Theory of Programming Languages

Week 8

SUBPROGRAMS

Fundamental Characteristics of Subprograms

1. **A subprogram has a single entry point**
2. **The caller is suspended during execution of the called subprogram**
3. **Control always returns to the caller when the called subprogram's execution terminates**

Basic definitions:

A *subprogram definition* is a description of the actions of the subprogram abstraction

A *subprogram call* is an explicit request that the subprogram be executed

A *subprogram header* is the first line of the definition, including the name, the kind of subprogram, and the formal parameters

The *parameter profile* of a subprogram is the number, order, and types of its parameters

The *protocol* of a subprogram is its parameter profile plus, if it is a function, its return type

A subprogram *declaration* provides the protocol, but not the body, of the subprogram

A *formal parameter* is a dummy variable listed in the subprogram header and used in the subprogram

An *actual parameter* represents a value or address used in the subprogram call statement

Actual/Formal Parameter Correspondence:

1. *Positional*

2. *Keyword*
e.g. `SORT(LIST => A, LENGTH => N);`

***Advantage*: order is irrelevant**
***Disadvantage*: user must know the formal parameter's names**

Default Values:

```
e.g. procedure SORT(LIST : LIST_TYPE;
                    LENGTH : INTEGER := 100);
     ...
     SORT(LIST => A);
```

Procedures provide user-defined statements

Functions provide user-defined operators

Design Issues for Subprograms

1. **What parameter passing methods are provided?**
2. **Are parameter types checked?**
3. **Are local variables static or dynamic?**
4. **What is the referencing environment of a passed subprogram?**
5. **Are parameter types in passed subprograms checked?**
6. **Can subprogram definitions be nested?**
7. **Can subprograms be overloaded?**
8. **Are subprograms allowed to be generic?**
9. **Is separate or independent compilation supported?**

Local referencing environments

If local variables are stack-dynamic:

- *Advantages:*
  a. Support for recursion
  b. Storage for locals is shared among some subprograms

- *Disadvantages:*
  a. Allocation/deallocation time
  b. Indirect addressing
  c. Subprograms cannot be history sensitive

Static locals are the opposite

Language Examples:

1. FORTRAN 77 and 90 - most are static, but can have either (SAVE forces static)

2. C - both (variables declared to be `static` are) (default is stack dynamic)

3. Pascal, Modula-2, and Ada - dynamic only

Parameters and Parameter Passing

Semantic Models: in mode, out mode, inout mode

Conceptual Models of Transfer:

1. Physically move a value

2. Move an access path

Implementation Models:

1. Pass-by-value (in mode)

- Either by physical move or access path

- Disadvantages of access path method:
  - Must write-protect in the called subprogram
  - Accesses cost more (indirect addressing)

- Disadvantages of physical move:
  - Requires more storage
  - Cost of the moves

2. Pass-by-result (out mode)

- Local's value is passed back to the caller

- Physical move is usually used
  - Disadvantages:
    a. If value is passed, time and space
    b. In both cases, order dependence may be a problem
    e.g.

```
procedure sub1(y: int, z: int);
  ...
sub1(x, x);
```

Value of x in the caller depends on order of assignments at the return

3. Pass-by-value-result (inout mode)

- Physical move, both ways

- Also called pass-by-copy

- Disadvantages:
  - Those of pass-by-result
  - Those of pass-by-value

4. Pass-by-reference (inout mode)

- Pass an access path
- Also called pass-by-sharing

- *Advantage*: passing process is efficient

- *Disadvantages*:
  a. Slower accesses
  b. Can allow aliasing:
    i. *Actual parameter collisions:*
    e.g.

```
procedure sub1(a: int, b: int);
  ...
  sub1(x, x);
```

ii. Array element collisions:
e.g.

```
sub1(a[i], a[j]);  /* if i = j  */
```

Also, `sub2(a, a[i]);`

iii. Collision between formals and globals

- Root cause of all of these is: The called subprogram is provided wider access to nonlocals than is necessary

- Pass-by-value-result does not allow these aliases (but has other problems!)

5. Pass-by-name

- By textual substitution

- Formals are bound to an access method at the time of the call, but actual binding to a value or address takes place at the time of a reference or assignment

- Purpose: flexibility of late binding

- ***Resulting semantics***:

- If actual is a scalar variable, it is pass-by-reference.
- If actual is a constant expression, it is pass-by-value
- If actual is an array element, it is pass-by-reference.

e.g.

```
procedure sub1(x: int; y: int);
  begin
   x := 1;
   y := 2;
   x := 2;
   y := 3;
  end;

 sub1(i, a[i]);
```

If actual is an expression with a reference to a variable that is also accessible in the program, a special case arises:

e.g. (assume k is a global variable)

```
procedure sub1(x: int; y: int; z: int);
  begin
    k := 1;
    y := x;
    k := 5;
    z := x;
  end;
  sub1(k+1, j, i);
```

- Disadvantages of pass by name:

- **Inefficient references (and late binding of values)**
- **Too tricky; hard to read and understand**

Implementing Parameter Passing

• **Pass by value:** copy it to the stack; references are direct to the stack (pass by result is the same).

• **Pass by reference:** regardless of form, put the address in the stack. References are indirect through the address on the stack.

•**Pass by Name:** run-time resident code segments or subprograms evaluate the address of the parameter; called for each reference to the formal; these are called ***thunks.***

Multidimensional Arrays as Parameters

-If a multidimensional array is passed to a subprogram and the subprogram is separately compiled, the compiler needs to know the declared size of that array for mapping function (e.g. to map 2D coordinates to 1D)

- C and C++

- Programmer is required to include the declared sizes of all but the first subscript in the actual parameter. This reduces flexibility.

void someFunc(int someArray[][3]) ;

Solution: pass a pointer to the array and the sizes of the dimensions as other parameters; the user must include the storage mapping function, which is in terms of the size parameters.

- Pascal: Not a problem (declared size is part of the array's type)

- Ada

Constrained arrays - like Pascal

Unconstrained arrays - declared size is part of the object declaration.

Pre-90 FORTRAN

- Formal parameter declarations for arrays can include passed parameters

```
SUBPROGRAM SUB(MATRIX, ROWS, COLS, RESULT)
  INTEGER ROWS, COLS
  REAL MATRIX (ROWS, COLS), RESULT
  ...
  END
```

Design Considerations for Parameter Passing

1. Efficiency
2. One-way or two-way

But these two are in conflict with one another!

- Good programming => limited access to variables, which means one-way whenever possible.
- Efficiency => pass by reference is fastest way to pass structures of significant size.

Parameters that are Subprogram Names

1. Are parameter types checked?

- Early Pascal and FORTRAN 77 do not
- Later versions of Pascal, Modula-2, and FORTRAN 90 do
- Ada does not allow subprogram parameters
- C and C++ - pass pointers to functions; type checking possible.

2. What is the correct referencing environment for a subprogram that was sent as a parameter?

- *Possibilities:*

a. It is that of the subprogram that enacted it.
  - *Shallow binding*

b. It is that of the subprogram that declared it.
  - *Deep binding*

c. It is that of the subprogram that passed it.
  - *Ad hoc binding* (Has never been used)

- For static-scoped languages, deep binding is most natural. Why?
- For dynamic-scoped languages, shallow binding is most natural.

An ***overloaded subprogram*** is one that has the same name as another subprogram in the same referencing environment

C++ and Ada have overloaded subprograms built-in, and users can write their own overloaded subprograms

These are differentiated based on their profiles.

Generic Subprograms

A ***generic*** **or** ***polymorphic*** **subprogram** is one that takes parameters of different types on different activations

Overloaded subprograms provide ***ad hoc polymorphism***

A subprogram that takes a generic parameter that is used in a type expression that describes the type of the parameters of the subprogram provides ***parametric polymorphism***

Example of a generic subprogram in Ada

```
generic
  type ELEMENT is private;
  type VECTOR  is array (INTEGER range <>) of ELEMENT;
  procedure GENERIC_SORT(LIST: in out VECTOR);

  procedure GENERIC_SORT(LIST: in out VECTOR) is
    TEMP : ELEMENT;
    begin
    for INDEX_1 in LIST'FIRST ..
        INDEX_1'PRED(LIST'LAST) loop
        for INDEX_2 in INDEX'SUCC(INDEX_1) ..
            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;  -- for INDEX_1 ...
      end loop;  -- for INDEX_2 ...
    end GENERIC_SORT;


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

C++

Templated functions are generic subprograms

```
template <class Type>
Type max(Type first, Type second) {
    return first > second ? first : second;
}
```

C++ template functions are instantiated ***implicitly*** **when the function is named in a call or when its address is taken with the & operator**

Another example:

```
template <class Type>
void generic_sort(Type list[], int len) {
  int top, bottom;
  Type temp;
  for (top = 0; top < len - 2; top++)
    for (bottom = top + 1; bottom < len - 1;
         bottom++) {
      if (list[top] > list[bottom]) {
        temp = list [top];
        list[top] = list[bottom];
        list[bottom] = temp;
       }  //** end of for (bottom = ...
 }  //** end of generic_sort
```

Example use:

```
float flt_list[100];
...
generic_sort(flt_list, 100); // Implicit
                             // instantiation
```

Independent and Separate Compilations

Independent compilation is compilation of some of the units of a program independently from the rest of the program, without the benefit of interface information (variable types and protocols of sub-programs).

Separate compilation is compilation of some of the units of a program separately from the rest of the program, using interface information to check the correctness of the interface between the two parts.

Reading assignment: Read about these in detail, and how various languages allow or do not allow them.

Accessing Nonlocal Environments

The **nonlocal variables** of a subprogram are those that are visible but not declared in the subprogram

Global variables are those that may be visible in all of the subprograms of a program

What are the various methods to access Nonlocal Environments?

Methods to access Nonlocal Environments

1. **FORTRAN COMMON**
   COMMON is a special statement that allows access to nonlocal variables stored in a special block.

   Following is one sub-program, adding to a specific block.

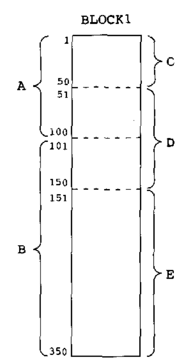
```
REAL A(100)
INTEGER B(250)
COMMON /BLOCK1/ A, B
```

following is another subprogram adding to the same block:

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

2. STATIC SCOPING – already discussed.

Two perspectives of the common block



User-Defined Overloaded Operators

Nearly all programming languages have *overloaded operators*

Users can further overload operators in C++ and Ada
(Not carried over into Java)

Example (Ada) (assume `VECTOR_TYPE` has been defined to be an array type with `INTEGER` elements):

```
function "*"(A, B : in VECTOR_TYPE)
    return INTEGER is
  SUM : INTEGER := 0;
  begin
  for INDEX in A'range loop
    SUM := SUM + A(INDEX) * B(INDEX);
  end loop;
  return SUM;
  end "*";
```

Are user-defined overloaded operators good or bad?

Co-Routines

Co-routines

A **coroutine** is a subprogram that has multiple entries and controls them itself (also called symmetric control)

- 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

- Typically, co-routines repeatedly resume each other, possibly forever.

- Co-routines provide quasi-concurrent execution of program units (the co-routines)

- Their execution is interleaved, but not overlapped

More on these later.