3 Procedure Activations

3.1 Basics

3.1.1 Procedures (Subroutines)

Two kinds of procedures:
- Function procedures return a value.
- Proper procedures do not.

A procedure has 4 parts:
1. name of procedure
2. formal parameters
3. body, consisting of
   - local declarations
   - statement (list)
4. an optional return type

\[
\text{int fact (int n)} \{ \text{return n*fact(n-1);} \}
\]

FUNCTION Fact (n: integer): integer;
BEGIN
  Fact := n*Fact(n-1)
END

3.1.2 Bindings

A binding is an association of a value with some name/entity.
In any program, “names” must somehow be bound to their
- types
- locations (l-values)
- value (r-values)

Some of these bindings are static, others dynamic.
Some Common Bindings

<table>
<thead>
<tr>
<th>Name</th>
<th>Declaration</th>
<th>L-Value</th>
<th>R-Value</th>
<th>Scope</th>
<th>Activation</th>
<th>State</th>
</tr>
</thead>
</table>

3.2 Parameter Passing

A procedure call supplies **actual parameters** to be passed to the called routine.

Within the procedure body, these are referenced via the **formal parameter** names.

```
int fact (int n) { return n * fact(n-1); }
```

```
int i = fact(23);
```

23 is the actual parameter.

```
n is the corresponding formal parameter
```

The process of matching formal parameters to actuals is called **parameter passing**.

There are many techniques for parameter passing:

- **Call-by-Value**
- **Call-by-Reference**
- **Call-by-Value-Result**
- **Call-by-Name**

Alternatively, we can classify parameter passing by the programmer’s **intent**

3.2.1 Call-by-Value

In **call-by-value**, the r-value of the actual parameter is copied into a local l-value within the called routine.

For a procedure P with formal parameter x, a call

`P(E);`

is equivalent to

```
x := E;
body of P;
if P is a function, return a result;
```

Call-by-value is the primary passing mechanism in Pascal, C, and C++.

Limitations:

- Cannot be used to send values back to the caller

```
procedure BadSwap (x, y : T);
var temp : T;
begin
  temp := x; x := y; y := temp;
end;
```

- Passing large objects (e.g., arrays) is time-consuming

3.2.2 Call-by-Reference

In call-by-reference, the formal parameter name is bound to the l-value of the actual (it becomes a synonym for the actual).

- Pascal `var` parameters are call-by-reference.
- FORTRAN uses call-by-reference.

- C programmers fake it by passing pointers (but the pointers are actually passed by value).
- C++ has reference types, which emulate call-by-reference.

Call-by-reference has constant overhead, can be used for output:

```
procedure Swap (var x : T; var y : T);
var temp : T;
begin
  temp := x; x := y; y := temp;
end;
```

C-simulation of call-by-reference:

```c
void swap (T* x, T* y)
{
  T temp;
  temp = *x; *x = *y; *y := temp;
}
```

C++’s call-by-reference:

```c++
void swap (T& x, T& y)
{
  T temp;
  temp = x; x = y; y := temp;
}
```

Limitations of call-by-reference:
Can only be applied to actual parameters that have l-values.
- OK for Swap(a, b) where a and b are variables
- OK for Swap(A[i], Rec.field)
- Not allowed: Swap(a, 2+3)
- Not allowed: Swap(1,2)

Minimal protection against inadvertent changes

Aliasing
Aliasing refers to the ability to manipulate the same r-value through different names/expressions.
Call-by-reference can result in unexpected behavior due to aliasing.

Reference & Aliasing

```c
void addVectors(const Vector& a,
                const Vector& b,
                Vector& c)
{ // Vector addition: c = a + b
    assert(a.size() == b.size());

    // make c empty
    c.erase(c.begin(), c.end());

    for (int i = 0; i < a.size(); ++i)
        c.push_back(a[i] + b[i]);
}
```

What happens if the application code says:
```
// x = x + y;
addVectors(x, y, x);
```

3.2.3 Call-by-Value-Result
Also called copy-in, copy-out.
For a procedure P with formal parameter x,
P(E):
is equivalent to
```
x := E;
body of P;
E := x;
if P is a function, return
    a result;
```
So the parameter value is copied twice, once for input and once for output.

Properties of Value-Result
- Somewhat more predictable behavior in the presence of aliasing.
- Can be used for output
- High overhead (2 copies) for large objects

Value-Result & Aliasing

```c
procedure addVectors (a, b, c: in out Vector) is
begin — Vector add: c := a + b
    c.clear(); — make c empty
    for i in 1..a’length loop
        c.push_back(a[i] + b[i]);
    end loop;
end addVectors;
```

If application does
```
addVectors(x, y, x);
```
the function will run properly, but final value of x depends on which parameter gets copied last.

3.2.4 Call-by-Name

Call-by-name is equivalent to passing the actual text of the actual parameter to the procedure, substituting it for each occurrence of the formal parameter name:

```
#define min(x, y) (x<y) ? x : y
```
expands in the following calls as:

- \(\min(a,b)\) becomes \((a<b) \ ? \ a \ : \ b\)
- \(\min(a,0)\) becomes \((a<0) \ ? \ a \ : \ 0\)
- \(\min(a,4*a*c - b)\) becomes \((a<4*a*c - b) \ ? \ a \ : \ 4*a*c - b\)
- \(\min(a,b++)\) becomes \((a<b++) \ ? \ a \ : \ b++\)

Problems occur if an actual parameter
- is a time-consuming expression
- has side-effects

Call-by-name is seen mostly in macro expansion (e.g., the C/C++ `#define`).
was used in Algol 60

- now generally regarded as a bad decision

- but many specialized, interpreted languages still exist that work by macro expansion (e.g., TeX, TCL)

Implementing Call-by-Name

For compiled languages, call-by-name is not easy to do. In Algol 60, a call \( f \circ (a+f(b)) \) is translated by

- compiling the actual parameter expression \( a+f(b) \) into a small chunk of “stand-alone” object code, called a “thunk”.
- passing the address of the thunk to the \( f \circ \) routine.

So, in the body of \( f \circ \),

\[
\text{procedure } f \circ (x: \text{integer});
\]

every reference to the formal \( x \) is translated as a subroutine call to the address \( x \).

3.2.5 Intent

Classify a programmer’s expectations when choosing formal parameters for a new procedure:

Direction:

- An \textbf{in} parameter supplies input to the procedure
- An \textbf{out} parameter receives output from the procedure
- An \textbf{in out} parameter supplies an input value that can be modified, with the modified value forming an output of the procedure.

Size: The actual parameters may range from \textit{small} (1–2 words at most) to \textit{large} (thousands of bytes).

Preferred Passing Modes: Time

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>in:</td>
<td>small</td>
<td>—</td>
<td>\checkmark</td>
</tr>
<tr>
<td></td>
<td>large</td>
<td>\xmark</td>
<td>\checkmark</td>
</tr>
<tr>
<td>out:</td>
<td>small</td>
<td>—</td>
<td>\checkmark</td>
</tr>
<tr>
<td></td>
<td>large</td>
<td>\checkmark</td>
<td>\xmark</td>
</tr>
<tr>
<td>in out:</td>
<td>small</td>
<td>—</td>
<td>\checkmark</td>
</tr>
<tr>
<td></td>
<td>large</td>
<td>\checkmark</td>
<td>\xmark</td>
</tr>
</tbody>
</table>

\(\checkmark\): nearly optimal, \(\xmark\): poor, —: acceptable

Preferred Passing Modes: Safety

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>\checkmark</td>
<td>\xmark</td>
<td>\xmark</td>
</tr>
<tr>
<td>out</td>
<td>\checkmark</td>
<td>\checkmark</td>
<td></td>
</tr>
<tr>
<td>int out</td>
<td>\checkmark</td>
<td>\checkmark</td>
<td></td>
</tr>
</tbody>
</table>

Languages can aid in safety by forbidding modification of “in” parameters.

Parameter Passing in Pascal

- Default is call-by-value
- \texttt{VAR} parameters are passed by reference

Dilemma: how to pass array in

\[
\text{PROCEDURE } \text{FIND}(\text{A: ARRAY}[1..1000] \text{OF INTEGER};\text{N: INTEGER};\text{VAR POSITION: INTEGER});
\]

- Pass by value is slow
- \texttt{VAR} is unsafe and may give callers false impression about in/out intention

Preferred Passing Modes in Pascal

<table>
<thead>
<tr>
<th>Dir.</th>
<th>small</th>
<th>large</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>(value)\text{VAR}</td>
<td></td>
</tr>
<tr>
<td>in out</td>
<td>\texttt{VAR}</td>
<td>\texttt{VAR}</td>
</tr>
</tbody>
</table>

Parameter Passing in C++

- Call-by-value
- C++has reference types (\&) that hold a “reference” to an object
  - Passing a reference by value is functionally and lexically equivalent to call-by-reference

Parameter Passing in C++(cont.)

- Reference types can be modified with const
  - attempts to modify referenced object are illegal
Preferred Passing Modes in C++

<table>
<thead>
<tr>
<th>Dir.</th>
<th>small</th>
<th>large</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>(value)</td>
<td>const &amp;</td>
</tr>
<tr>
<td>out</td>
<td>&amp;</td>
<td>&amp;</td>
</tr>
<tr>
<td>in out</td>
<td>&amp;</td>
<td>&amp;</td>
</tr>
</tbody>
</table>

Parameter Passing in Ada

- Programmer labels formal parameters as \texttt{in}, \texttt{out}, or \texttt{in out}.
  - Modification of \texttt{in} parameters is forbidden
- Compiler must pass integers and floating point numbers by value for \texttt{in}, “copy” for \texttt{out}, and value-result for \texttt{in out}.
- For other types, compiler may choose between above techniques and call-by-reference, whichever is faster.

Preferred Passing Modes in Ada

<table>
<thead>
<tr>
<th>Dir.</th>
<th>small</th>
<th>large</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>\texttt{in}</td>
<td>\texttt{in}</td>
</tr>
<tr>
<td>out</td>
<td>\texttt{out}</td>
<td>\texttt{out}</td>
</tr>
<tr>
<td>in out</td>
<td>\texttt{in out}</td>
<td>\texttt{in out}</td>
</tr>
</tbody>
</table>

3.3 Scopes

Scope rules explain how a use of a name is mapped back to its declaration.

3.3.1 Scope

The \textit{scope} of a declaration is the range of source code within which that declaration is effective.

Scope rules can be \textit{static} (lexical) or \textit{dynamic}.

Static vs. Dynamic Scope

```c
int n = 0;
void increment() { ++n; }
void printn() { cout << n; }
int main()
{
    int n = 0;
    increment();
    printn();
}
```

Output under static scope? Dynamic?

- Static scoping is most common.
- Dynamic is seen mainly in specialized, interpreted languages and macros.
- LISP featured dynamic scoping, but this is largely being replaced by static.

Static Scope

- Static scope rules largely work via “containment”:
  - The scope of a declaration extends to the end of the procedure/block/whatever that contains it.
- Syntactic structure provides barriers that divide a program’s “namespace” into separate regions.
- A surprising amount of the history of HLL’s is tied up in the evolution of scope rules.

“Flat Space” Scope Rules

Global Name Space

- Names are either global or local to a procedure: FORTRAN, COBOL, BASIC, ALGOL 60

FORTRAN scope example

```fortran
SUBROUTINE IEXIT (ISTATE)
COMMON B(10), C
IF (ISTATE .NE. 2) GO TO 10
ITAIL = 3
```
“Block Structured” Scope Rules

In block structured HLLs, procedures can nest within each other. Each procedure has its own local names, and can access names of containers as well.

Pascal scope example
Pascal has very pure nesting rules:

- procedures can nest within one another
- the “main” program is treated as an outermost procedure, within which all others are nested
- any use of a name refers to the innermost nested declaration of that name
  - cannot reference declarations that are not local to the current routine or to one of the routines it is nested within
  - additionally, each name must be declared before being used
- a declaration of a name hides any outer declarations of the same name throughout the rest of this procedure and any later procedures nested within this one

```plaintext
program Compiler (input, output);
var i: integer;
procedure scan;
procedure getch;
... getch...
```
The procedures calls in `parse` will mimic the structure of the grammar:

\[
\begin{align*}
\langle \text{exp} \rangle & ::= \langle \text{exp} \rangle + \langle \text{term} \rangle \\
& \quad | \langle \text{exp} \rangle - \langle \text{term} \rangle \\
& \quad | \langle \text{term} \rangle \\
\langle \text{term} \rangle & ::= \langle \text{term} \rangle \ast \langle \text{factor} \rangle \\
& \quad | \langle \text{term} \rangle \div \langle \text{factor} \rangle \\
& \quad | \langle \text{factor} \rangle \\
\langle \text{factor} \rangle & ::= \text{id} | \text{number} \\
& \quad | (\langle \text{exp} \rangle)
\end{align*}
\]

Note that the nesting rules support this nicely.

**“Nesting is for the Birds†”**

Nesting is meant as a way to impose control on the namespace.

- Why is control needed?

- What happens if `factor` and `getch` need to share a symbol?
  
  - The shared symbol must be promoted to the innermost common container.

  In many cases, this forces symbols to be global.

- Studies showed that, in practice, nesting in production code seldom went more that a few levels deep.

---

**C scope example**

C’s rules fall between the flat structure of early languages and the full nesting of Pascal.

- Procedures (functions) cannot nest within one another

- Statement lists `{...}` can nest within one another.

- the “main” program is just another procedure, at the same level as the others.

- All functions are declared at “file” scope - the declaration remains in effect to the end of the containing file.

- Other names may be declared at file scope, within a function, or within a statement list.

  - scope of that declaration extends to the end of the innermost containing statement list, function, or file.

  - hides any outer declarations of the same name

- any use of a name refers to the innermost nested declaration of that name

- Some names may be used before/without declaration. These are implicitly declared at file scope.
Procedures are grouped into **modules**.

Each module has **public** and **private** namespaces.

Only procedures belonging to a module may access its private names.

### 3.4 Activation Records

<table>
<thead>
<tr>
<th>scope</th>
<th>activation</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>declaration</td>
<td>l-value</td>
</tr>
</tbody>
</table>

A procedure is **activated** when a call to it begins.

The **activation** ends when it returns to the caller.

In early FORTRAN (no recursion), procedure calls could be implemented by associating with each procedure a hidden variable to hold the return address.

- Caller would
  - place its current PC (program counter) in that variable,
  - then jump to start of procedure code

- Called routine would
  - execute its code body variable,
  - then jump to address stored in that variable

This does not work with recursion, because the same procedure may have many simultaneous activations.

In general, call-and-return follow a LIFO discipline.

- The data required for each activation is collected into an **activation record** or **frame**.

- Activation records are collected into an **activation stack** or **runtime stack**.

### 3.4.1 Anatomy of an Activation

Structure of AR’s is machine/compiler dependent.
A typical one is

Typical contents of saved state includes

- return address
- contents of critical registers
- access link or display level (later)

Parameters include

- actual parameter values
- space for function return value

Activating a procedure: Caller
For a call `foo(a, b+c, d);`

1. Push state information, including return address, TF, and TOS
   - text calls saved value of TF a control link.
2. Evaluate each actual expression, push result/address onto stack
3. jump to `foo`’s starting address …
4. Copy function return value (if any)
5. Restore state information from below TF

Activating a procedure: Called
For a call `foo(a, b+c, d);`

a. Set TF to TOS
b. Push enough bytes to hold local variables
c. Execute code body
   - parameters accessed as negative offsets from TF
   - locals accessed as positive offsets from TF

d. Jump to return address in saved state area.

For a call `foo(a, b+c, d);`

1. Push state information, including return address, TF, and TOS
2. Evaluate each actual expression, push result/address onto stack
3. jump to `foo`’s starting address

For a call `foo(a, b+c, d);`

a. Set TF to TOS
b. Push enough bytes to hold local variables

c. Execute code body

d. Jump to return address in saved state area.

4. Copy function return value (if any)

- As compiler sees parameter & local declarations, assigns an offset to each declaration. Offset gives position of variable in AR relative to TF.

- Because $z$’s scope is disjoint from that of $y_2$ and $q$, it can share storage with one of these variables.

Most compilers will allocate the local storage all at once:

5. Restore state information from below TF

### 3.5 Case study: C

- Any statement list can have local variables.
- The scope of each local declaration ends with the enclosing `{ }.

A few will place code in each `{ } to push and pop locals for each statement list:

#### Local variables: C

```c
int foo ( int x ) {
    int y;
    y = 0;
    if ( x < 0 ) {
        int z;
        z = x;
        y = foo ( z );
    } else {
        int y, q;
        y = x + 1;
        q = x - 1;
        x = y*q;
    }
    return x+y;
}
```
Access to nonlocals: C
That which is not local must be global (static).

- Globals are easy to access because they reside at a fixed address.

Procedure Parameters: C
C allows (pointers to) functions to be passed as parameters to other functions.

- No big deal — because C functions don’t nest
- We’ll see that this is more of a problem for nesting languages

3.6 Case study: Pascal
Pascal nests procedures, but not statement lists.
The combination of nesting and recursion complicates AR’s.

- Scope rules supply only part of the answer
they determine which *procedure* holds the object

- We need the most recent activation (MRA) of that procedure

Two approaches to finding most recent activations:

- access links
- displays

### 3.6.1 Access Link

An **access link** in an AR for procedure P is a pointer to the MRA of P’s immediate container.

#### Accessing Data: Access Links

For Q to find a non-local variable at offset x in P:

1. Let \( \Delta = \text{depth}(Q) - \text{depth}(P) \)
2. Follow \( \Delta \) access links back to get the MRA of P
3. Add x to the MRA address

**Constructing Access Links**

Adding access links to AR’s requires slight modification to calling sequence.

When saving state info on stack,

- If caller and called are at same nesting depth, copy caller’s access link into the new AR.
- If called routine is deeper than caller, it must be immediately nested within the caller. New access link must point to caller’s AR.

- If caller is deeper than called routine (recursion), then follow \( \Delta \) access links to MRA of called routine. Copy its access link into new AR.

### 3.6.2 Displays

A **display** is a global array of pointers to MRA’s, indexed by nesting depth.

#### Accessing Data: Displays

For Q to find a non-local variable at offset x in P:

1. Let \( d = \text{depth}(P) \)
2. Add x to \( \text{display}[d] \)

**Constructing Displays**

When saving state info on stack,

- Let \( d = \text{depth of called routine} \)
- Save \( \text{display}[d] \) in new AR
- After pushing all parameters, put TOS in \( \text{display}[d] \).

When returning from a routine at depth \( d \), restore \( \text{display}[d] \) from the saved state.

Unlike access links, displays allow constant-time access to data.

- But since most programmers don’t nest deeply, it’s not a significant difference.
Procedure Parameters: Pascal

Like C, Pascal allows procedures to be passed as parameters to other procedures.

- What to do with non-locals then?

Suppose P called Q directly. There is no activation of R. What to do with Q’s reference to x?

- not enough access links to follow
- display contains garbage at R’s depth

Later languages would address this by

**Ada**: forbidding procedure parameters — other constructs were invented to achieve similar results

**Modula 2**: procedure parameters are allowed, but the actual procedure must not be nested within another procedure.

C nests statement lists, not functions.
Pascal nests functions, not statement lists.
Ada nests both, and also has variable-size data types.