Chapter 9 Subprogram Units

- Introduction
- Fundamentals of Subprograms
- Design Issues for Subprograms
- Local Referencing Environments
- Parameter-Passing Methods
- Parameters that are Subprogram Names
- Overloaded Subprograms
- Generic Subprograms
- Design Issues for Functions
- User-Defined Overloaded Operators
- Coroutines

Introduction

- Two fundamental abstraction facilities
  - Process abstraction (this chapter) / control abstraction (Chapters 7-10)
  - Data abstraction (Chapters 5, 6 + 11)

- Process abstraction
  - Procedures provide user-defined statements
  - Functions provide user-defined operators

Fundamentals of Subprograms

- Actual/Formal Parameter Correspondence:
  1. Positional
  2. Keyword
    - e.g. in Ada: `SORT(LIST => A, LENGTH => 10);`
    - 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);`
Design Issues for Subprograms

1. What parameter passing methods are provided?
2. Are parameter types checked?
3. Are local variables static or dynamic?
4. Can subprogram definitions appear in other subprogram definitions?
5. What is the referencing environment of a passed subprogram?
6. Can subprograms be overloaded?
7. Are subprograms allowed to be generic?

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

Local referencing environments

- Language Examples:
  1. FORTRAN 77 and 90 - most are static, but the implementor can choose either (User can force static with `SAVE`)
  2. C - both (variables declared to be `static` are) (default is stack dynamic)
  3. Pascal, Java, and Ada - dynamic only
Parameter Passing Methods

- We discuss these at several different levels:
  - Semantic models
    - in mode, out mode, inout mode
  - Conceptual models
    - Physically move a value
    - Move an access path (pointer, reference)

Models of Parameter Passing

Parameter Passing Methods

- 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 (duplicated space)
       - Cost of the moves (if the parameter is large)
Parameter Passing Methods

2. Pass-by-result (out mode)
   - Local’s value is passed back to the caller
   - Physical move is usually used
   - Disadvantages:
     • If value is passed, time and space
     • In both cases, order dependence may be a problem
     e.g. 
     
     \[ \text{procedure sub1}(y: \text{int}, z: \text{int}); \]
     \[ \ldots \]
     \[ \text{sub1}(x, x); \]
   - Value of \( x \) in the caller depends on order of assignments at the return
   - Address evaluation: at the time of call or at the time of return?

Parameter Passing Methods

3. Pass-by-value-result (inout mode)
   - Physical move of value, both ways
   - Also called pass-by-copy
   - Disadvantages:
     • Those of pass-by-result
     • Those of pass-by-value

Parameter Passing Methods

4. Pass-by-reference (inout mode)
   - Pass an access path
   - Advantage: passing process is efficient (no copying and no duplicated storage)
   - Disadvantages:
     • Slower accesses
Parameter Passing Methods

• Pass-by-reference - disadvantages (cont)
  b. Allows aliasing:
    i. Actual parameter collisions:
      e.g. `procedure sub1(a: int, b: int);`  
          `...`  
          `sub1(x, x);`

Parameter Passing Methods

• Pass-by-reference - disadvantages (cont)
  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!)

Parameter Passing Methods

5. Pass-by-name (multiple mode)
   – 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
Parameter Passing Methods

5. Pass-by-name (multiple mode continued)

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 like nothing else
e.g.

```pascal
procedure sub1(x: int; y: int);
begin
  x := 1;
y := 2;
x := 2;
y := 3;
end;
sub1(i, a[i]);
```

---

Disadvantages of pass by name:
- Very inefficient references
- Too tricky; hard to read and understand

---

Parameter Passing Methods

5. Pass-by-name (multiple mode, continued)

- If actual is an expression with a reference to a variable that is also accessible in the program, it is also like nothing else
e.g. (assume `k` is a global variable)

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

---

Parameter Passing Methods

- Language Examples:
  1. FORTRAN
     - Before 77, pass-by-reference
     - 77 - scalar variables often passed by value-result
  2. ALGOL 60
     - Pass-by-name is default; pass-by-value is optional
  3. ALGOL W
     - Pass-by-value-result
Parameter Passing Methods

4. C
   - Pass-by-value
   - One-way pass by reference
     void fun(const int &p1, int &p2, ...)

5. Pascal and Modula-2
   - Default is pass-by-value; pass-by-reference is optional

6. C++
   - Like C, but also allows reference type parameters,
     int &ref_i;

7. Ada
   - All three semantic modes are available
   - If out, it cannot be referenced
   - If in, it cannot be assigned
     procedure adder(A: in out integer; B: in integer, ...)

8. Java
   - Like C++, but Java has no pointer, only references

• Type checking parameters (now considered very important for reliability)
  - FORTRAN 77 and original C: none
  - Pascal, FORTRAN 90, Java, and Ada: it is always required
  - ANSI C and C++: parameters in prototype form
  - Original C and C89: choice is made by the user
    Original C:
      double sin(x)
      double x;
      {....
    C99:
      double sin(double x)
Parameter Passing Methods

- Implementing Parameter Passing
  - ALGOL 60 and most of its descendants use the run-time stack
  - Pass-by-Value: copy to stack; references are indirect to stack
  - Pass-by-Result: same as value
  - Pass-by-Reference: put the address in 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
    - Very expensive

Parameter Passing Methods

- Ada
  - Simple variables are passed by copy (value-result)
  - Structured types can be either by copy or reference
  - This can be a problem, because
    - of aliases (reference allows aliases, but value-result does not)
    - procedure termination by error can produce different actual parameter results
      - Programs with such errors are “erroneous”
Parameter Passing Methods

- 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 to build the storage mapping function
- C and C++
  - Programmer is required to include the declared sizes of all but the first subscript in the actual parameter
    ```c
    void fun(int matrix[][10])
    {...
    }
    ```
  - This disallows writing flexible subprograms
  - 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 (Java is similar)
- Pre-90 FORTRAN
  - Formal parameter declarations for arrays can include passed parameters, e.g.
    ```fortran
    SUBPROGRAM SUB(MATRIX, ROWS, COLS, RESULT)
    INTEGER ROWS, COLS
    REAL MATRIX (ROWS, COLS), RESULT
    ...
    END
    ```

Parameter Passing Methods

- Design Considerations for Parameter Passing
  1. Efficiency
  2. One-way or two-way
     - 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
       - Also, functions should not allow reference parameters --- side effect
```c
void swap1(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}
swap1(c, d)
```

```c
void swap2(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
swap2(&c, &d)
```

In C++:
```cpp
void swap2(int &a, int &b) {
    int tmp = a;
    a = b;
    b = tmp;
}
swap2(c, d)
```

```c
int i = 3;
void fun(int a, int b) {
    i = b;
}
void main() {
    int list[10];
    list[i] = 5;
    fun(i, list[i]);
}
```

Pass-by-reference: 
- `i` is 5;

Pass-by-value-result: 
- `i` is set to 5, then copied back to 3.

### Parameters that are Subprogram Names

- **Issues: Are parameter types checked?**
  - Early Pascal and FORTRAN 77 do not
  - Later versions of Pascal and FORTRAN 90 do
  - Ada does not allow subprogram parameters
  - Java does not allow method names to be passed as parameters
  - C and C++ - pass pointers to functions; parameters can be type checked

### Parameters that are Subprogram Names

2. What is the correct referencing environment?
   - **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
   - For dynamic-scoped languages, shallow binding is most natural
Parameters that are Subprogram Names

Example:

```
sub1
  sub2
    sub3
      call sub4(sub2)
      call subx
    call subx
  call sub3
```

- What is the referencing environment of sub2 when it is called in sub4?
  - Shallow binding => sub2, sub4, sub3, sub1
  - Deep binding => sub2, sub1
  - Ad hoc binding => sub2, sub 3, sub1

Overloaded Subprograms

- Def: 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

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
Generic Subprograms

- Examples of parametric polymorphism
  
C++ Templated functions
e.g.

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

Generic Subprograms

```cpp
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;
            }
}
```

Example use:

```cpp
float flt_list[100];
...
generic_sort(flt_list, 100);
```

Separate or Independent Compilation

- Def: Independent compilation is compilation of some of the units of a program separately from the rest of the program, without the benefit of interface information.
- Def: 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.

Language Examples:
- FORTRAN II to FORTRAN 77 - independent
- FORTRAN 90, Ada, C++, Java - separate
- Pascal - allows neither
Design Issues for Functions

- Are side effects allowed?
  - Two-way parameters (Ada does not allow)
  - Nonlocal reference (all allow)
- What types of return values are allowed?
  - FORTRAN, Pascal - only simple types
  - C - any type except functions and arrays
  - Ada - any type (but subprograms are not types)
  - C++ and Java - like C, but also allow classes to be returned

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)

User-Defined Overloaded Operators

- 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?
Coroutines

- 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, coroutines repeatedly resume each other, possibly forever
- Coroutines provide quasi-concurrent execution of program units (the coroutines)
- Their execution is interleaved, but not overlapped