CSci 5106: Programming Languages
Procedures and Procedural Abstraction

Gopalan Nadathur

Department of Computer Science and Engineering
University of Minnesota

Lectures in Fall 2019

Procedures and Procedural Abstraction

Mechanism for naming a parameterized block of code

The main components of the feature

- Procedure definition

  An example in C-like syntax

  ```
  int square(int x) { return x*x; }
  ```

  Components of the abstract syntax: *name, return type, formal parameters and their types, body*

- Procedure call

  An example in C-like syntax: `square(5)`

  Abstract syntax components: *name, actual parameters*

Syntactic wellformedness actually requires a matching between declared type and use

Procedures and Restructuring the Machine

Procedures impact along both *organizational* and the *execution* dimensions

- The effects of procedural abstraction on the organizational model
  - Supports stepwise development based on problem decomposition
  - Basis for separating functionality from implementation details using the principle of *information hiding*
  - Provides the basis for realizing modularity
- The effects to the execution model
  - Allows the granularity of atomic steps to be controlled by the user
  - Provides for *recursion* that is a distinct way of thinking about computations

Issues to Study Concerning Procedures

The main issues relate to the interaction between procedures and the rest of the program

- What are the mechanisms for parameter passing?
  - *Call-by-value*
  - *Call-by-reference*
  - *Call-by-value-result*
  - *Call-by-name*
- How is the name space managed?

  In particular, given a procedure or variable name
  - Which declaration governs it? Scope Issue
  - Which cell does it finally refer to? Activation Issue
- Given schemes for the above, how are they implemented?

Parameter Passing Mechanisms

The main question is that of how actual and formal parameters match up during invocation

The standard mechanisms

- Call-by-Value (CbV)
  Here we pass only the *value* of the actual parameter
  - actuals can be arbitrary expressions
  - if they are l-values, then they are unaffected by call
- Call-by-reference (CbR)
  Here the formal parameter becomes an alias for the actual parameter
  - Actual must be an l-value for this to work
  - Actual can be changed by the call
  - Not directly supported in C, but can be simulated by pointers and taking addresses of variables

Call-by-Value vs Call-by-Reference

Consider the following program written in pseudo-Pascal

```
var i,j : integer;
procedure swap(x,y : integer);
   var temp : integer;
   begin temp := x; x := y; y := temp;
   end;
begin i := 5; j := 7; swap(i,j); end.
```

What is the behaviour under the two parameter passing modes?

In C, we would have to write

```
void swap(int *xp, int *yp) { ... }
```

and the call would be `swap(&i,&j)`

Call-by-Value Result

Like CbV in that actuals and formals are kept separated in a procedure invocation

However, at the end, the values of the formals are copied back into the actuals (which must be l-values)

Deviates from CbR in the presence of *aliasing*

```
var i,j : integer;
procedure foo(x,y : integer);
   begin i := y; end;
begin i := 2; j := 3; foo(i,j); end
```

Consider what happens in the two cases

Was introduced in *Ada* to draw attention to aliasing and program behaviour, but is not a commonly used mechanism

Names, Binding and Bound Occurrences

- Names play an important role in programs

  They identify variables, procedures, constants, types, parameters, etc
- Corresponding to names, there are two important concepts
  - declarations introducing them with their properties
  - name uses

  These two kinds of uses are called *binding* and *bound* occurrences
- It is important to be able to say which binding occurrence pertains to which bound occurrence
- Scope rules provide the mechanisms for doing this

Scoping Paradigms

These come in two flavours

- *Static* or *lexical scoping*

  Guiding Principle: Binding occurrence should be determinable purely from the text

- Dynamic scoping

  Binding occurrences depend on execution pattern

  In particular, procedure invocations “activate” binding occurrences that govern subsequent usage

Note: a distinction arises *only when* there are non-local name occurrences

An Example that Differentiates the Two Notions

```
program testscope;
var i,j : integer;

procedure W;
begin i := j; end;

procedure D;
var j : integer;
begin j := 5; W; end;

begin j := 3; D; write(i); end.
```

What value is printed in each case?

Which Scoping Paradigm is Better?

The answer to this question is related to issues of modularity and abstraction

> *We should be able to understand what a block of code does independently of where it is going to be used*

Lexical scoping supports this principle

> *If a procedure affects a nonlocal variable, the variable that is changed is known from the textual context in which the procedure occurs*

Dynamic scoping *does not* support this principle

For this reason, dynamic scoping is seldom used

It has actually been a red herring: McCarthy “introduced” it in Lisp but long ago said it was a bug

Call by Name

This is a parameter passing mechanism that is based on macro expansion done right

Here is the way to understand how it works

- Textually substitute actuals for formals in the procedure body
- Textually substitute resulting code at places of call

However, when doing the substitution, *rename* variables to make sure to avoid accidental capture

Such renaming is justified: particular names should not be relevant to the meanings of programs

Example of Call by Name

```
int i, j;
void bar1 (int x) {
   int j;
   i = x; }
void bar2() {
   int i;
   bar1(j); }
int main() {
   ... bar2(); ... }
```

Now consider what happens to the call `bar1(j)`

To get it right we must

- rename the `j` in `bar1` when replacing `x` by `j`
- rename the `i` in `bar2` when replacing `bar1(j)`

Call-by-Name versus Call-by-Reference

CbN is like CbR but with *lazy address calculations*

Delayed address calculation can make a difference

For example, consider the following code

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

begin
   i := 1; A[1] := 2; A[2] := 2;
   swap(i,A[i]);
end.
```

Call-by-reference: `i = 2; A[1] = 1; A[2] = 2;`

Call-by-name: `i = 2; A[1] = 2; A[2] = 1;`

Nested Scopes and Visibility

Binding occurrences have scopes over blocks of text

When such blocks can be nested, there is a possibility for creating visibility “holes”

```
{
  int i;
    .  .  .
  {
    char i;
      .  .  .
  }
    .  .  .
}
```

The inner declaration of `i` creates an *occlusion* zone for the outer declaration

The outer declaration is still active but just not visible

Procedure Activation

Refers to the commencement of execution of a procedure

The questions that are relevant to this

- How do we deal with space for local variables and parameters?
  - where is this space to be provided?
  - how do we locate the cell for a particular variable?
- What happens after the invocation completes?
  - how do we reclaim the space that was allocated?
  - how do we return the results?
  - Where should control return to?

The main complexity arises from the presence of recursion

Recursion and Procedure Activation

Recursion makes it impossible to allocate space and to identify the reference of names statically

```
void ReverseLine() {
  char ch;
  while (getc(ch) != '\n') {
    ReverseLine(); write(ch); }
  }
}
```

The number of incarnations of `ReverseLine`, and, hence, of `ch` is *a priori* unknown

In this situation we have to think of a *dynamic* scheme for dealing with both issues

Stack Based Activation Records

The common procedure activation model makes the following assumptions

- Procedures complete their work in reverse order of invocation
- Procedure space is not needed after completion; specifically, local variables are not needed

The assumptions do not always hold, e.g. in functional programming and for co-routines

When they hold, space can be provided and gathered back in a stack-based fashion

The is often hardware support for maintaining such stacks, known as *activation record* stacks

Activation Record Structure

Activation records contain information of the following sort:

- Space for parameters, local variables, etc
- Return address
- Pointer to the activation record of the runtime caller

  Dynamic or Control Link
- Pointer to the activation record of the nearest statically enclosing procedure (for lexical scoping)

  Static or Access Link

The architecture designer usually describes a standard format to enhance language interoperability

Resolving Variable References

The issue here is that of finding the cell in an activation record for a variable use

Usefully thought of as a two-step process:

- Figure out the connection between variable use and binding occurrences

  Scope issue, resolved at compile-time
- Figure out where the space is for the binding occurrence for the relevant invocation

  Run-time connection between the procedure and its activation record

Once we have understood the latter issue, then we can design a composition of the two mappings

Activation Records and Variable Reference (Example)

We can illustrate these issues by considering the following code

```
program M;
  var b,c;
  procedure P;
    var x,y,z;
    procedure Q;
      var b;
      begin (* Q *) b = x; ... R ... end;
    procedure R;
      var c;
      begin (* R *) x = b + c; ... P ... end;
    begin (* P *) ... Q ... end;
  begin (* M *) ... P ... end.
```

Consider now how activation records are set up, how access links are calculated and how variable references are resolved

Setting Up Access Links

One way to do this is for the caller to pass the access link as a parameter in a procedure call

The access link can be calculated using the *nesting depth* of a procedure, i.e. the “level” at which a procedure is declared

More concretely, the compiler can set up code to pass the static link parameter as follows

- Statically determine the difference between the nesting depths of the caller and the callee
- Generate code to chain back through a number of access links equal to one more than this difference
- Insert this code at the place where the access link needs to be passed as a parameter

Resolving Variable References

Once again, the nesting depths are useful for this

Associated with a variable declaration is a nesting depth and an *offset*

Associated with the variable use is also a nesting depth

The compiler can then generate access code that carries out the following steps:

- Chain back through a number of activation records equal to the difference between declaration and use nesting depths
- Use offset associated with the declaration to get to the right cell

Speeding Up Variable Access

Chaining back through activations records at runtime can be avoided using the following ideas

- Maintain a vector whose i^{th} entry points to the activation record at nesting level i, if it is active

  This vector is called a *display*
- To look up a variable declared at nesting depth i
  - get the pointer to the activation record from ith display entry
  - use the offset information for the variable to determine the cell to access
- At a call to a procedure at nesting depth i
  - Save value in the ith display cell in the activation record
  - Save a pointer to the new activation record in the ith cell
  - Restore the ith cell in display upon return from the call

Procedures as Parameters

The following code shows the kind of scenario that is of interest

```
procedure P;
   var x;
   procedure Q;
   begin ... x ... end;
   procedure R(proc X);
   begin ... X ... end;
begin ... R(Q) ... R(P) ... end
```

The question: How do we set the access link for the call to `X` from within `R`?

One solution: Pass the access link along as one component of the procedure parameter