Statement Basics

• The meaning of a single statement executed in a state $s$ is a new state $s'$, which reflects the effects of the statement
  \[ M_{\text{stmt}}(\text{stmt}, s) = s' \]

Assignment Statements

\[ M_a(x := E, s) \triangleq s' = \{<i_1', v_1'>, <i_2', v_2'>, ..., <i_n', v_n'>\}, \]
where for $j = 1, 2, ..., n$,
\[ v_j' = \text{VARMAP}(i_j, s) \text{ if } i_j \neq x \]
\[ v_j' = M_e(E, s) \text{ if } i_j = x \]
Sequence of Statements

\[ M_{\text{stmt}}(\text{stmt1}; \text{stmt2}, s) \triangleq M_{\text{stmt}}(\text{stmt2}, M_{\text{stmt}}(\text{stmt1}, s)) \]

or

\[ M_{\text{stmt}}(\text{stmt1}; \text{stmt2}, s) = s'' \text{ where } \]

\[ s' = M_{\text{stmt}}(\text{stmt1}, s) \]

\[ s'' = M_{\text{stmt}}(\text{stmt2}, s') \]

\[ x := 5; \]
\[ y := x + 1; \]
\[ \text{write}(x \ast y); \]

Initial state \( s_0 = \langle \text{mem}_0, i_0, o_0 \rangle \)

\[ M_{\text{stmt}}(P_0, s_0) = M_{\text{stmt}}(P_1, M_a(x := 5, s_0)) \]

\[ s_1 = \langle \text{mem}_1, i_1, o_1 \rangle \text{ where } \]

\[ \text{VARMAP}(x, s_1) = 5 \]

\[ \text{VARMAP}(z, s_1) = \text{VARMAP}(z, s_0) \text{ for all } z \neq x \]

\[ i_1 = i_0, o_1 = o_0 \]
Sequence of Statements

```
x := 5;
y := x + 1;
write(x * y);
```

\[
M_{stmt}(P_1, s_1) = M_{stmt}(P_2, M_a(y := x + 1, s_1))
\]

\[
s_2 = <\text{mem}_2, i_2, o_2>, \text{ where}
\]

\[
\text{VARMAP}(y, s_2) = M_e(x + 1, s_1) = 6
\]

\[
\text{VARMAP}(z, s_2) = \text{VARMAP}(z, s_1) \text{ for all } z \neq y
\]

\[
i_2 = i_1, o_2 = o_1
\]
Sequence of Statements

Therefore,
\[ M_{stmt}(p, s_0) = s_3 = <mem_3, i_3, o_3> \] where
- \( \text{VARMAP}(y, s_3) = 6 \)
- \( \text{VARMAP}(x, s_3) = 5 \)
- \( \text{VARMAP}(z, s_3) = \text{VARMAP}(z, s_0) \) for all \( z \neq x, y \)
- \( i_3 = i_0 \)
- \( o_3 = o_0 \cdot 30 \)

Logical Pretest Loops

- The meaning of the loop is the value of program variables after the loop body has been executed the prescribed number of times, assuming there have been no errors.
- The loop is converted from iteration to recursion, where the recursion control is mathematically defined by other recursive state mapping functions.
- Recursion is easier to describe with mathematical rigor than iteration.
Logical Pretest Loop

• $M_l(\text{while } B \text{ do } L, s) \triangleq$
  if $M_b(B, s) = \text{false}$ then
    $s$
  else
    $M_l(\text{while } B \text{ do } L, M_{stmt}(L, s))$

Posttest Loop?

• $M_{ptl}(\text{do } L \text{ until not } B, s) \triangleq ?$
Key Points of Denotational Semantics

• Advantages
  – *Compact & precise*, with solid mathematical foundation
  – Provide a *rigorous* way to think about programs
  – Can be used to prove the correctness of programs
  – Can be an aid to language design

• Disadvantages
  – *Require mathematical sophistication*
  – Hard for programmer to use

• Uses
  – Semantics for Algol-60, Pascal, etc.
  – Compiler generation and optimization
Summary

- Each form of semantic description has its place
- Operational semantics
  - Informally describe the meaning of language constructs in terms of their effects on an ideal machine
- Denotational semantics
  - Formally define mathematical objects and functions to represent the meanings

Subroutine

In Text: Chapter 9
Outline [1]

- Definitions
- Design issues for subroutines
- Parameter passing modes and mechanisms
- Advanced subroutine issues

Subroutine

- A sequence of program instructions that perform a specific task, packaged as a unit
- The unit can be used in programs whenever the particular task should be performed
Subroutine

- Subroutines are the fundamental building blocks of programs
- They may be defined within programs, or separately in libraries that can be used by multiple programs
- In different programming languages, a subroutine may be called a \textit{procedure}, a \textit{routine}, a \textit{method}, or a \textit{subprogram}

Characteristics of Subroutines/Subprograms

- Each subroutine has a \textit{single entry point}
- The caller is suspended during the execution of the callee subroutine
- Control always returns to the caller when callee subroutine's execution terminates
Parameters

• A subroutine may be written to expect one or more data values from the calling program
• The expected values are called parameters or formal parameters
• The actual values provided by the calling program are called arguments or actual parameters

Actual/Formal Parameter Correspondence

• Two options
  – Positional parameters
    • In nearly all programming languages, the binding is done by position
    • E.g., the first actual parameter is bound to the first formal parameter
  – Keyword parameters
    • Each formal parameter and the corresponding actual parameter are specified together
    • E.g., Sort (List => A, Length => N)
Keyword Parameters

• Advantages
  – Order is irrelevant
  – When a parameter list is long, developers won’t make the mistake of wrongly ordered parameters

• Disadvantages
  – Users must know and specify the names of formal parameters

Default Parameter

• A parameter that has a default value provided to it
• If the user does not supply a value for this parameter, the default value will be used
• If the user does supply a value for the default parameter, the user-specified value is used
An Example in Ada

procedure sort (list   : List_Type;
    length : Integer := 100);
...
sort (list => A);

Design issues for subroutines

• What parameter passing methods are provided?
• Are parameter types checked?
• What is the referencing environment of a passed subroutine?
• Can subroutine definitions be nested?
• Can subroutines be overloaded?
• Are subroutines allowed to be generic?
• Is separate/independent compilation supported?
Parameter-Passing Methods

- Ways in which parameters are transmitted to and/or from callee subroutines
  - Semantic models
  - Implementation models

Semantic Models

- Formal parameters are characterized by one of three distinct semantic models
  - **In mode**: They can receive data from the corresponding actual parameters
  - **Out mode**: they can transmit data to the actual parameters
  - **Inout mode**: they can do both
Models of Parameter Passing

An Example

```java
public int[] merge(int[] arr1, int[] arr2) {
    int[] arr = new int[arr1.length + arr2.length];
    for (int i = 0; i < arr2.length; i++) {
        arr[i] = arr1[i];
        arr2[i] = arr1[i] + arr2[i];
        arr[i + arr1.length] = arr2[i];
    }
    return arr;
}
```

Which parameter is in mode, out mode, or inout mode?
**Implementation Models**

- A variety of models have been developed by language designers to guide the implementation of the three basic parameter transmission modes
  - Pass-by-value
  - Pass-by-result
  - Pass-by-value-result
  - Pass-by-reference
  - Pass-by-name

**Pass-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
- Implement in-mode semantics
- Implemented by copy
Pros and Cons

• Pros
  – Fast for scalars, in both linkage cost and access time
  – No side effects to the parameters

• Cons
  – Require extra storage for copying data
  – The storage and copy operations can be costly if the parameter is large, such as an array with many elements

Pass-by-Result

• No value is transmitted to a subroutine
• The corresponding formal parameter acts as a local variable, whose value is transmitted back to the caller’s actual parameter
  – E.g., void Fixer(out int x, out int y) {
    x = 17;
    y = 35;
  }
• Implement out-mode parameters
Pros and Cons

• Pros
  – Same as pass-by-value

• Cons
  – The same cons of pass-by-value
  – Parameter collision
    • E.g., Fixer(x, x), what will happen?
    • If the assignment statements inside Fixer() can be reordered, what will happen?

Pass-by-Value-Result

• A combination of pass-by-value and pass-by-result, also called pass-by-copy
• Implement inout-mode parameters
• Two steps
  – The value of the actual parameter is used to initialize the corresponding formal parameter
  – The formal parameter acts as a local variable, and at subroutine termination, its value is transmitted back to the actual parameter
Pros and Cons

• Pros
  – Same as pass-by-reference, which is to be discussed next

• Cons
  – Same as pass-by-result

Pass-by-Reference

• A second implementation model for inout-mode parameters
• Rather than copying data values back and forth, it shares an access path, usually an address, with the caller
  – E.g., void fun(int &first, int &second)
Pros and Cons

• Pros
  – Passing process is efficient in terms of time and space

• Cons
  – Access to the formal parameters is slower than pass-by-value parameters due to indirect access via reference
  – Side effects to parameters
  – Aliases can be created