# Programming Languages 2012: Imperative Programming: Procedures

(Based on [Sethi 1996])

Yih-Kuen Tsay

## 1 Introduction

### Procedures

- *Procedures* are a construct for giving a name to a piece of code; the piece is referred to as the procedure *body*.
- When the name is called, the body is executed. Each execution of the body is called an *activation* of the body.
- Two forms of procedures:
  - *function procedures* or simply *functions* Functions extend the built-in operators of a language.
  - *proper procedures* or simply *procedures* Procedures extend the built-in actions or statements.

### Elements of a Procedure

- A procedure declaration makes explicit the elements or parts of a procedure:
  - *procedure name*, a name for the declared procedure
  - *formal parameters*, placeholders for actual parameters
  - *result type*, which is optional
  - *procedure body*, consisting of local declarations and a statement
- Example:

```
function square (x : integer) : integer;
begin
   square := x * x
end
```

| | |
|---|---|
| Procedure Name: | *square* |
| Formal Parameter: | $x$, of type **integer** |
| Result Type: | **integer** |
| Procedure Body: | return the value of $x * x$ |

### Benefits of Procedures

- The user of a procedure needs to know *what* a procedure does, not *how* the procedure works.
- The benefits of procedures include the following:
  - *Procedure abstraction.* abstract away from implementation details
  - *Implementation hiding.* changes can be made locally
  - *Modular programs.* divide and manage larger programs
  - *Libraries.* a way to extend the language

## 2 Parameter-Passing Methods

### Parameter-Passing Methods

- *Parameter passing* refers to the matching of actuals with formals when a procedure call occurs.
- Possible interpretations of a procedure call like $P(A[i])$ include the following:
  - *Call-by-value*. Pass the value of $A[i]$.
  - *Call-by-reference*. Pass the location of $A[i]$.
  - *Call-by-value-result* (*copy-in/copy-out*).
  - *Call-by-name*. Pass the text $A[i]$ itself, while avoiding "name clashes."

**Call-by-Value**

- Under call-by-value, a formal parameter corresponds to the value of an actual parameter. Call-by-value is the primary parameter-passing method in C and Pascal.

- Example:

  **procedure** $muchAdo(x,\ y : T)$; **var** $z : T$; **begin** $z := x;\ \ x := y;\ \ y := z;$ **end**

- A call $muchAdo(a, b)$ has the following effect:

```
x := a;                        { pass the value of a to x }
y := b;                        { pass the value of b to y }
z := x;  x := y;  y := z;      { a and b are unchanged }
```

- The program segment does not change $a$ or $b$, though the values of $x$ and $y$ are indeed exchanged.

**Call-by-Reference**

- Under call-by-reference, a formal parameter becomes a synonym for the the location of an actual parameter.

- Example:

  **procedure** $swap$(**var** $x$, $y$ : **integer**); **var** $z$ : **integer**; **begin** $z := x;\ \ x := y;\ \ y := z$; **end**

- A call $swap(i, A[i])$ is implemented as follows:

  make the location of $x$ the same as that of $i$; make the location of $y$ the same as that of $A[i]$; $z := x;\ \ x := y;\ \ y := z;$

- If $i$ is 2 and $A[2]$ is 99, the effect of these statements is

  $z := 2;\ \ i := 99;\ \ A[2] := z;$

  Thus, these assignments exchange the values of $i$ and $A[2]$.

**Call-by-Reference (cont.)**

- The only parameter-passing method in C is call-by-value; however, the effect of call-by-reference can be achieved using pointers.

- Example:

  **void** $swapc$(**int** $*px$, **int** $*py$) { **int** $z$; $z = *px;\ \ *px = *py;\ \ *py = z;$ }

- A call $swapc(\&a, \&b)$ is implemented as follows:

  $px = \&a;\ \ py = \&b;\ \ z = *px;\ \ *px = *py;\ \ *py = z;$

  These assignments exchange the values of $a$ and $b$.

**Call-by-Value-Result**

- *Call-by-value-result* is also known as *copy-in/copy-out* because (a) the actuals are initially copied into the formals and (b) the formals are finally copied back to the actuals.

- Ada, for example, supports three kinds of parameters:

  - **in** parameters, corresponding to value parameters.
  - **out** parameters, corresponding to just the copy-out phase of call-by-value-result.
  - **in out** parameters, corresponding to either reference parameters or value-result parameters, at the discretion of the implementation.

- Legal Ada programs are expected to have the same effect under call-by-reference and copy-in/copy-out.

**By-Value-Result vs. By-Reference**
Consider a contrived program fragment:

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

**By-Value-Result vs. By-Reference (cont.)**

- Under call-by-value-result, the procedure *foo* has two ways of changing the value of $i$:
  1. directly through an assignment to $i$ and
  2. indirectly trough the copy-out of the formal $x$.
- The indirect change will undo the effect of the direct assignment.

```
px := &i;  { save the location of actual i }
py := &j;  { save the location of actual j }
x := i;    { copy-in value of actual i into formal x }
y := j;    { copy-in value of actual j into formal y }
i := y;    { change value of i }
*px := x;  { copy-out x, thereby restoring i }
*py := y;
```

- Call-by-reference, on the other hand, will change the value of $i$.

## 3 Scopes of Names

**Scope Rules for Names**

- Names in programming languages can denote anything, including constants, variables, types, and procedures. A declaration of a name introduces a new sense in which a name is used.
- The *scope rules* of a language determine which declaration of a name $x$ applies to an occurrence of $x$ in a program.
- Two kinds of scope rules:
  - lexical scope rules (also known as static scope rules)
  - dynamic scope rules

**Lexical Scope vs. Dynamic Scope**

```
program L;
var n : char;                 { n declared in L }
  procedure W;
  begin write(n) end;         { occurrence of n in W }
  ...
  procedure D;
  var n : char;               { n declared in D }
  begin n := 'D'; W end;      { W called within D}
begin { L }
  n := 'L'; W; D              { W called from program L }
end.
```

Under lexical scope, the program produces the output

LL

Under dynamic scope, the program produces the output

LD

**Renaming of Locals and Lexical Scope**

After renaming the local variable $n$ in procedure $D$ to $r$,

```
program L;
var n : char;                 { n declared in L }
  procedure W;
  begin write(n) end;         { occurrence of n in W }
  ...
  procedure D;
  var r : char;               { r declared in D }
  begin r := 'D'; W end;      { W called within D}
begin { L }
  n := 'L'; W; D              { W called from program L }
end.
```

The program produces (under lexical or dynamic scope) the output

LL

The renaming principle: consistent renaming of local names has no effect on the computation set up by a program.

**Macro Expansion and Dynamic Scope**

- If a procedure body is simply copied or substituted at the point of call, we get dynamic scope.
- A *macro processor* does the following:
  1. Actual parameters are textually substituted for the formals.
  2. The resulting procedure body is textually substituted for the call.
- Using macro expansion for the call $W$,

```
procedure D;
var n : char; begin n := 'D'; W end;
```

will become

```
procedure D;
var n : char; begin n := 'D'; writeln(n) end;
```

- The occurrence of $n$ in *writeln*($n$) is “captured” by the declaration of $n$ in procedure $D$. The output of the macro-expanded program will be the same as that under dynamic scope.

**Call-by-Name and Lexical Scope**

- Call-by-name and call-by-value were the two parameter-passing methods in Algol 60.
- The rules for call-by-name were carefully specified to get lexical scope:
  1. Actual parameters are textually substituted for the formals. Possible name conflicts between names in the actuals and local names in the procedure body are avoided by renaming the locals in the body.
  2. The resulting procedure body is substituted for the call. Possible conflicts between non-locals in the procedure body and locals at the point of call are avoided by renaming the locals at the point of call.

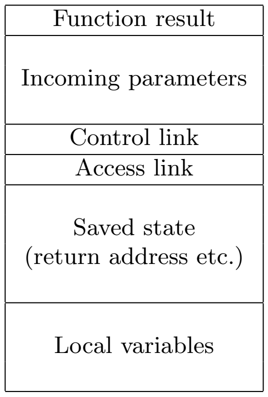
**Nested Scopes**

```
int main(···)
{
    int i;
    for(···);
    {
        int c;
        if(···);
        {
            int i;
            ···
        }
        ···
    }
    while(···)
    {
        int i;
        ···
    }
    ···
}
```

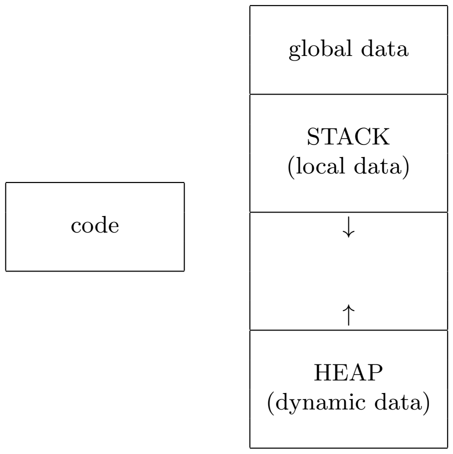
# 4 Activation Records

**Activation Records**

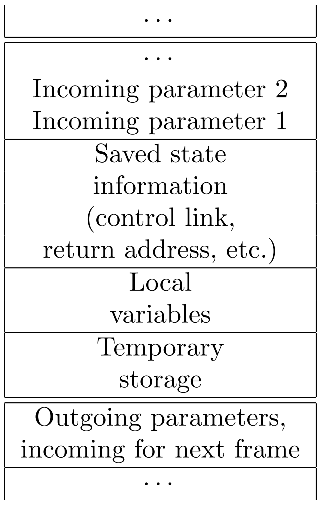
- Data needed for an activation of a procedure is collected in a record called an *activation record* or *frame*.
- The elements of an activation record:



**Memory Layout for C Programs**



**Activation Records for C**



# 5 Tail-Recursion

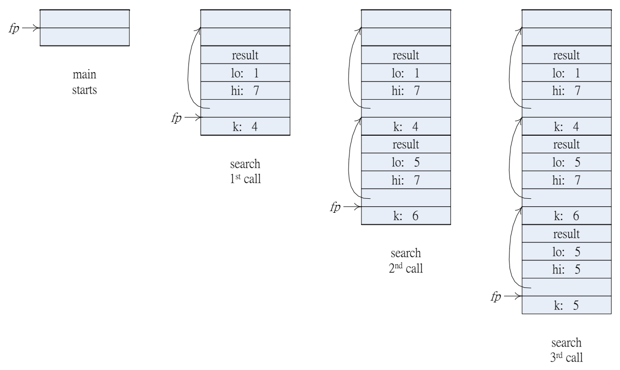
### Tail-Recursion

- When the last statement executed in the body of a procedure is a recursive call, the call is said to be *tail recursive*.
- A procedure as a whole is *tail recursive* if all its recursive calls are tail recursive.
- Tail-recursive calls can be eliminated and replaced by control flow within the procedure, thereby avoiding the overhead of a call.

### A Tail-Recursive Binary Search

```
#include <stdio.h>
int yes = 1, no = 0;
#define N 7
int X[] = { 0, 11, 22, 33, 44, 55, 66, 77 };
int T;
int search(int lo, int hi) {
    int k;
    if (lo > hi) return no;
    k = (lo + hi) / 2;
    if (T == X[k]) return yes;
    else if (T < X[k]) return search(lo, k-1);
    else if (T > X[k]) return search(k+1, hi);
}
int main(void) {
    scanf("%d", &T);
    if (search(1,N)) printf("found\n");
    else printf("not found\n");
    return 0;
}
```

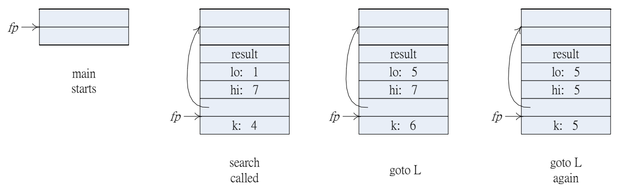
### A Tail-Recursive Binary Search (cont.)



### Eliminating Tail Recursion

```
#include <stdio.h>
int yes = 1, no = 0;
#define N 7
int X[] = { 0, 11, 22, 33, 44, 55, 66, 77 };
int T;
int search(int lo, int hi) {
    int k;
L:  if (lo > hi) return no;
    k = (lo + hi) / 2;
    if (T == X[k]) return yes;
    else if (T < X[k]) hi = k-1;
    else if (T > X[k]) lo = k+1;
    goto L;
}
int main(void) {
    scanf("%d", &T);
    if (search(1,N)) printf("found\n");
    else printf("not found\n");
    return 0;
}
```

### Eliminating Tail Recursion (cont.)


fp
main starts
result
lo: 1
hi: 7
fp
k: 4
search called
result
lo: 5
hi: 7
fp
k: 6
goto L
result
lo: 5
hi: 5
fp
k: 5
goto L again