Chapter 9

Subprograms

"Modular Programming"

"Subprograms are the fundamental building blocks of programs and are therefore among the most important concepts in programming language design."

9.1 Introduction

- Two fundamental abstraction facilities
  - Process abstraction : (subprogram)
    - ⇔ procedure call :
      - ⇒ an abstraction of a collection of statements
  - Data abstraction :
    - ⇔ abstract data type
- In a modern programming language, a collection of statement is *reused* and ends up as a collection of machine instructions in memory
  - memory space saving, coding time saving
- Such reuse is also an abstraction if the collection is placed in a program by a statement that "calls" that collection.
  - Instead of explaining how some computation is to be done, that explanation (the collection of statement) is enacted by a "call" statement, effectively abstracting away the details
- caller *vs*. callee
- *Procedure* vs. *Macro*

caller

```
main() {
  …
  sub1() ;
  …
}
```

callee

```
sub1() {
  …
  …;
  …
}
```

9.2 Fundamentals of Subprograms

(1) General Subprogram Characteristics

- **Basic characteristics of subprograms**
  - Each subprogram has a single entry point
  - the calling program unit is suspended during the execution of the called subprogram, which implies that there is only one subprogram in execution at any given time
  - Controls always returns to the caller when the subprogram execution terminates

(2) Basic Definitions

- **Basic definitions**
  - *Subprogram definition* describes the action of the subprogram abstraction
  - *Subprogram call* is the explicit request that the subprogram be executed
  - a subprogram is *active* if, after having been called, it has begun execution but has not yet completed that execution

caller

```
main() {
  …
  sub1() ;
 …
}
```

callee

```
sub1() {
   …
   …;
   …
}
```

Subprogram definition

- ***subprogram header :*** the first line of the definition
  - ⇔ objectives
    - ⇒ specifies that the following syntactic unit is a subprogram definition
    - ⇒ provides the name for the subprogram
    - ⇒ may *optionally* specify a list of parameters
  - ⇔ In FORTRAN,

```
SUBROUTINE ADDER (parameters)
```

  - ⇔ In Ada,

```
procedure ADDER (parameters) is
```

  - ⇔ In C
    - ⇒ C has only one kind of subprogram, the function
    - ⇒ header is recognized by its context

```
adder (parameters)
```

```
int a, b ;
int p(int i) {
   static int a = 0, p = 0 ;
   a = a+1; b = 1; p = p+2;
   return(p) ;
}
void main(void) {
   int i, j ;
   a = p(i)+p(i) ;
}
```

(3) Parameters

- two ways that a subprogram can gain access to data
  - through *direct access to nonlocal variables* (declared elsewhere but visible in the subprogram, or variables in the reference environment)
    - ⇔ extensive access to nonlocal causes reduced reliability
  - through parameter passing
    - ⇔ a parameterized computation (subprogram이 어떤 computation을 할지는 parameter에 의하여 결정된다.)

```
void main(void) {
   int i, sum=0 ;

   for (i=1;i<1001;i++){
     sum=sum + factorial(i);
}
```

```
int factorial(int n) {

   if (n==1) return(1)
   else return(n * factorial(n-1));
}
```

- In some situations, it is convenient to be able *to transmit computations*, rather than *data*, as parameters to subprograms
  - the name of subprogram may be used as a parameter
- Formal parameters and Actual Parameters
  - *formal parameters* : the parameters in the subprogram header
  - *actual parameters* : a list of parameters in subprogram call that would be bound to the formal parameters of the subprogram

- Parameter Passing
  - *Positional parameters* : the binding of actual parameters to formal parameters is done *by simple position*.
    - ⇔ the first parameter is bound to the first formal parameters
  - *Keyword parameters* : the name of formal parameter to which an actual parameter is to be bound is *specified with actual parameter*
    - ⇔ In Ada,

```
SUMER (LENGTH => MY_LENGTH,
       LIST => MY_LIST,
       SUM => MY_SUM) ;
```

⇒ the user of the subprogram must know the names of formal parameters

- In C++ and Ada, formal parameters can have default values
  - It is used if no actual parameter is passed to the formal parameter in the subprogram header
    - ⇔ In Ada,

```
procedure COMPUTE_PAY (
            INCOM : FLOAT ;
            EXEMPTION : INTEGER := 1 ;
            TAX_RATE : FLOAT;
            ....) is
```

- the number of actual parameters in a call must match the number of formal parameters in the subprogram definition header. (Exception : C language)

```
main() {
  int i, j ; char c;
  …
  printf("%d %d", i,j);
  Printf("%c", c);
}
```

(4) Procedures and Functions

- *Procedure*
  - collections of statements that define parameterized computations
  - It defines, in effect, *new statements*
  - two ways to pass the results to caller
    - ⇔ by changing visible variables (excluding formal parameters)
    - ⇔ by changing formal parameters that allows the transfer of data to the caller
- *Functions*
  - *functions are called by appearances of their names*, along with the required actual parameters, in expressions (user defined operator)
  - the value produced by a function's execution is returned to calling code, effectively replacing the call itself
  - In Pascal,

```
function power (base, exp : real) : real ;
   begin
       .....
   end
.......
result := 3.4 * power(10.0, x)
```

In FORTRAN,

```
Result = 3.4*10.0**x
```


9.3 Design Issues for Subprograms

- Issues
  - What parameter-passing method or methods are used ?
    - ⇔ *Pass-by-Value*
    - ⇔ *Pass-by-Result*
    - ⇔ *Pass-by-Value-Result*
    - ⇔ *Pass-by-Reference*
    - ⇔ *Pass-by-Name*
  - Are the type of the actual parameters checked against the types of the formal parameters ?
  - Are local variables statically or dynamically allocated ?
  - What is the referencing environment of a subprogram that has been passes as a parameter ?
  - If subprograms can be passed as parameters, are the types of parameters checked in calls to the passed subprograms ?
  - Can subprograms be overloaded ?
  - Can subprograms be generic ?
  - Is either separate or independent compilation possible ?

9.4 Local Referencing Environments

- Variables that are declared inside subprograms are called *local variables*,
  - ⇔ *access to local variable is usually restricted to the subprogram in which they are declared (scope of local variable)*
  - *Stack-dynamic local variables*
    - ⇔ bound to storage when the subprogram begins execution and unbound from storage when that execution terminates
    - ⇔ Advantages
      - ⇒ allows recursive subprograms
      - ⇒ storage sharing
    - ⇔ Disadvantages
      - ⇒ cost of the time to allocate, initialize, and deallocate such variables for each call
      - ⇒ indirect referencing -> slow
      - ⇒ do not allow history-sensitive procedures
  - *Static local variables*
    - ⇔ bound to storage when the program begins the execution
    - ⇔ Advantages
      - ⇒ allows fast referencing
      - ⇒ history-sensitive procedures
    - ⇔ but, do not allow *recursion*

- In ALGOL 60 and its descendant languages, local variables are by default stack-dynamic
  - ⇔ In C,

```
adder (list, listlen)
int list[], listlen ;
{ static int sum = 0 ; /* ➔ static local variable */
  int count ;          /* ➔ stack dynamic local variable */
  for (count = 0 ; count < listlen ; count++)
           sum = sum + list[count] ;
  return sum ;
}
```

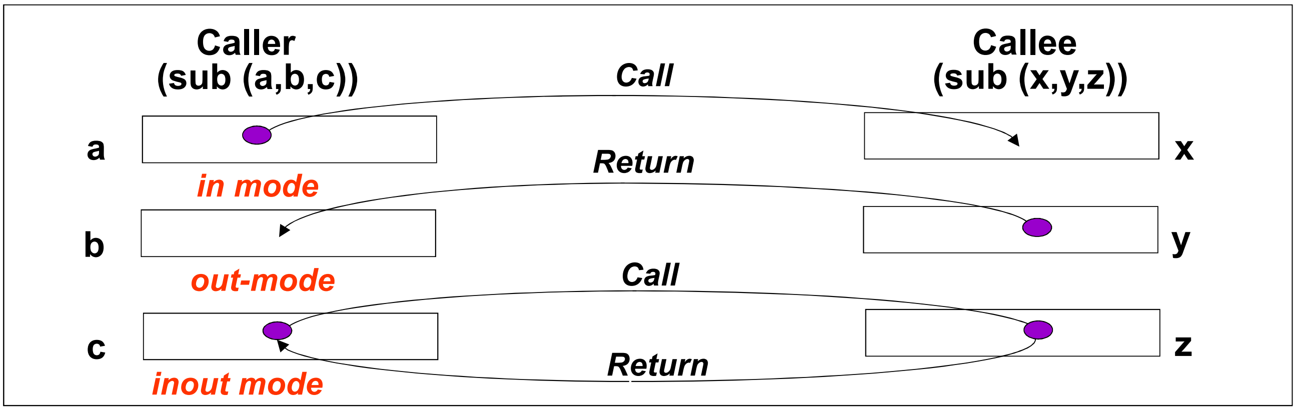
  - ⇔ In FORTRAN 77,
    - ⇒ No recursion -> all local variables are *static*

9.5 Parameter-Passing Methods

– the ways in which parameters are transmitted to and/or from subprograms

(1) Semantics Models of Parameter Passing

- Formal parameters are characterized by one of three distinct semantics models



① **in mode** : formal parameters can receive data from corresponding actual parameters
② **out mode** : formal parameters can transmit data to corresponding actual parameters
③ **inout mode** : both of them

- two conceptual models of how data transfers take place in parameter transmission
  – actual value is physically moved to
  – an access pass (pointer) is moved

(2) Implementation Models of Parameter Passing

– a variety of models has been developed by language designers to guide the implementation of three basic parameter transmission modes

- ***Pass-by-Value*** *(call-by-value)*
  – the value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local variable in the subprogram
  – provides *in-mode* semantics
  – normally implemented by actual data transfer
  – the extra storage and the move operations can be costly if the parameter is a large object, such as a long array

```
int p(int i[][100]) {
  ………;
}

void main(void) {
  int a[100][100];
  … … …
  p(a);
  … … …
}
```

- ***Pass-by-Result***
  - an implementation model of ***out-mode*** parameters
    - ⇔ no value is transmitted to the subprogram
  - the corresponding formal parameter acts as a local variable, but just before control is transferred back to the caller, its value is passed back to the caller's actual parameter, which *must be variable*
  - Problems
    - ⇔ the extra storage and move operation could be problems
    - ⇔ there can be an actual parameter collision
      - → what is the value of p1 after return ?

```
subroutine sub(x,y){
  x=3 ;
  y=5;
}
main() {
  int p1;
  sub(p1, p1);
  p1???
}
```

    - ⇔ the implementor may be able to choose between two different times to evaluate the address of the actual parameters

```
int index, list[10];

subroutine sub(a){
 index = 5;
 a = 3;
}
main() {
  index = 3;
  sub(list[index]);
}
```

index[3] *or* index[5] ?

- **Pass-by-Value-Result** (pass-by-copy)
  - an implementation model for ***inout-mode*** parameters in which actual values are moved
  - the value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local variable. At subprogram termination, the value of the formal parameter is transmitted back to the actual parameter
- **Pass-by-Reference**
  - a second implementation method for ***inout-mode*** parameters
  - transmits ***an access path***, usually just ***an address***, to the called subprogram
    - ⇔ the actual parameter is ***shared*** with the called subprogram
  - ***No copying overhead***, and ***no duplicate space***
  - ***Problems***
    - ⇔ accesses to the formal parameters will most likely be ***slower*** because of one more level of ***indirect addressing***
    - ⇔ inadvertent and erroneous change may be made to the actual parameter
    - ⇔ aliases can be created

```
procedure bigsub ;
   var global : integer ;
    procedure smallsub(var local:integer) ;
        begin
           global = 3; local = 5;
        end
    begin
       smallsub(global) ;
    end
```

```
procedure sub(var first, second : integer)
…
call sub(total, total);
…
```

- **Pass-by-Name**
  - an inout-mode parameter transmission method
  - the actual parameter is , in effect, *textually substituted* for the corresponding formal parameter in all its occurrences in the subprogram
  - a pass-by-name formal parameter is bound to access method at the time of the subprogram call, but the actual binding to a value or an address is delayed until the formal parameter is assigned or referenced (*late binding*)
  - the form of the actual parameter dictates the implementation method of pass-by-name parameters
    - ⇔ passing variable -> pass-by-reference
    - ⇔ passing constant -> pass-by-value
    - ⇔ passing array element (or expression with variable) -> the value of array element (expression) can change with each reference to the formal parameter

```
procedure BIGSUB ;
  integer GLOBAL ;
  integer array LIST[1..2] ;
  procedure SUB (PARAM) ;
      int PARAM ;
      begin
          PARAM := 3 ;
          GLOBAL := GLOBAL + 1 ;
          PARAM := 5 ;
      end ;
  begin
      LIST[1] := 2 ;
      LIST[2] := 2 ;
      GLOBAL := 1 ;
      SUB(LIST[GLOBAL]) ;
end ;
```


- **Jesen's Devices** (Single procedure can be used for a variety of purposes)
  - ⇔ passing an expression and one or more variables that appear in that expression as parameters to subprogram
  - ⇔ whenever one of the variable from parameters is changed in the subprogram, that change can cause a change of the values of later occurrences of the formal parameter that corresponds to the expression actual parameter

```
real procedure SUM (ADDER, INDEX, LENGTH) ;
     real ADDER ;
     integer INDEX, LENGTH ;
     begin
        real TEMPSUM ;
        TEMPSUM := 0.0 ;
        for INDEX := 1 step 1 until LENGTH do
           TEMPSUM := TEMPSUM + ADDER ;
        SUM := TEMPSUM
    end ;
```

```
SUM (A, I, 100)-> 100*A
  for I := 1 step 1 until 100 do
     TEMSUM := TEMPSUM + A ;
```

$$\sum_{i=0}^{100} (A[i])^2$$

```
SUM (A[i]*A[I], I, 100)
  for I := 1 step 1 until 100 do
     TEMSUM := TEMPSUM + A[I]*A[I];
```

SUM (A[I], I, 100) -> $\sum_{i=0}^{100} A[i]$

```
  for I := 1 step 1 until 100 do
     TEMSUM := TEMPSUM + A[I] ;
```

SUM (A[I]*B[I], I, 100) $\sum_{i=0}^{100} (A[i]*B[i])$

```
  for I := 1 step 1 until 100 do
     TEMSUM := TEMPSUM + A[I]*B[I] ;
```

– Interchanging the values of two given actual parameters (swapping)

```
procedure swap (FIRST, SECOND)
    integer FIRST, SECOND ;
    begin
       integer TEMP ;
       TEMP := FIRST ;
       FIRST := SECOND ;
       SECOND := TEMP
    end ;
```

Pass-by-name

```
swap(KK,II) ;

       TEMP = KK ;
       KK = II ;
       II = TEMP ;
```

OK !

```
swap(I,A[I]) ;

       TEMP := I ;
       I := A[I] ;
       A[I] = TEMP ;
```

OK ?

```
A[A[I]] = TEMP
```

– Pass-by-Name provides *great flexibility*, but *slow process* and *difficult to implement* and confuse both reader and writers of the program

(3) Parameter-Passing Methods of the Major Languages

- C
  - Pass-by-value
  - Pass-by-reference is achieved by using *pointers as parameters*
- C++
  - A special pointer type called *reference type* for pass-by-reference
- Java
  - All parameters are passed are *passed by value*
  - *Object parameters are passed by reference*
- Fortran 95+
  - Parameters can be declared to be in, out, or inout mode
- C#
  - Default method: pass-by-value
    - Pass-by-reference is specified by preceding both a formal parameter and its actual parameter with `ref`
- PHP: very similar to C#, except that either the actual or the formal parameter can specify ref
- Perl: all actual parameters are implicitly placed in a predefined array named `@_`

- **Pascal and Modula-2**
  - **default parameter-passing method is pass-by-value, and pass-by-reference can be specified by prefacing formal parameters with the reserved word *var***

```
procedure adder (var a : integer ; /* call-by-reference */
                     b : integer ; /* call-by-value */
                 var c : real) ;   /* call-by-reference */
```

- **Ada**

```
procedure ADDER (A : out INTEGER ;
                 B : in INTEGER ;
                 C : in_out FLOAT)   is
```

ADDER()의 body에서

```
aa = A ;      /* error */
B = aa  ;     /* error */
C = C + aa ;  /* OK */
```

  - **out : can be assigned, but not referenced**
  - **in : can be referenced, but not assigned**
  - **In_out : both**
- **Python**
  - **use pass-by-assignment (all data values are objects, so pass-by-object or pass-by-object-reference); the actual para is assigned to the formal para**

```
def spam(eggs):
    eggs.append(1)
    eggs = [2, 3]

ham = [0]
spam(ham)
print(ham)
```

- eggs becomes a new name for the same value [0] that ham is a name for ➔ [0,1]
- In Python a variable is not an alias for a location in memory. Rather, it is simply a binding to a Python object.

```
[0, 1]
```


- **Python**
  - **uses a mechanism, which is known as "Call-by-Object", sometimes also called "Call by Object Reference" or "Call by Sharing".**

① If you pass *immutable arguments* like *integers, strings or tuples* to a function, the passing acts like *call-by-value.*

⇔ **The object reference is passed to the function parameters.**

⇔ **They can't be changed within the function, because they can't be changed at all, *i.e.* they are immutable.**

② It's different, if we pass mutable arguments. They are also passed by object reference, but they *can be changed* in place in the function.

⇔ **If we pass a list to a function, we have to consider two cases:**

⇒ **Elements of a list can be changed in place, *i.e.* the list will be changed even in the caller's scope.**

⇒ **If a new list is assigned to the name, the old list will not be affected, *i.e.* the list in the caller's scope will remain untouched.**

Python initially behaves like call-by-reference, but as soon as we are changing the value of such a variable, Python "switches" to call-by-value.

```
def ref_demo(x):
    print "x=",x," id=",id(x)
    x=42
    print "x=",x," id=",id(x)
```

```
>>> x = 9
>>> id(x)
41902552
>>> ref_demo(x)
x= 9  id= 41902552
x= 42  id= 41903752
>>> id(x)
41902552
>>>
```

```
>>> def func1(list):
...     print list
...     list = [47,11]
...     print list
...
>>> fib = [0,1,1,2,3,5,8]
>>> func1(fib)
[0, 1, 1, 2, 3, 5, 8]
[47, 11]
>>> print fib
[0, 1, 1, 2, 3, 5, 8]
>>>
```

```
>>> def func2(list):
...     print list
...     list += [47,11]
...     print list
...
>>> fib = [0,1,1,2,3,5,8]
>>> func2(fib)
[0, 1, 1, 2, 3, 5, 8]
[0, 1, 1, 2, 3, 5, 8, 47, 11]
>>> print fib
[0, 1, 1, 2, 3, 5, 8, 47, 11]
>>>
```

– It is important to understand that *variables in Python are really just references to objects in memory*

– containers and user-defined types are generally *mutable* while everything else is *immutable.*

⇔ immutable objects

⇒ an immutable object (unchangeable object) is an object whose state cannot be modified after it is created

⇒ Numeric types (int, float, complex), string, *tuple*, frozen set, bytes

⇔ mutable objects : *list*, dict, set, byte array

```
a = 1
s = 'abc'
l = ['a string', 456, ('a', 'tuple', 'inside', 'a', 'list')]

/* create new objects, and the variable point to a
different object (newly created ones in our examples).  */
a = 7
s = 'xyz'
l = ['a simpler list', 99, 10]
```

only mutable objects can be changed in place (l[0] = 1 is ok in our example, but s[0] = 'a' raises an error).

```
def append_to_sequence (myseq):
    myseq += (9,9,9)
    return myseq

tuple1 = (1,2,3)      # tuples are immutable
list1  = [1,2,3]      # lists are mutable

tuple2 = append_to_sequence(tuple1)
list2  = append_to_sequence(list1)

print 'tuple1 = ', tuple1   # outputs (1, 2, 3)
print 'tuple2 = ', tuple2   # outputs (1, 2, 3, 9, 9, 9)
print 'list1  = ', list1    # outputs [1, 2, 3, 9, 9, 9]
print 'list2  = ', list2    # outputs [1, 2, 3, 9, 9, 9]
```

Real Generic Subprogram

– When used on an immutable object (as in a += 1 or in s += 'qwertz'), Python will silently create a new object and make the variable point to it.

– However, when used on a mutable object (as in l += [1,2,3]), the object pointed to by the variable will be changed in place.

– Command Line Arguments

```
# Module sys has to be imported:
import sys

# Iteration over all arguments:
for eachArg in sys.argv:
        print eachArg
```

argumente.py

```
>>> python argumente.py python course for beginners

argumente.py
python
course
For
beginners
```

– Variable Length Arguments

⇔ The asterisk "*" is used in Python to define a variable number of arguments. The asterisk character has to precede a variable identifier in the parameter list.

```
>>> def varpafu(*x): print(x)
...
>>> varpafu()
()
>>> varpafu(34,"Do you like Python?", "Of course")
(34, 'Do you like Python?', 'Of course')
>>>
```

```
def arithmetic_mean(x, *l):
    """ The function calculates the arithmetic mean of a
        non-empty arbitrary number of numbers """
    sum = x
    for i in l:
        sum += i

    return sum / (1.0 + len(l))
```

```
>>> from statistics import arithmetic_mean
>>> arithmetic_mean(4,7,9)
6.666666666666667
>>> arithmetic_mean(4,7,9,45,-3.7,99)
26.71666666666667
```

- Homework
  - ✓ Python built-in data types and parameter passing method (2 pages)

(4) Type-Checking Parameters

- It is now widely accepted that software reliability demands that *the types of actual parameters be checked for consistency with the corresponding formal parameters*
  - FORTRAN 77 : no parameter type checking
  - Pascal, Modula-2, FORTRAN 90 : parameter type checking
  - Original C : neither the number of parameters nor their types were check
  - ANSI C : the formal parameters of functions can be declared two ways
    - ⇔ the same way as original C : no type checking

```
double sin(x) double x ; { ..... }
double value ;
int count ;
.....
value = sin(count) ;
```

    - ⇔ prototype method : (*type checking* and *coercion*)
      - ⇒ coercion is used to match the types, or syntax error is reported

```
double sin(double x) ; { ..... }
…
value = sin(count) ;
```

  - C++
    - ⇔ the formal parameter list can have both typed parameters and ellipsis

```
printf(const char*, ...) ;
```

at least one parameter (a char pointer)

  - Relatively new languages Perl, JavaScript, and PHP do not require type checking. In Python and Ruby, variables do not have types (objects do), so parameter type checking is not possible

(5) Implementing Parameter-Passing Methods

- How are the primary implementation models of parameter passing actually implemented ?
  - In ALGOL 60 and its descendant languages, parameter communication takes place through the *run-time stack*
    - ⇔ *Pass-by-Value*
      - ⇒ Pass-by-value parameters have their values copied into stack location
    - ⇔ *Pass-by-Result*
      - ⇒ the values assigned to the pass-by-result actual parameters are placed in the stack, where they can be retrieved by the calling program unit upon termination of the called subprogram
    - ⇔ *Pass-by-Value-Result*
      - ⇒ a combination of pass-by-value and -result
    - ⇔ *Pass-by-Reference*
      - ⇒ regardless of the type of actual parameter, only *its address must be placed in the stack*
      - ⇒ in the case of an expression, the compiler must build code to evaluate the expression just before the transfer of control to the called subprogram. The address where that code place result of its expression is then placed in the stack (예 : `call sub(a*b)`)

```
main() {
 int a, b;
 sub1(a*b);
}
```

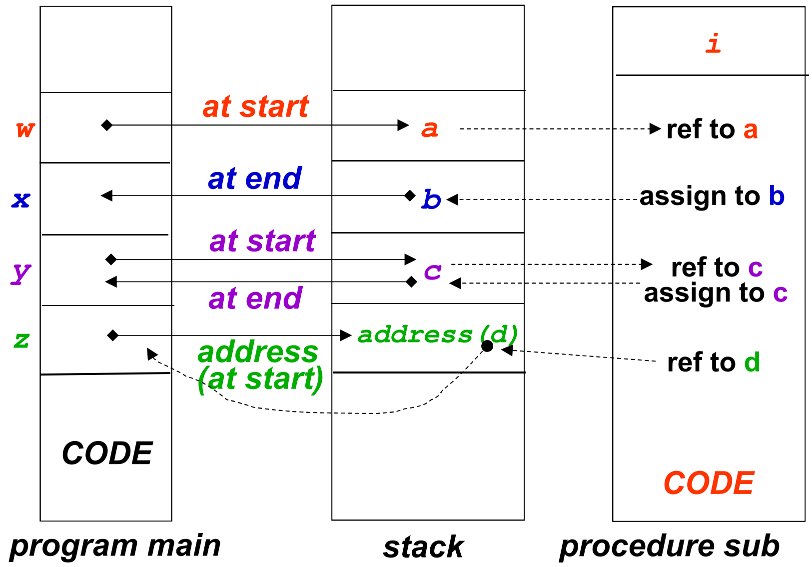
```
sub1(int x){
 …
 …
}
```

⇔ ***Pass-by-name*** **parameters are usually implemented with parameterless procedures or run-time-resident code segments, called *thunks* (costly process)**

- ⇒ **the thunks must be called for every reference to a pass-by-name parameter in the called subprogram**
- ⇒ **the thunk evaluates the reference in the proper referencing environment, which is that of subprogram in which the passed subprogram was declared, and returns the address of the actual parameter**

```
program main;
  var w,x,y,z:integer;
  procedure sub(a,b,c,d:integer);
    var i:integer;
    …
    end
  begin
    …
    call sub(w,x,y,z);
    …
  end
```

w : pass-by-value
x : pass-by-result
y: pass-by value-result
z : pass-by-reference



(6) Design Considerations

- **Design considerations**
  - **efficiency**
  - **one-way or two-way data transfer is desired**
    - ⇔ **SE principles dictate that access by subprogram code to data outside the subprogram be minimized (*in-mode only*)**

9.6 Parameters That are Subprogram Names

- **subprogram names are sent as parameters to other subprograms**

- **The description of the subprogram's parameters must be sent, along with the subprogram name (for *type checking*)**
  - **In ALGOL 68 and later version of Pascal**

```
procedure integrate (function fun (x : real) : real  ;
                     lowerbd, upperbd : real ;
                     var result : real ) ;
     var funval : real ;
     begin
       . . . . .
       funval := fun(lowerbd); … ; funval := fun(upperbd) ;
       . . . . .
      end ;
```

```
call integrate(sub1(), 5.0, 10.0) ;
```

```
sub1(x : real) { return(x*x);}
```

```
sub2(x : real) { return(2*x);}
```

- What is the correct referencing environment for executing the passed subprogram ?
  - *shallow binding* : the environment of subprogram that calls the passed subprograms (*SUB4*) (결과 : x=4)
    - ⇔ in dynamically scoped languages (ex. SNOBOL)
  - *deep binding* : the environment of subprogram in which the passed subprogram is declared (*SUB1*) (결과 : x=1)
    - ⇔ in block structured languages (ex. Pascal)
  - *others* : the environment of subprogram that includes the call statement that passed the subprogram as an actual parameter (*SUB3*) (결과 : x=3)

```
procedure SUB1 ;
   var x : integer ;
   procedure SUB2 ;
      begin
         write(`x=`, x);
      end ; {of SUB2}
   procedure SUB3 ;
      var x : integer ;
      begin
         x := 3 ; SUB4(SUB2) ;
      end ; {of SUB3}
   procedure SUB4(SUBX) ;
      var x : integer ;
      begin
         x := 4 ; SUBX;
   end ; {of SUB4}
begin {of SUB1}
    x := 1 ; SUB3;
end ; {of SUB1}
```

SUB1 -> SUB3 -> SUB4 -> SUB2

9.7 Overloaded Subprograms

- Overloaded subprogram
  - a subprogram that has the same name as another subprogram in the same referencing environment
  - every incarnation of an overloaded procedure must be unique in *the types of its parameters* and *return values*
  - the meaning of a call to an overloaded subprogram is *determined by the actual parameter list*
  - In Ada,
    - ⇔ allows both functions and procedures to be overloaded

```
procedure MAIN is
       type F_VECTOR is array (INTEGER range <>) of FLOAT;
       type I_VECTOR is array (INTEGER range <>) of INTEGER;
       ....
       procedure SORT(FLOAT_LIST : in out  F_VECTOR ;
                      LOWER_BOUND : in INTEGER ;
                      UPPER_BOUND : in INTEGER ) is
       …
       end SORT ;
       procedure SORT(INT_LIST : in out  I_VECTOR ;
                      LOWER_BOUND : in INTEGER ;
                      UPPER_BOUND : in INTEGER ) is
       …
       end SORT ;
       .....
    end MAIN ;
```

– C++ functions can be overloaded as long as *the number or types of parameters of each version are unique*

```
void fun(float b = 0.0) {
  …
}
void fun() {
  …
}
main() {
  …
  fun() ; /* ?? */
  …
}
```

– Ada, Java, C++, and C# allow users to write multiple versions of subprograms with the same name

9.8 Generic Subprograms

- In Ada,
  - provides a construction of a subprogram whose parameters can not only have different values, but also different types
  - the different versions of the subprogram are constructed by the compiler upon request of the user program
    - ⇔ generic unit is *nothing more than a template* for a procedure; no code is generated for it by compiler and it has no effect on program, unless it is instantiated for some type
    - ⇔ compiler builds a version of GENERIC_SORT named INTEGER_SORT that sorts INTEGER type variables

Not a real generic program !
(but a just macro expansion)

```
generic
   type ELEMENT is private ;
   type INDEX is (<>) ;
   type VECTOR is array (INDEX) of ELEMENT ;
   procedure GENERIC_SORT (LIST : in out VECTOR) is
      TEMP : ELEMENT ;
      begin
      for INDEX_1 in LIST's FIRST..INDE'PRED(LIST'LAST)  loop
        for INDEX_2 in INDEX'SUCC(INDEX_2) ..LIST'LAST  loop
           if LIST(INDEX_1) > LIST(INDEX_2) then
                     TEMP := LIST(INDEX_1);LIST(INDEX_1):=LIST(INDEX_2);
                     LIST(INDEX_2):=TEMP ;
           end if ;
          end loop
        end loop
   end GENERIC_SORT ;

procedure INTEGER_SORT is new GENERIC_SORT (
        ELEMENT=>INTEGER; INDEX=>INTEGER; VECTOR=>INT_LIST_TYPE) ;
```

9.9 Design Issues for Functions

- Two design issues specific to functions :
  - Are side effect allowed ?
  - What types of values can be returned ?
- Functional Side Effects
  - In Ada,
    - ⇔ because of the problems of side effects of functions that are called in expressions, parameters to function should always be ***in mode***
      - ⇒ effectively prevents a function from causing side effects through its parameters
  - In Pascal (and C)
    - ⇔ functions can have either pass-by-value or pass-by-reference parameters
      - ⇒ allowing functions that cause side effects
- Type of Returned Values
    - ⇔ Most imperative languages *restrict the types that can be returned* by their functions
  - In FORTRAN 77 : functions allow only unstructured types to be returned
  - In Pascal and Modular-2 : only simple types can be returned by function
      - ⇒ integer, real, char, Boolean, pointers, and enumeration types
  - In C : any type can be returned by its functions, excepts arrays and functions
  - Java and C# methods can return any type

9.10 Accessing Nonlocal Environments

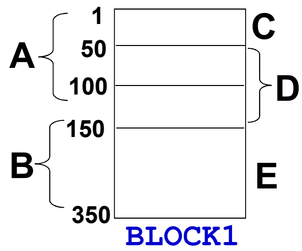
  - Although much of the required communication between subprograms can be accomplished *through parameters*, most languages provide some other method of accessing variables from external environments
  - Nonlocal variables of a subprogram are those that are visible within subprogram but are not locally declared
- In static scoping languages
  - more access to nonlocals is provided than is necessary
- In dynamic scoping languages,
  - all local variables of the subprogram are visible to any other executing subprogram regardless of its textual proximity
  - *an inability to statically type check references to nonlocals*

(1) FORTRAN common Blocks

- FORTRAN provides access to blocks of global storage through its COMMON
  - a common block is created when the first COMMON statement that mentions the block's name is found by the compiler
  - Problem : two subprogram can include the same data blocks with different names

```
SUB1 {
   REAL  A(100)
   INTEGER B(250)
   COMMON /BLOCK1/ A, B
}
```

```
SUB2 {
    REAL  C(50), D(100)
   INTEGER E(200)
   COMMON /BLOCK1/ C,D,E
}
```



(2) External Declarations and Modules

- Modular-2 and Ada
  - provide an alternative method of data sharing by allowing units to specify the external modules to which access is required
    - ⇔ every module can specify exactly the other modules to which access is needed, no more and no less ("with")
- C language (*no nesting of procedures*)
  - global variables can be created by placing their declarations outside function definition
  - Access is provided to a variable in a function that declares the variable to be external with an *extern* statement

9.11 User-Defined Overloaded Operators

- Operators can be overloaded by user in Ada and C++ program, if the types or number of parameters differ or the return types are different

Local Variables

```
function "*"(A,B : in MATRIX) return MATRIX is
  RESULT : MATRIX (A'FIRST(1) .. A'LAST(1)),
                   B'FIRST(2) .. B'LAST(2))
  SUM : integer ;
  begin
      for ROW in A'RANGE(1) loop
         for COL in B'RANGE(2) loop
             SUM := 0.0 ;
             for INNER in A'RANGE(2) loop
                 SUM := SUM + A(ROW, INNER) * B(INNER, COL) ;
              end loop
              RESULT(ROW, COL) := SUM ;
           end loop
         end loop
      end "*"

 . . . .
C := A * B
 . . . .
```

integer multiplication

User Defined Operator Overloading

```
*(A, B)  /* A와 B는 MATRIX type) */
```

function call

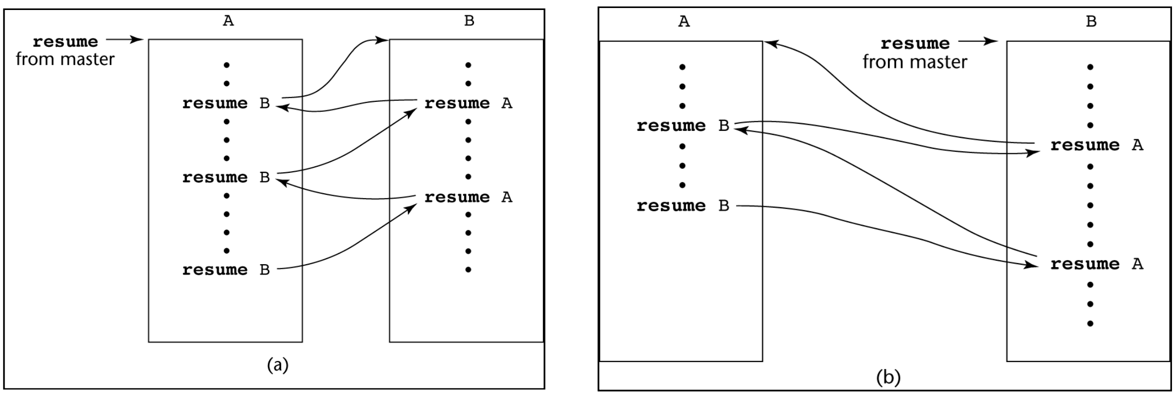
- **Closures**
  - A *closure* is a subprogram and the referencing environment where it was defined
  - A JavaScript closure
    - ⇔ The closure is *the anonymous function* returned by `makeAdder`

```
function makeAdder(x) {
     return function(y) {return x + y;}
} ➔ /* 함수의 return 값은 데이터가 아니라 함수이다. */
   ...
var add10 = makeAdder(10);
var add5 = makeAdder(5);
document.write("add 10 to 20: " + add10(20) + "<br />");
document.write("add 5 to 20: " + add5(20) + "<br />");
```

비슷한 일을 하는 함수를 여러 개 선언하지 않고
함수 하나를 선언하고 다른 부분만 parameter로 넘겨주어서
새로운 함수 (Closure)를 만들어서 사용한다.
(Macro 혹은 C++ Template 비교하여 장점은 ??)

9.12 Coroutines

- **Coroutines**
  - A coroutine is a subprogram that has multiple entries and controls them itself – supported directly in Lua
  - Also called *symmetric control*: caller and called coroutines are on a more equal basis
  - A coroutine call is named a resume
  - The first resume of a coroutine is to its beginning, but subsequent calls enter at the point just after the last executed statement in the coroutine
  - Coroutines repeatedly resume each other, possibly forever
  - Coroutines provide quasi(표면상)-concurrent execution of program units (the coroutines); their execution is interleaved, but not overlapped



6. 다음의 C-like한 프로그램에서 아래와 같은 5가지 parameter-passing 방법들에 의하여 swap()이 호출된다고 가정하였을 때, 이 프로그램의 수행이 끝난 뒤 (즉, 두 번의 swap() 호출이 모두 끝난 뒤) value 및 list[]에 저장된 내용은 각각 무엇인가 ? 단, by result 및 by copy인 경우에 actual parameter의 주소는 호출 전에 계산된다고 가정하고, 최종 값을 알 수 없는 경우는 "Unknown"으로 답할 것. (5점 x 5 = 25점)

```
void main() {
 int value=2, list[5]={1,3,5,7,9};
 swap(list[0], list[1]);
 swap(value, list[value]);
}

void swap(int a, int b) {
 int temp;
 temp = a; a = b ; b = temp;
}
```

(a) by value
(b) by result
(c) by copy
(d) by reference
(e) by name

	value	list[]
(a) by value	2	1, 3, 5, 7, 9
(b) by result	Unknown	U,U,U, 7, 9
(c) by copy	5	3, 1, 2, 7, 9
(d) by reference	5	3, 1, 2, 7, 9
(e) by name	5	3, 1, 5, 7, 9

1. Consider the following program written in C syntax:
For each of the following parameter-passing methods, what are all of the values of the variables value and list after each of the three calls to swap?
a. Passed by value
b. Passed by reference
c. Passed by value-result
d. python

```
void swap(int a, int b) {
        int temp;
        temp = a;
        a = b;
        b = temp;
}
void main() {
        int value = 2, list[5] = {1, 3, 5, 7, 9};
        swap(value, list[0]);
        swap(list[0], list[1]);
        swap(value, list[value]);
}
```

2. Consider the following program written in C syntax:
For each of the following parameter-passing methods, what are all of the values of the variables value and list after each of the three calls to swap?
a. Passed by value
b. Passed by reference
c. Passed by value-result
d. python

```
void fun (int first, int second) {
        first += first;
        second += second;
}
void main() {
        int list[2] = {1, 3};
        fun(list[0], list[1]);
}
```

과제 : Due to 5월 14일 (화요일)