Chapter 8 – Subroutines and Control Abstractions

Three major parts of a runtime environment:

- Static area allocated at load/startup time. Examples: global/static variables and load-time constants.
- Stack area for execution-time data that obeys a last-in first-out lifetime rule. Examples: method variables and temporaries.
- Heap or dynamically allocated area for "fully dynamic" data, e.g. data allocated with new
Procedure Overview

- When functions are “first class” data items themselves, they can be dynamically created and used like values just like any other data structure. (e.g., Haskell curried functions, eval)
- pass functions as arguments
- A procedure is called or activated.
- Activation record: collection of data needed to maintain a single execution of a procedure.
- access to local and non-local references.
- Static or dynamic environment (depending on scoping) must be accessible during runtime.
- When a procedure depends only on parameters and fixed language features – closed form.
- The code for a function together with its defining environment is called closure – as we can resolve all outstanding non-local environments.

Implementing “Simple” Subprograms

- Caller responsibilities
  1. Save the execution status of the caller (calling environment)
  2. Carry out the parameter-passing process by putting the parameters somewhere that the called function can access.
  3. Pass the return address of the caller to the callee
  4. Transfer control to the callee
Implementing “Simple” Subprograms

Return Actions:
1. If it is a function, move the return value to a place the caller can get it
2. Restore the execution status of the caller
3. Transfer control back to the caller

Called routine must
Create an activation record:
- local variables
- return address
- points to other environments
- parameters

See CSILM lesson “Call Stack” for an example
csilm.usu.edu, Browse Resources, CSILM Activities, Programming Languages, Call Stack
**Parameter Passing**

- **Aliases** may be created
- Type checking parameters – for legality and to pick between overloaded methods
- A reference parameter is a nonlocal variable
- The same data object passed for two parameters
  CALL S(X,X) causes aliasing
- With aliasing, interesting problems in optimizations occur.

\[
\begin{align*}
  x+2 \\
  y++ \\
  x+2
\end{align*}
\]

If \( x \) and \( y \) are aliases, \( x+2 \) isn’t a common subexpression

---

**Models of Parameter Passing**

[Diagram showing the models of parameter passing with call and return arrows between caller and callee.]
• 1. Pass-by-value (in mode)
  – Typically we copy the value in, but can be done with a constant reference pointer.
  – Parameters are viewed as local variables of the procedure
  – Disadvantages of copy:
    • Requires more storage (duplicated space)
    • Cost of the move (if the parameter is large)

  – Disadvantages of implementing by constant reference:
    • Must write-protect in the called subprogram or compiler check that there are no assignments.
    • Accesses cost more (indirect addressing)

2. Pass-by-result (out mode)
  – function return value(s)
  – Local’s value is passed back to the caller
  – Physical move is usually used
  – Disadvantages:
    • If value is passed, time and space costs to copy
    • order dependence may be a problem (if output values are aliased)

procedure subl(y: int, z: int);{
  y=0; z=5;
}
subl(x, x);
  – Value of x in the caller depends on order of assignments at the return
3. Inout mode
   - Pass by value-result  (aka copy-in copy-out or copy-restore)
   - Used to save cost of indirect access. Physical move, both ways
   - Disadvantages
     - ordering may be a problem with a call like doit(x,x)
     - time/space issues
   - Need to know whether address is computed again before copying back. doit(i,a[i])

   Issues:
   - passing is faster (as no data copy)
   - formal parameter is local object of type pointer
   - If expression is passed as an in/out parameter: a temporary location may be passed (and then the copy is changed, not the original)
   - Disadvantages:
     - access slower as is indirect (always follow a pointer to access), but passing is fast (only copy a pointer, not a whole structure)
     - may make inadvertent changes to parameters
5. Pass-by-name (Delayed evaluated parameters)
   - By textual substitution
   - Parameter is evaluated everytime it is used, but it is evaluated in the caller’s environment
   - Purpose: flexibility of late binding
   - Costly

Thunks: a pass by name argument is implemented by a little procedure (called a thunk) which evaluates the argument.

A thunk is a method to evaluate an expression that is yet to be evaluated.

---

Pass-by-name

How is it like other methods?

- If actual is a scalar variable, it is pass-by-reference
- If actual is a constant expression, it is pass-by-value
- If actual is an array element, it is like nothing else

E.g.

```
procedure subl(x: int; y: int);
begin
  x := 1;   % Seems like nothing is happening
  y := 2;   % with first assignments but it is
  x := 2;
  y := 3;
end;
subl(i, a[i]);
```
• Example:
procedure R(var i,j: integer);
begin
  m:boolean;
m := true;
i++;
j++;
end;

m := 2;
R(m,c[m]);

• pass by reference: adds 1 to m and c[2]
  Pass by name: adds 1 to m and c[3]

Parameter Passing Methods

• Design Considerations for Parameter Passing
  1. Efficiency
  2. One-way or two-way
     – These two are in conflict with one another!
     – Good programming => limited access to variables,
       which means one-way whenever possible
     – Efficiency => pass by reference is fastest way to pass
       structures of significant size
     – Also, functions should not allow reference parameters
       – Look at csilm.usu.edu Programming/
         PROGRAMMING LANGUAGES/Parameter Passing
Languages and Environments

- Languages differ on where activation records are in the environment:
  - Fortran is static: all data, including activation records, are statically allocated. (Each function has only one activation record—no recursion!)
  - Functional languages (Scheme, ML) and some OO languages (Smalltalk) are heap-oriented: all (or almost all) data, including activation records, are allocated dynamically.
  - Most languages are in between: data can go anywhere (depending on its properties); activation records go on the stack.

Simple stack-based allocation

- Nested declarations are added to the stack as their code blocks are entered, and removed as their code blocks are exited.
- Example:

```c
    { int x; int y;
      { int z;
      }
      { int w;
        // Point 1
      }
    }
```

- Note, z has been removed at point 1 as have exited scope.
Example (C): main → q → p

```c
int x;
void p ( int y)
{ int i = x;
  char c; ...
}
void q ( int a)
{ int x;
  p(1);
}
main()
{ q(2);
  return 0;
}
```

Local variable access using the ep

- In a typical language with a stack-based runtime environment, the local declarations in a procedure are fixed at compile-time, both in size and in sequence.
- This information can be used to speed up accesses to local variables, by precomputing these locations as offsets from the ep.
- Then the local frame need not have a name-based lookup operation (unlike the symbol table).
- In fact, names can be dispensed with altogether.
Non-local variable access

- Requires that the environment be able to identify frames representing enclosing scopes.
- Using the dynamic link results in dynamic scope (and also kills the fixed-offset property as you are not sure which method will contain the x. Thus, you can’t depend on a fixed location).
- If procedures can't be nested (C, C++, Java), the enclosing scope is always locatable by other means: it is either global (accessed directly) or belongs to the current object.
- If procedures can be nested, to maintain lexical scope a new link must be added to each frame: the static link, pointing to the activation of the defining environment of each procedure.

Nested Subprograms

- The process of locating a nonlocal reference:
  1. Find the correct activation record instance
  2. Determine the correct offset within that activation record instance

May need to follow several links (static chaining)
The number of links is known from compile time.
If used stack of symbol tables, can count how many tables you had to search to find it.
If used individual stacks for each value, you can record the nesting depth of each variable.
Procedure values as pointer pairs

- Each procedure becomes a pair of pointers: a code pointer (called the instruction pointer or ip), and an environment pointer (ep) pointing to the definition environment of the procedure (which will become the access link during a call).
- Such an <ep,ip> pair is sometimes called a closure.
- In computer science, a closure is a function together with a referencing environment for the nonlocal names (free variables) of that function. Such a function is said to be "closed over" its free variables.

The Process of Locating a Nonlocal Reference

- Finding the offset is easy
- Finding the correct activation record instance:
  - Static semantic rules guarantee that all nonlocal variables (that can be referenced) have been allocated in some activation record instance that is on the stack when the reference is made
Nested Subprograms

- Static Chain Maintenance
  - At the call:
    - The activation record instance must be built
    - The dynamic link is just the old stack top pointer
    - The static link must point to the most recent ari of the static parent (in most situations)
  - Two Methods to set static chain:
    1. Search the dynamic chain until the first ari for the static parent is found—easy, but slow

2. Treat procedure calls and definitions like variable references and definitions (have the compiler compute the nesting depth, or number of enclosing scopes between the caller and the procedure that declared the called procedure; store this nesting depth and send it with the call)
**Nested Subprograms**

- Evaluation of the Static Chain Method
  - Problems:
    1. A nonlocal reference is slow if the number of scopes between the reference and the declaration of the referenced variable is large
    2. Time-critical code is difficult, because the costs of nonlocal references are not equal, and can change with code upgrades and fixes

**Nested Subprograms**

- Technique 2 (for locating non-local variables with static scope) - Displays
  - The idea: Put the static links in a separate stack called a display. The entries in the display are pointers to the array's that have the variables in the referencing environment.
  - Represent references as
    (display_offset, local_offset)
    where display_offset is the same as chain_offset
  - Can access via computation. display_offset of 10 is one lookup (not a chain of length 10)
Stack is shown growing downwards. Display contains pointers to each activation record at each reachable level.

When `s` calls `q`, a single element is added to the table.
At your seats...

- Why do we do this?
- This seems well and good – but how do we keep it current?

When q calls p, a new level 1 entry is needed. Store the old one, so you can get it back. Level 2 and level 3 are unused (but unchanged).
When p calls t, a new level 2 entry is needed
Level 3 and level 4 are unused (but unchanged)
Implementing Dynamic Scoping

1. **Deep Access (search)** - nonlocal references are found by searching the activation record instances on the dynamic chain
   - Length of chain cannot be statically determined
   - Every activation record instance must have variable names recorded

2. **Shallow Access** - put locals in a central place
   - How implemented?
     a. One stack for each variable name
     b. Central referencing table with an entry for each variable name

At subprogram entry, add location for each variable.
At subprogram exit, remove location for each variable.
Using Shallow Access to Implement Dynamic Scoping

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>MAIN_6</td>
<td>MAIN_6</td>
<td>B</td>
</tr>
<tr>
<td>u</td>
<td>v</td>
<td>x</td>
</tr>
</tbody>
</table>

(The names in the stack cells indicate the program units of the variable declaration.)

Parameter Passing Conventions

- Actual/Formal Parameter Correspondence:
  1. Positional (this is what we are used to)
  2. Keyword
     - e.g. \texttt{SORT(LIST => A, LENGTH => N)};
     - Advantage: order is irrelevant
     - Disadvantage: user must know the formal parameter’s names
  3. Default Values
Overloaded Subprograms

- Def: An overloaded subprogram is one that has the same name as another subprogram in the same referencing environment
- C++ and Ada have overloaded subprograms built-in, and users can write their own overloaded subprograms
- Overloaded subprograms provide ad hoc (non-generalizable) polymorphism

Generic Subprograms

- Analogy: Would you rather have:
  - Vacuum that can adjust to any carpet height versus
  - A different vacuum for each type of carpet.

- A generic or polymorphic subprogram is one (not many) subprogram that takes parameters of different types on different activations
- A subprogram that takes a generic parameter needs to figure out what type was passed (Haskell)