### Elements of a Procedure

#### Procedure Declaration

```c
double BM_Log (double x) {
    if (x <= 0) {
        cout << "Error\n";
        return -1;
    } else {
        return log(x);
    }
}
```

#### Procedure Call

A call of the procedure will be something like:

```c
BM_Log(2.5); /* 2.5 is the actual parameter */
```
Procedure

- **function** (function procedure):
  - returns a result to the caller
  - extends the built-in operators (+, −, ×, /). e.g. sin(x)

- **procedure** (proper procedure):
  - does not return a result
  - extends the built-in actions/statements. e.g. free(x)

But they are both called “functions” in C.

And, unfortunately, functions are called “procedures” in Scheme.

Procedures/functions are called using prefix notation.

i.e. `<procedure-name> ( <formal-parameter-list> )`

(c.f. Built-in binary operations are in infix notation.)

The parentheses “(” and “)”) are redundant.

The use of a procedure = a **call** of the procedure.

The **execution** of a procedure body = an **activation** of the procedure.
Modular Design: program $\rightarrow$ set of subprograms

- better organization $\Rightarrow$ easier to read/maintain
- easier to develop ("divide-and-conquer")

Procedure Abstraction: during the design phase, it abstracts away from how it works, and let’s think in terms of what it does.

Implementation Hiding: allows programmers to modify the underlying algorithm without affecting the high-level design.

Libraries: allow procedures of well-designed interface to be shared (reusable codes)
int factorial(int x)
{
    if (x < 0)
        exit(-1);
    else if (x <= 1)
        return 1;
    else
        return x*factorial(x-1);
}

A recursive procedure can have multiple activations in progress at the same time.
e.g F(4) ⇒ 4*F(3) ⇒ 4*(3*F(2)) ⇒ 4*(3*(2*F(1)))
Recursion: Example 1

boolean Even(int x) {
    if (x == 0) return TRUE;
    else return Odd(x-1);
}

boolean Odd(int x) {
    if (x == 0) return FALSE;
    else return Even(x-1);
}

- In this example, two recursive procedures run in “parallel”, calling each other.
int main()
{
    A(); B();
}

void B()
{
    C(); D();
}

void D()
{
    E();
}

- if $P()$ calls $Q()$, then $Q$ is a child of $P$.
- if $P()$ calls $Q()$ and then $R()$, then $Q$ appears to the left of $R$. 
Part I

Activation Records
<table>
<thead>
<tr>
<th></th>
<th>callee saves</th>
<th></th>
<th>callee saves</th>
<th></th>
<th>callee saves</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td></td>
<td>control link (to its caller’s AR)</td>
<td></td>
<td>(optional) access link</td>
<td></td>
</tr>
<tr>
<td>actual parameter #1</td>
<td></td>
<td>saved machine status</td>
<td></td>
<td>local variables</td>
<td></td>
</tr>
<tr>
<td>actual parameter #2</td>
<td></td>
<td>temporary storage</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
When a procedure is activated, temporary memory called **activation record** (AR) is allocated to run the procedure.

AR of procedure $P()$ usually contains memory for:

- **returned result** (if $P()$ is a proper function)
- **actual parameters**
- **control link** (dynamic link) — points to the AR of $P$’s **caller**.  
  e.g. if $F()$ calls $P()$, then the control link in $P$’s AR points to $F$’s AR.
Activation Records

- **access link (static link)** — points to the **most recent** AR of the **innermost** enclosing procedure in which $P()$ is defined.
  - used to implement the **lexical scope rule**.
  - Pascal has access links.
  - C does **not** need access links as C does **not** allow nested procedures. Thus, all variables are either local or global.
  - C++, however, **does** have nested scopes.
  - Scheme also uses lexical scope, so needs access links.
  - Lisp uses the **dynamic scope rule**, so doesn’t need access links.

- **saved machine status**: e.g.
  - registers values just before $P()$'s activation
  - return program counter so as to resume caller’s execution when $P()$ is done

- **local variables**

- **temporary storage**
Where to Put Activation Records?

The figure shows the memory layout of a C program during its execution.

Activations can be managed in the

- **stack**: traditional method for imperative language
- **heap**: if the activation of a procedure or function may be returned as a result, stored in a variable and used in an outer scope then its activation record must be stored in a heap so that its variables still exist when it is used.
  (e.g. functional programming languages)
A language that uses a stack to manage activation records is said to obey a **stack discipline** — last-in/first-out.

Thus, AR is also called a **stack frame**.

- **Advantage:** efficient
- **Disadvantage:** doesn’t allow function activations to be stored or passed around dynamically
When a procedure $Q()$ is called in the body of procedure $P()$, $P$ and $Q$ share responsibility in filling $Q$’s AR:

- $P$ evaluates the actual parameters and put their values in $Q$’s AR.
- $P$ stores information in $Q$’s AR so that when $Q$ is done, $P$ may continue execution from where it is left.
- $P$ set $Q$’s control link to point to its AR.
- $Q$ allocates space for its locals, and some temporary storage.
- The body of the procedure is executed.
- Control returns to the caller $P$, and $Q$’s AR, which is no longer needed, is popped out of the stack. The frame pointer is also reset from the control link.
Tail-recursive procedure: when the last executable statement in its body is the recursive call.

- Recursion simplifies programming, but naive implementation pays a price of worse efficiency since procedure call involves a lot of overhead.
- This problem can be eliminated by replacing any tail-recursive call with a loop.
- Scheme actually requires elimination of tail-recursion in its language specification.
int bsearch(int* a, int x, int lo, int hi)
{
    if (lo > hi) return NOT_FOUND;
    int k = (lo + hi) / 2;
    if (x == a[k]) {
        return k;
    } else if (x < a[k]) {
        return bsearch(a, x, lo, k-1);
    } else if (x > a[k]) {
        return bsearch(a, x, k+1, hi);
    }
}

int bsearch(int* a, int x, int lo, int hi)
{
    while (1) {
        if (lo > hi) return NOT_FOUND;
        int k = (lo + hi) / 2;
        if (x == a[k]) {
            return k;
        } else if (x < a[k]) {
            // a = a;
            // x = x;
            // lo = lo;
            hi = k-1;
        } else if (x > a[k]) {
            // a = a;
            // x = x;
            lo = k+1;
            // hi = hi;
        }
    }
}
(define M (lambda (j k)

  (define P (lambda (x y z)

    (define Q (lambda ()

      (define R (lambda ()
        (P j k z))) ; end R

      (* (R) y))) ; end Q

      (+ (Q) x))) ; end P

    (P j k 2))) ; end M)
Part II

Parameter Passing
Parameter-Passing: Running Example

```c
int a[] = {1, 2, 3, 4};
void Swap(int ... x, int ... y)
{
    int temp = x;
    x = y;
    y = temp;
    a[1] = 0;       // nonlocal a[]
}

int main()
{
    int j = 1; Swap(j, a[j]);
}
```

- Result depends on the relation between the actuals and formals.
What does it mean by:

\[ x = x + 1; \]

- variable \( x \) is assigned the sum of 1 and the value of \( x \)
- \( \text{location}(x) \leftarrow \text{value}(x) + 1 \)
- \( \text{l-value}(x) \leftarrow \text{r-value}(x) + 1 \)
- the meaning of the variable “\( x \)” is overloaded
A **macro** preprocessor in C/C++ supports language extensions:

```
#define BUFFER_SIZE 1024
#define BIGGER(a,b) ((a)>(b) ? (a) : (b))
```

C++’s **inline functions** are better macros allowing type-checking:

```c
inline int Bigger(int a, int b) {
    return (a > b) ? a : b;
}
```

However, it is just a recommendation to the compiler to expand the procedure before compilation; the compiler might not do so!

- Macro expansion is more efficient: no overhead in procedure calls.
- Macro expansion cannot handle recursion
  ⇒ should be used only on simple codes
Scope rules of a language determine which declaration of a name “x” applies to an occurrence of “x” in a program.

- **static/lexical scope rules**: the binding of name occurrences to declarations is done statically, at compile time.

- **dynamic scope rules**: the binding of name occurrences to declarations is done dynamically, at run time.

- Most languages use lexical scope rule.

- Dynamic scope are used for macros and inline functions.
int main()
{
    int j; // apply to S1, S5, S6
    int k; // apply to S1, S2, S3, S4, S6
    S1;

    for (...)
    {
        int j; // apply to S2, S4
        S2;

        while (...)
        {
            int j; // apply to S3
            S3;
        }
        S4;
    }

    while (...)
    {
        int k; // apply to S5
        S5;
    }
    S6;
}
Renaming Principle of Local Variables:
Consistent renaming of local names in the source text does not change the meaning of a program.

- Under lexical scope rule, we can always rename local variables until each name has only one declaration in the entire program.

- Most-closely-nested rule: an occurrence of a name is in the scope of the innermost enclosing declaration of the name.
program dynamic_scope(input, output);
  var x : real;
procedure show;
  begin write(x) end;
procedure tricky;
  var x : real;
  begin x = 1.2; show end;
begin x := 5.6; show; tricky; end.

What is the output if lexical scope rule is used?
What is the output if dynamic scope rule is used?

Dynamic scope rule may be implemented by macros.
Call-by-Reference (CBR): Running Example

// Using C++ syntax
// Declare as: void Swap(int& x, int& y)
// Call as: Swap(j, a[j]);

x and j refer to the same object; // int& x = j;
y and a[j] refer to the same object; // int& y = a[j];
temp <- x; x <- y; y <- temp;
a[1] <- 0;

j =
a = { , , , , }

x is called an alias of j, and y an alias of a[j]
Call-by-Value (CBV): Example 7

```c
int square(int x) { return x*x; }
int main()
{
    int y = 8; y = square(y+y);
}
```

Under **CBV**, 

```c
    u <- y+y       // done before calling square()
    x <- r-value(u) // int x = u;
    result <- x*x
default
return result
```
// Using C syntax
// Declare as: void Swap(int x, int y)
// Call as: Swap(j, a[j]);

x <- j;  // int x = j;
y <- a[j];  // int y = a[j];
temp <- x; x <- y; y <- temp;
a[1] <- 0;

j =
a = { , , , }  

Actually NO swapping has happened.
```c
int a[] = {1, 2, 3, 4};
void Swap(int* x, int* y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
    a[1] = 0; // nonlocal a[]
}

int main()
{
    int j = 1; Swap(&j, &a[j]);
}
```
// Using C syntax
// Declare as: void Swap(int* x, int* y)
// Call as: Swap(&j, &a[j]);

x <- l-value(j);    // int* x = &j;
y <- l-value(a[j]);  // int* y = &a[j];
temp <- r-value(object that x points to);
l-value(object that x points to)
    <- r-value(object that y points to);
l-value(object that y points to) <- temp;
a[1] <- 0;

j =
a = { , , , }
// C, C++ don’t use this; but assuming C++ syntax
// Call as: Swap(j, a[j]);

x <- r-value(j);       // Copy in the values
y <- r-value(a[j]);

temp <- x; x <- y; y <- temp; // Execute procedure
a[1] <- 0;

l-value(j) <- x;       // Copy out the results
l-value(a[j]) <- y;

j =
a = { , , , }
CBVR = CBR if the called procedure does not use any nonlocal variables.

CBVR may differ from CBR if the called procedure has more than one way of accessing a location in the caller.

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

if CBR: \( i = \), \( j = \)

if CBVR: \( i = \), \( j = \)
Call-by-Name (CBN): Running Example

// C, C++ don’t use this; but assuming C++ syntax
// Call as: Swap(j, a[j]);

// textually substitute j for x, a[j] for y
int temp = j;
j = a[j];
a[j] = temp;
a[1] = 0;

j =
a = { , , , , }

CBN is NOT the same as macro expansion
program TRY;
    int n; n = 10;
procedure P(x);
    begin int i; i = i + n; x = x + n; end;

begin
    int i, n; int A[10];
    i = 3; n = 5;
    P(A[i]);
end;

- CBN does more than just textual substitution.
if we simply substitute $A[i]$ for $x$ in $P(x)$

\[
i = i + n; \ A[i] = A[i] + n;
\]

$\Rightarrow$ conflict between the actuals ($A[i]$) and locals ($i$)
$\Rightarrow$ renaming locals in the procedure body of $P(x)$
$\Rightarrow$ int $j; j = j + n; A[i] = A[i] + n$;

if we simply do macro expansion in the main program

\[
i = 3; \ n = 5;
\]
\[
j = j + n; \ A[i] = A[i] + n;
\]

$\Rightarrow$ conflict between $n$ of main program and $n$ of $P(x)$
$\Rightarrow$ renaming locals in the caller of $P(x)$
$\Rightarrow$ int $i, m; i = 3; m = 5; j = j + n; A[i] = A[i] + n$;
### Summary on Parameter Passing

<table>
<thead>
<tr>
<th>Method</th>
<th>What is Passed</th>
<th>Language</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBV</td>
<td>value</td>
<td>C, C++</td>
<td>simple, passed parameters will not change</td>
</tr>
<tr>
<td>CBR</td>
<td>address</td>
<td>FORTRAN, C++</td>
<td>be careful: passed parameters may change</td>
</tr>
<tr>
<td>CBVR</td>
<td>value + address</td>
<td>FORTRAN, Ada</td>
<td>can be better than CBR, but more expensive</td>
</tr>
<tr>
<td>CBN</td>
<td>text</td>
<td>Algol</td>
<td>complicated; not used anymore</td>
</tr>
</tbody>
</table>