COMP3031: Procedure Activations

Qiong Luo

HKUST

Procedures

- A *procedure* is a construct for giving a name to a piece of code, its body.
- When a procedure is called the body is executed. Each execution of the body is called an *activation* of the body.
- Functional procedures – extend the built-in operators. Like expressions, return a value when called.
- Proper procedure – extend the built-in actions or statements. Normally do not return anything. Affect states through side effects.

Our main concerns are what happens when a procedure is called, and its body activated.

Elements of a Procedure

A procedure normally has the following elements:

- a name for the declared procedure,
- a body consisting of local declarations and a statement,
- the formal parameters, which are placeholders for actuals, and
- an optional result type.

```
/* square(x) computes the square of x */
int square(int x)
{
  int result;
  result = x*x;
  return result;
}
```

Benefits of Procedures

Procedures are one way of organizing programs. The user of a procedure needs to know what a procedure does, not how it works. This separation of "what" from "how" allows a program to be partitioned into pieces that can be understood in isolation. This leads to

- implementation hiding. Algorithms isolated within a procedure can later be modified by changing only the procedure.
- modular programs.
- Libraries.

Issues Raised by Procedures

Consider the following procedure call:

```
main()
{ int result;
  ...
  result = 4;
  result = square(result);
  ...
  }
```

- How is the actual **result** passed into the procedure body?
- Is this **result** the same as the one declared locally?
- How are name conflicts like this resolved?
- What happens if a procedure is activated (called) within its own body?

Recursion: Multiple Activations of a Procedure

A procedure is *recursive* if it can be activated from within its own procedure body, either by calling itself, or indirectly through calls to other procedures.

```
/* fact(x) computes factorial of x */
int fact(int x)
{ if (x == 0) return 1
  else return x*fact(x-1);
 }
```

```
fact(5) = 5*fact(5-1) = 5*(5-1)*fact(5-1-1) = ...
```

Recursive procedures make many issues such as name conflicts more acute.

L-Values and R-Values of Variables

To understand parameter passing methods, a good understanding of variables and their implementation is helpful.

- A variable denotes certain value, and is implemented by a memory location.
- The l-value of a variable is this memory location.
- The r-value of a variable is what is stored in this location, i.e. the value denoted by this variable.
- L-values and r-values are so called because of their roles in assignments such as: `x = x + 1`
- If x and y have the same l-values, then they are *aliases* or *synonyms*.
- If x is an alias to y, what's x's value after the assignment `y = 2`?

Parameter-Passing Methods

The formal parameters in a procedure declaration are placeholders for the actual parameters supplied in a procedure call:

- The call *square*(2) computes 4 by evaluating the body of the procedure `x*x` with 2 in place of x.

Parameter passing refers to the matching of actuals with formals when a procedure call occurs:

- Call-by-value pass the values (r-values) of the actuals.
- Call-by-reference pass the locations (l-values) of the actuals.
- Call-by-name pass the texts of the actuals.

Call-by-Value

Suppose a procedure P has one formal, x. Under call-by-value, a call P(E) is evaluated as follows:

```
x = E;   {pass the value of E to x}
execute the body of P
if P is a function, return a result
```

For example, the call *square*$(2+1)$ is evaluated as follows:

```
x = 2+1;          /*pass the value of 2+1 to x*/
result = x*x;     /*execute the body*/
return result;    /*return a result*/
```

Call-by-Value (Cont'd)

- Call-by-value is very simple - simple to understand and implement.
- It works most of time, and is the main parameter passing method of C, C++, and Pascal.
- The following example shows that sometimes we need other methods.

The following procedure tries to exchange the values of x and y:

```
muchAdo(int x, int y)
{ int z;
  z = x; x = y; y = z; }
```

Now what is the outcome of the following program?

```
main()
{ int a=1,b=2;
  muchAdo(a,b);
  printf("%d\n",a); printf("%d\n",b); }
```

Call-by-Reference

- Under call-by-reference, a formal parameter becomes a synonym (alias) for the location (l-value) of an actual: Suppose a procedure P has one formal, x. Under call-by-reference, a call P(y) is evaluated as follows:

```
assign x the location of y; (pass the l-value
  of y to x so that x and y are aliases)
execute the body of P
if P is a function, return a result
```

- Under this method, an actual need to have an l-value.
- If the procedure *square* were to use call-by-reference, then the call *square*(2) would be illegal. Instead, you have to do something like: $y = 2$; *square*(y).

Call-by-Reference (Cont'd)

- In Pascal, the keyword *var* makes a variable a reference parameter and call-by-reference is used:

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

- In C++, the address operator & makes a variable a reference parameter and overrides the default pass-by-value mechanism:

```
void swap( int &x, int &y) {
  int z;
  z = x; x = y; y = z; }
```

- In C, the only parameter passing method is call-by-value. But call-by-reference can be simulated by pointers.

```
void swap( int * px, int * py) {
  int z;
  z = *px; *px = *py; *py = z; }
int a = 1, b=2; swap(&a, &b);
```

Call-by-Value-Result (Copy-in/Copy-out)

Call-by-value-result, a method used by Ada, works as follows:

- Actuals such as 2+3 that do not have l-values are passed by value.
- For actuals with locations:
  - Copy-in phase: the values of actuals are passed to formals; the locations are saved for the copy-out phase.
  - Copy-out phase: after the body is executed, the final results of the formals are copied back out to the locations of the actuals computed.

Example: under call-by-value-result, $swap(3, b)$ assigns 3 to b, and $swap(i, A[i])$ indeed changes the value of i and $A[i]$:

```
px := &i;  (save the l-value of i)
py := &A[i]; (save the l-value of A[i])
x = i; y = A[i]; (copy-in the values of actuals)
z = x; x = y; y = z; (execute the body)
*px = x; *py = y; (copy-out the results)
```

Call-by-Value-Result (Cont'd)

- Call-by-value-result is used primarily in Ada, and similar to call-by-reference.
- Legal Ada programs are expected to have the same effect under call-by-reference and call-by-value-result.
- Call-by-reference and call-by-value-result may differ when there are aliases, as the following example shows. (By definition, Ada programs cannot have aliases. But it is hard to check if a program uses aliases or not.)

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

After $foo(i, j)$, the value of i changes if Call-by-reference is used, but not if call-by-value-result is used.

Scope Rules for Names

```
int x=1;
main()
{ int x=2;
  { int x=3;
    printf("%n",x);
    test;  }
  test;
  printf("%n",x);
}
test()
{ printf("%n",x);}
```

The *scope rules* of a language determine which declaration of a name x applies to an occurrence of x in a program:

$$\text{Name occurrence} \overset{scope\ rule}{\longrightarrow} \text{Declaration}$$

Scope Rules

There are two basic scope rules: lexical (static) and dynamic.
Under *lexical scope rules*, the binding of name occurrences to declarations can be done statically, at compile time, for all programs in a language. So this rule is also know as *static scope rule*.
Under *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 so-called inline functions.

Dynamic Scope Rule

Dynamic scope rule is closely related to *macro expansion* (*call-by-name under dynamic scope*):

- actuals are textually substituted for the formals;
- the resulting body is textually substituted for the call.

Example:

```
int x = 1;                    int x = 1;
main() {                      main() {
 int x = 2;          ==>      int x = 2;
 p;}                           printf("%d",x);}
void p() {
 printf("%d",x);}
```

Dynamic Scope (Cont'd)

- C and C++ uses a macro preprocessor to support language extensions such as:

```
#define MAXBUF 4
#define ABSDIFF(a, b) ((a)>(b) ? (a)-(b) : (b)-(a))
#include <stdio.h>
```

- C++ allows a procedure to be declared as *inline* as a *recommendation* to compiler to expand the procedure before compilation:

```
inline int abs(int i) {
  return ( i<0 ? -i : i ); }
```

- Macro expansion is normally more efficient: no overhead in procedure calls.
- Macro expansion cannot handle recursion, should be used only on simple code.

Lexical Scope Rule

A language using this rule should satisfies the following renaming principle:

> Renaming of local variables. *Consistent renaming of local names in the source text has no effect on the computation set up by a program.*

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

```
int x = 1;                    int x = 1;
main() {                      main() {
 int x = 2;          ==>       int x1 = 2;
 p;}                           p;}
void p() {                    void p() {
 printf("%d",x);}              printf("%d",x);}
```

Call-by-Name under Lexical Rule

Call-by-name (under lexical rule) was used by Algol 60. It is complicated and now primarily of historical interest:

- Actuals are textually substituted for the formals. Possible conflicts between names in the actuals and local names in the procedure body are resolved by renaming locals in the procedure body:

```
int y = 0;
int p(int x) {
  int i;
  for (i = 0; i < x; i++)
    y = y + x; }
main() {                        main() {
  int i = 3;                     int i = 3;
  p(i);                  ===>    { int j;
}                                  for (j = 0; j < i; j++)
                                   y = y + i; }}
```

Call-by-Name under Lexical Rule

- The resulting procedure body is substituted for the call. Possible conflicts between nonlocals in the procedure body and locals at the point of call are avoided by renaming the locals in the procedure body:

```
int x = 1;                    int x = 1;
main() {                      main() {
 int x = 2;          ==>      int x1 = 2;
 p;}                           printf("%d",x);}
void p() {
 printf("%d",x);}
```

- Morale: rename locals to avoid the capture of variables declared in the outer scopes.

Lexical Scopes for Nested Structures

Many programming constructs allow nesting. Example: C blocks.

```
int main(...)
{ int i;   /* applies to S1,S2,S4,S5 */
  S1
    { int c:
      S2
      { int i; /* applies to S3 */
        S3 }
      S4 }
  S5
  { int i; /* applies to S6 */
    S6 }
}
```

Lexical Scope for Nested Structure

The scoping rule for variable declarations in C can be stated using the following rule:

> Most-closely-nested rule: *an occurrence of a name is in the scope of the innermost enclosing declaration of the name.*

This is a lexical rule because the scope of a variable declaration is defined in terms of (nesting of) the source texts. It applies to any constructs with the following properties:

- The constructs nest one inside the other.
- The construct can contain declarations and the scope of a declaration is contained within that construct – normally extends from the point of declaration to the end of the construct.
- The redeclaration of a name in a nested construct creats a hole in the scope of any enclosing declarations of the name.

Activation Records

An activation record (AR) or *a frame* is storage used to store data needed for an activation of a procedure. Whenever a procedure is activated, an AR associated with it is created. An AR normally contains the following elements:

- Parameters.
- Function result.
- Local variables.
- Temporary storage needed to evaluate expressions. Example: (a + b)∗c is evaluated by t1 := a+b; t2 := t1∗c. t1 and t2 are temporary storage.
- Storage needed to manage activation records: where's the calling procedure's AR, where's the locations for global variables, etc.

AR (Example)

The storage needed to implement the following PASCAL-like procedure:

```
function read(u : link) : boolean;
var i: integer; c: char;
begin ... end;
```

includes storage for

- formal parameter u;
- boolean result;
- variables i and c;
- any temporary variables;
- anything needed for linking to other ARs.

Control and Access Links

ARs can be managed by maintaining two kinds of links:

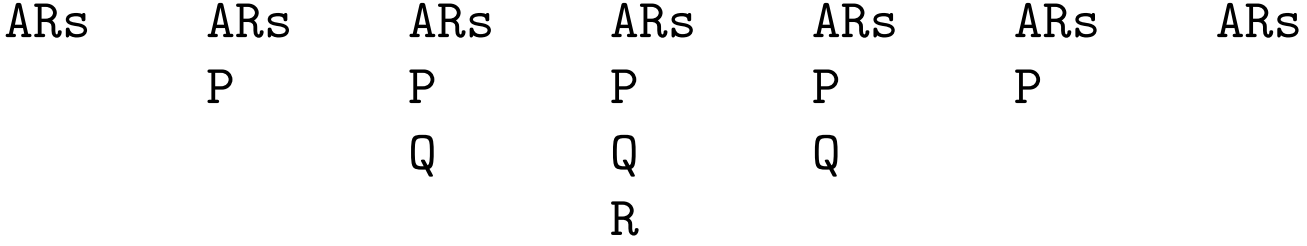
- Control (dynamic) link: points to the AR of the run time caller. Example: if P calls Q, then the control link in Q's AR points to P's AR.
- An access (static) link: points to the AR of the innermost enclosing procedure in which P is defined. It is used to implement lexically scoped languages which allow procedure declarations to be nested, like in Pascal. Not needed for C. Example: access links are needed to properly find the non-local variable *n* when W is called in D:

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

Managing Activation Records

- A general technique for managing ARs is to allocate storage for them in an area called the *heap*.
- The ARs stay on the heap as long as they are needed.
- A technique called *garbage collection* is used to automatically reclaim storage for ARs that are no longer needed.
- The allocation and deallocation of ARs in the heap are normally implemented using stacks for imperative PLs.

The following picture illustrates the states of the heap when P calls Q and Q calls R:



Activation Records (Continued)

A language that permits stack like allocation and deallocation of ARs is said to obey a *stack discipline.*

- Imperative PLs generally obeys the stack discipline.
- The stack discipline imposes restrictions on language design: Procedures must be declared in the source text. They cannot be created in the run time, returned as values.

Case Study: C Procedures

Procedure implementation in C is made easier by not allowing nested procedure declarations:

- There are only two places to look for a declaration of a name in a procedure: either within this procedure or outside all procedures (global declarations).
- Only control links are needed, not access links.

Storage management for Pascal requires access links and represents the general case of storage management for imperative language that obey the stack discipline.

C Procedures (Continued)

The memory layout for a C program consists of:

- a static memory area to store global data (global variables).
- A stack for storing local data. This stack grows and shrinks as activations start and end.
- A heap for dynamic data that is allocated and deallocated through procedures and pointers.

C Procedures (Continued)

An activation record, also called *a frame*, for an activated C procedure has the following elements:

- Incoming (actual) parameters.
- Returned values for functions.
- Saved state information including the control link. Used by the caller to continue execution after the completion of the procedure call.
- Storage for local variables and temporary storage for expression evaluation.

C Procedures (Continued)

When a procedure P is called in the body of procedure Q, the following actions occur:

- The caller (Q) evaluates the actual parameters and saves the results to the AR.
- The caller stores saved-state-information in the AR.
- The callee (P) allocates space for its locals, and sets aside some temporary storage.
- The body of the procedure is executed. This may in turn call other procedures.
- Control returns to the caller, and the AR, which is no longer needed, is popped out of the stack.

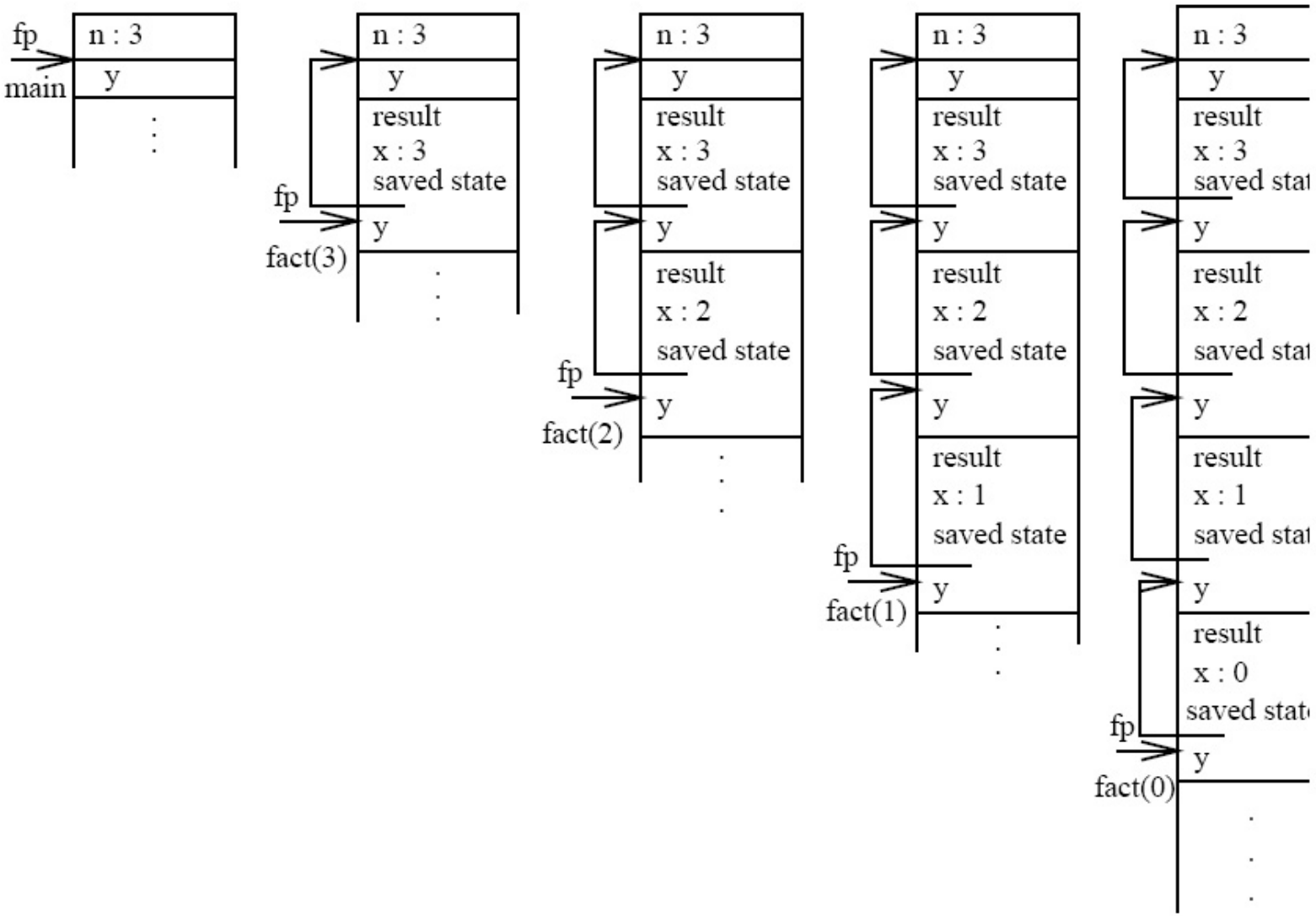
C Procedures (Example)

Consider the following program that calls a recursive procedure:

```
int n = 3;
main() {
 int y;
 y = fact(n);
 printf("%d\n",y); }
fact(int x) {
 int y;
 if (x == 0) y = 1;
 else y = x*fact(x-1);
 return y; }
```

The following slide illustrates the AR storage management during the execution of the main program.

AR Flow (Example)



AR Flow (Example, Cont'd)