \text{(in a)} \\
    \text{level}(a) &= \text{level}(b) \\
    \text{level}(a) &> \text{level}(b) \quad \text{(b outside a)}
    \end{align*}
    \]
  - it works!
Procedure Parameters

```plaintext
proc p;
  var x;
  proc h;
    ...x...
    end {h}
    ...g(h)... end {g}
  end {h}
proc g(f : procedure);
  var x;
  ...f...
  end {g}
```

- `g` can’t set up h’s static link; it must be passed in from `p`
  - this is the `env` part of a thunk = (code, env)!

Name/Value Graph for a Functor

```
attributes:: function scope (env)

This becomes the thunk
```

This becomes the thunk
Procedure Parameter Confusion

program main;
    procedure sub1;
        ...
        end (sub1)
    end (sub1)
    procedure sub2 (proc f)
        var x:
            procedure sub3;
                ...
                x := 0;
                ...
                end (sub3)
            end (sub3)
        ...
        f;
        sub2(sub3);
        end (sub2)
    ...
    sub2(sub1);
    end (main)

main -> sub2 -> sub2 -> f

referencing environment is not most recent invocation of sub2

Example: A -> B -> C -> D

proc A;
    var x,y;
    proc B;
        var y;
            proc C;
                var x;
                ...
                x + y;
                D;
            end (C)
            proc D(z);
                ...
x + y...
            end (D)
        C;
        ...
        end(B)
    B;
    ...
    end(A)

Dynamic Links:

D: x? y?

C: x

B: y

A: x,y

Static Links:

D: return addr

C: return addr

B: return addr

A: return addr

--- When are dynamic & static links the same?
Example: A -> B -> C -> D  Continued

```
proc A;
  var x,y;
  proc B;
    var y;
    proc C;
      var x;
      ... x + y..
      D;
      end (C)
      proc D(z);
      ...x + y...
      end (D)
      C;
      ...
      end(B)
      B;
      ...
      end(A)
end {C}
end {B}
end {A}
```

Which x and which y depending on whether static or dynamic scoping?

Displays -- an alternative to Static Links

- A display contains pointers to the currently accessible activation records at each static level.
- A display is usually implemented as an array, with size equal to the maximum nesting depth of the program.
A -> B -> C -> D

- proc A;
  var xy;
  ① proc B;
    var y;
  ② proc C;
    var x;
    ... x + y...
    D;
    end (C)
  ③ proc D(z);
    ...x + y...
    end (D)
  C;
  ... end(B)
  B;
  ...
  end(A)

Display Ptr in AR
is placed there when
AR is created
Its value is: Display [Proc Level]

when D exits
DSP[2] is reset
to value stored
in D's AR
i.e. ^ to C

Proc P;
var x;
  ① proc q;
    var y;
  ② proc r;
    ... x + y
    end (r)
    ...
    S(r)
    end(q)
    ...
    q;
    end (P)

Proc S (proc f);
var x,y;
  proc t(proc g);
    ... g...
    end (t)
    ... t(f)...
    end (S)

for uniformity
you might want
to store entire
display all the time
Displays vs. Static Links

- References to non-locals:
  - static links: follow \((\text{level}_{\text{def}} - \text{level}_{\text{use}})\) static links
  - display: follow one pointer

- Procedure call: \((f \rightarrow g)\)
  - static links: follow \((\text{level}_f - \text{level}_g + 1)\) static links
  - display: save all or part of display on stack, update with new values

- Procedure return:
  - static links: none
  - display: restore to saved value

Dynamic Scoping

- Deep access (reference to \(x\))
  - Follow dynamic links until an AR containing \(x\) is found.
  - NOTES:
    - # of dynamic links to be followed cannot be determined statically
    - Names of variables must be stored in ARs

- Shallow Access (reference to \(x\))
  - Maintain central table with entry for each variable in program.
    - Current values of variables live in table, not on stack.
  - Procedure call:
    - for each local var \(x\) save current value of \(x\) (in table) on stack.
  - Procedure return:
    - Restore saved values of variables to table.
  - All variable access refers to table.