9. Fundamentals of Subprograms

9.1. Introduction

- Two fundamental abstraction facilities
  - Process abstraction
    - Emphasized from early days
  - Data abstraction
    - Emphasized in the 1980s

9.2. Fundamentals of Subprograms

9.2.1. General Subprogram Characteristics

- Each subprogram has a single entry point
- The calling program is suspended during execution of the called subprogram
- Control always returns to the caller when the called subprogram’s execution terminates

9.2.2. Basic Definitions

- A subprogram definition describes the
  - interface to and the
  - actions of the subprogram abstraction
  - In Python, function definitions are executable; in all other languages, they are non-executable
- A subprogram call is an explicit request that the subprogram be executed
- A subprogram header is the first part of the definition, including
  - the kind of subprogram (a function or a procedure),
  - the name,
  - and, optionally, the formal parameters

Example:

```
def adder(parameters):
```

This the header of a Python subprogram named `adder`. Ruby subprogram headers also begin with `def`. The header of a JavaScript subprogram begins with `function`. In C, the header of a function named `adder` might be as follows:

```
void adder(parameters)
```

The reserve word `void` in this header indicates that the subprogram does not return a value.

- The body of a subprogram defines its actions.
  - In C-Based languages the body of a subprogram is delimited by braces.
  - In Ruby, an `end-statement` terminates the body of a subprogram.
  - As with compound statements, the statements in the body of a Python function must be indented and the end of the body is indicated by the first statement that is not indented.
• One characteristic of Python functions that sets them apart from functions of other programming languages is that function `def-statements` are executable. When a `def-statement` is executed, it assigns the given name to the function body. Until a function’s `def-statement` has been executed, the function cannot be called.

Example:
```python
if . . .
    def fun(. . .):
else
    def fun(. . .):
```

If the then clause of this selection construct is executed, that version of the function `fun` can be called, but not the version in the else clause. Likewise, if the else clause is chosen, its version of the function can be called but the on in the then clause cannot.

• Ruby methods are similar to functions in C and C++. Ruby methods are defined in class definitions by can also be defined outside class definitions, in which case they are considered methods of the root object, `Object`. Such methods can be called without an object receiver, as if they were functions in C or C++.

• All Lua functions are anonymous, although they can be defined using syntax that makes it appear as though they have names.

Example:
```lua
function cube(x) return x * x * x end
```

```lua
cube = function (x) return x * x * x end
```

The first of these uses conventional syntax, while the form of the second more accurately illustrates the namelessness of functions.

• The `parameter profile` of a subprogram is the number, order, and types of its parameters.
• The `protocol` is a subprogram’s parameter profile and, if it is a function, its return type.

• Function declarations in C and C++ are often called `prototypes`.

### 9.2.3. Parameters

• Subprograms describe computations. There are two ways that a subprogram that is not a method can gain access to the data that it is to process:
  1. through direct access to nonlocal variable that are declared elsewhere but visible in the subprogram
  2. through parameter passing

We prefer parameter passing because, through parameters, the computation is parameterized. Nonlocal variables must be altered to achieve the same effect and, although the computation performed on the nonlocal variables may be beneficial, the alteration of nonlocal variables may have a negative effect on other computation.
• A subprogram declaration provides the protocol, but not the body, of the subprogram.
• **Parameter:** A formal parameter is a dummy variable listed in the subprogram header and used in the subprogram.
• **Argument:** An actual parameter represents a value or address used in the subprogram call statement.

**Argument/Parameter Correspondence**

• Positional
  - The binding of actual parameters (arguments) to formal parameters is by position: the first actual parameter is bound to the first formal parameter and so forth
  - Safe and effective

• Keyword
  - The name of the formal parameter to which an actual parameter (argument) is to be bound is specified with the actual parameter
  - **Advantage:** Parameters can appear in any order, thereby avoiding parameter correspondence errors
  - **Disadvantage:** User must know the formal parameter’s names

**Actual/Formal Parameter Keyword Correspondence: Python Example**

```python
sumer(length = my_length, list = my_list, sum = my_sum)
```

Parameters: length, list, sum
Arguments: my_length, my_list, my_sum

Ada, Fortran 95+, and Python allow positional parameters to be mixed with keyword parameters as in the following example:

```python
sumer(my_length,
      sum = my_sum
      list = my_array)
```

**Formal Parameter Default Values**

• In certain languages (e.g., C++, Python, Ruby, Ada, PHP), formal parameters can have default values (if no actual parameter is passed)
  - Python Example
    ```python
def compute_pay(income, exemptions = 1, tax_rate)
    ```
    
    Note that no special syntax is required to call the function compute_pay even to the extent of including an additional comma to mark the absent argument.
    ```python
    pay = compute_pay(20000.0, tax_rate = 0.15)
    ```
• In C++, default parameters must appear last because parameters are positionally associated

Example

```c++
float compute_pay(float income, float tax_rate, int exemptions = 1)
```

and the corresponding call

```c++
pay = compute_pay(20000.0, 0.15);
```

• Variable numbers of parameters

  • In Ruby, the actual parameters are sent as elements of a hash literal and the corresponding formal parameter is preceded by an asterisk.
  • In Python, the actual is a list of values and the corresponding formal parameter is a name with an asterisk
  • In Lua, a variable number of parameters is represented as a formal parameter with three periods; they are accessed with a for-statement or with a multiple assignment from the three periods
  • C# methods can accept a variable number of parameters as long as they are of the same type—the corresponding formal parameter is an array preceded by `params`

C# Example

```csharp
public void DisplayList(params int[] list)
{
    foreach (int next in list)
    {
        Console.WriteLine("Next value {0}", next);
    }
}
```

If `DisplayList` is defined for the class `MyClass` as follows

```csharp
Myclass myobject = new MyClass;
int[] myList = new int[6] {2,4,6,8,10,12};

DisplayList could be called with either of the following:

```csharp
myobject.DisplayList(myList);
myobject.DisplayList(2,4,3*x-1,17);
```

Ruby supports a complicated but highly flexible actual parameter (argument) configuration. The initial parameters (arguments) are expressions, whose values objects are passed to corresponding formal parameters (parameters). The initial parameters can be followed by a list of key => value pairs, which are placed in an anonymous hash and a reference to that hash is passed to the next formal parameter. These are used as a substitute for keyword parameters, which Ruby does not support. The hash item can be followed by a single parameter preceded by an asterisk. This parameter is called the `array formal parameter`. When the method is called, the array formal parameter is set to
reference a new Array object. If the actual parameter that corresponds to the array formal parameter is an array, it must also be preceded by an asterisk, and it must be the last actual parameter. So, Ruby allows a variable number of parameters in a way similar to that of C#. Because Ruby arrays can store different types, there is no requirement that the actual parameters passed to the array have the same type.

Ruby Example

```ruby
list = [2,4,6,8]
def tester(p1, p2, p3, *p4)
    ...
end

tester('first', mon => 72, tue => 68, wed => 59, *list)
```

Inside tester, the values of its formal parameters are as follows

- `p1` is `'first'`
- `p2` is `{mon => 72, tue => 68, wed => 59}`
- `p3` is `2`
- `p4` is `[4,6,8]`

Lua uses a simple mechanism for supporting a variable number of parameters – such parameters are represented by an ellipsis ( . . . ). This ellipsis can be treated as an array or as a list of values that can be assigned to a list of variables.

Lua Example

```lua
function multiply(. . .)
    local product = 1
    for i, next in ipairs{. . .} do
        product = product * next
    end
    return product
end
```

`ipairs` is an iterator for arrays (it returns the index and value of the elements of an array one element at a time). `{ . . . }` is an array of the actual parameter values.

The three-period parameter need not be the only parameter – it can appear at the end of a list of named formal parameters.

9.2.4. Procedures and Functions

- There are two categories of subprograms
- *Procedures* are collection of statements that define parameterized computations
- *Functions* structurally resemble procedures but are semantically modeled on mathematical functions
- They are expected to produce no side effects
• In practice, program functions have side effects
• In Ada, Pascal, and Modula-2, procedures are called just that; in Fortran, they are called subroutines.
• Functions define new user-defined operators. For example, if a language does not have an exponentiation operator, a function can be written that returns the value of one of its parameters raised to the power of another parameter.

C++ example:

```cpp
def double power(double base, double exp)
```

The standard C++ library already includes a similar function `pow`.

Compare this with the same operation Perl, in which exponentiation is a built-in operation:

Perl example:

```perl
result = 3.4 * 10 ** x
```

Some programming languages allow users to overload operators by defining new functions for operators.

9.3. **Design Issues for Subprograms**

The nature of the local environment of a subprogram dictates the nature of the subprogram. Are local variables statically or dynamically allocated?

Can subprogram definitions be nested. Can subprogram names be passed as parameters. If a language allows nested subprograms and the passing of subprogram names, what is the referencing environment of the subprogram that was passed?

Can subprograms be overloaded. An overloaded subprogram is one that has the same name as another subprogram in the same referencing environment.

A closure is a nested subprogram and its referencing environment, which together allow the subprogram to be called from anywhere in a program.

• Are local variables static or dynamic?
• Can subprogram definitions appear in other subprogram definitions?
• What parameter passing methods are provided?
• Are parameter types checked?
• If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?
• Can subprograms be overloaded?
• Can subprogram be generic?
• If the language allows nested subprograms, are closures supported?
9.4. Local Referencing Environments

9.4.1. Local Variables

• Local variables can be stack-dynamic
  – Advantages
    • Support for recursion
    • Storage for locals is shared among some subprograms
  – Disadvantages
    • Allocation/de-allocation, initialization time
    • Cost of Indirect addressing
    • Subprograms cannot be history sensitive

• Local variables can be static
  – Advantages and disadvantages are the opposite of those for stack-dynamic local variables

In C and C++ function, locals are stack dynamic unless specifically declared to be static.

C++ example

```cpp
int adder(int list[], int listlen)
{
  static int sum = 0;
  for (int count=0; count < listlen; count++) sum += list[count];
  return sum;
}
```

9.4.2. Nested Subprograms

The idea of nesting subprograms originated with Algol 60. The motivation was to be able to create a hierarchy of both logic and scopes. If a subprogram is needed only within another subprogram, why not place it there and hide it from the rest of the program?

```plaintext
const
  Max = 100;

type
  List = Array[0..Max-1] of integer;

procedure sort(var A: List);
  var i, iom, eol: integer;
  procedure swap(var m, w: integer);
    var b: integer;
    begin{swap}
      b := m; m := w; w := b
    end{swap};
  begin{sort}
    for eol = Max downto 2 do
      begin
        for i := 1 to eol do
          begin{for}
          end{for};
          swap(A[eol], A[iom]);
        end{for}
    end{for}  
```
9.5. Parameter-Passing Methods

9.5.1. Semantic Models of Parameter Passing

- **In mode**
- **Out mode**
- **Inout mode**

![Diagram showing parameter passing](image)

**Figure 9.1 The three semantics models of parameter passing when physical moves are used**

9.5.2. Implementation Models of Parameter Passing

- **Conceptual Models of Transfer**
  - Physically move a path
  - Move an access path

9.5.2.1. Pass-by-Value (In Mode)

- The value of the actual parameter is used to initialize the corresponding formal parameter
- Normally implemented by copying
- Can be implemented by transmitting an access path but not recommended (enforcing write protection is not easy)
- *Disadvantages* (if by physical move): additional storage is required (stored twice) and the actual move can be costly (for large parameters - arrays)
- *Disadvantages* (if by access path method): must write-protect in the called subprogram and accesses cost more (indirect addressing)
9.5.2.2. Pass-by-Result (Out Mode)

- When a parameter is passed by result, no value is transmitted to the subprogram; the corresponding formal parameter acts as a local variable; its value is transmitted to caller’s actual parameter when control is returned to the caller, by physical move
  - Require extra storage location and copy operation
  - Potential problem: sub(p1, p1); whichever formal parameter is copied back will represent the current value of p1. The order in which values are copied back matters.

- Pass-by-Result: Example 1
  ```c
  void Fixer(out int x, out int y)
  {
    x = 17;
    y = 35;
  }
  ...
  f.Fixer(out a, out a);
  a = 17?
  a = 35?
  The order in which values are copied back to their actual parameters matters.
  ```

- Pass-by-Result: Example 2
  ```c
  void DoIt(out int x, int index)
  {
    x = 17;
    index = 42;
  }
  ...
  sub =21;
  f.DoIt(list[sub], sub);
  list[21]=17?
  list[42]=42?
  ```

The address of list[sub] changes between the beginning and end of the method. The implementor must choose the time to bind this parameter to an address – at the time of the call or at the time of the return. If the address is computed on entry to the method, the value 17 will be returned to list[21]; if computed just before returning, 17 will be returned to list[42].

9.5.2.3. Pass-by-Result (InOut Mode)

- A combination of pass-by-value and pass-by-result
- Sometimes called pass-by-copy
- Formal parameters have local storage
- Disadvantages:
  - Those of pass-by-result
  - Those of pass-by-value
- The actual parameter is copied to the formal parameter at subprogram entry and then copied back at subprogram termination.
9.5.2.4. Pass-by-Reference (Out Mode)

- Pass an access path (an address)
- Also called pass-by-sharing
- Advantage: Passing process is efficient (no copying and no duplicated storage)
- Disadvantages
  - Slower accesses (compared to pass-by-value) to formal parameters
  - Potentials for unwanted side effects (collisions)
  - Unwanted aliases (access broadened)

C++ example

```cpp
void fun(int& first, int& second)
```

If the call to `fun` happens to pass the same variable twice, as in

```cpp
fun(total, total)
```

then first and second will be aliases.

Collisions between array elements can also cause aliases.

C++ example

```cpp
fun1(list[i], list[j])
```

If `i == j` and variables are passed by reference then first and second are aliases.

```cpp
fun1(list[i], list)
```

The foregoing call to `fun1` could result in aliasing because all elements of the `list` can be accessed through the second parameter and access to a single element of the `list` is obtained through the first parameter.

9.5.2.5. Pass-by-Name (InOut 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
- Allows flexibility in late binding
9.5.3. Implementing Parameter-Passing Methods

- In most language parameter communication takes place through the run-time stack.
- Pass-by-value parameters have their values copied into stack locations. The stack locations then serve as storage for the corresponding formal parameters.
- Pass-by-value-result parameters are implemented as the opposite of pass-by-value. 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-reference is the simplest to implement; only an address is placed in the stack.

Figure 9.2 One possible stack implementation of the common parameter-passing methods

Function header: \texttt{void sub(int a, int b, int c, int d)}
Function call in \texttt{main: sub(w, x, y, z)}
(pass \texttt{w} by value, \texttt{x} by result, \texttt{y} by value-result, \texttt{z} by reference)

9.5.4. Parameter-Passing Passing Methods of 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
- Ada
  - Three semantics modes of parameter transmission: in, out, in out; in is the default mode
• Formal parameters declared out can be assigned but not referenced; those declared in can be referenced but not assigned; in out parameters can be referenced and assigned
• 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#
• Perl: all actual parameters are implicitly placed in a predefined array named @_
• Python and Ruby use pass-by-assignment (all data values are objects)

9.5.5. Type Checking Parameters

• 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++: choice is made by the user
• Prototypes
• 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

9.5.6. 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

9.5.6.1. Multidimensional Arrays as Parameters: C and C++

• Programmer is required to include the declared sizes of all but the first subscript in
  the actual parameter
• 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 in terms of the size
  parameters

9.5.6.2. Multidimensional Arrays as Parameters: Ada – not a problem

• Constrained arrays – size is part of the array’s type
• Unconstrained arrays - declared size is part of the object declaration

9.5.6.3. Multidimensional Arrays as Parameters: Fortran

• Formal parameter that are arrays have a declaration after the header
  • For single-dimension arrays, the subscript is irrelevant
For multidimensional arrays, the sizes are sent as parameters and used in the declaration of the formal parameter, so those variables are used in the storage mapping function.

9.5.6.4. Multidimensional Arrays as Parameters: Java and C#

- Similar to Ada
- Arrays are objects; they are all single-dimensioned, but the elements can be arrays
- Each array inherits a named constant (length in Java, Length in C#) that is set to the length of the array when the array object is created

9.5.7. Design Considerations

- Two important considerations
  - Efficiency
  - One-way or two-way data transfer
- But the above considerations are in conflict
  - Good programming suggest limited access to variables, which means one-way whenever possible
  - But pass-by-reference is far more efficient to pass structures of significant size

9.6. Parameters that are Subprograms

- It is sometimes convenient to pass subprogram names as parameters
- Issues:
  1. Are parameter types checked?
  2. What is the correct referencing environment for a subprogram that was sent as a parameter

9.6.1. Parameters that are Subprograms: Parameter Type Checking

- C and C++: functions cannot be passed as parameters but pointers to functions can be passed and their types include the types of the parameters, so parameters can be type checked
- FORTRAN 95 type checks
- Ada does not allow subprogram parameters; an alternative is provided via Ada’s generic facility
- Java does not allow method names to be passed as parameters
9.6.2. Parameters that are Subprograms: Referencing Environment

- **Shallow binding**: The environment of the call statement that enacts the passed subprogram
  - Most natural for dynamic-scoped languages
- **Deep binding**: The environment of the definition of the passed subprogram
  - Most natural for static-scoped languages
- **Ad hoc binding**: The environment of the call statement that passed the subprogram

```
function sub1()
{
    var x;
    function sub2()
    {
        alert(x); // Creates a dialog box for x
    };
    function sub3()
    {
        var x;
        x=3;
        sub4(sub2);
    };
    function sub4(subx)
    {
        var x;
        x=4;
        subx();
    };
    x=1;
    sub3();
}
```

9.7. Overloaded Subprograms

- An overloaded subprogram is one that has the same name as another subprogram in the same referencing environment
  - Every version of an overloaded subprogram has a unique protocol
- C++, Java, C#, and Ada include predefined overloaded subprograms
- In Ada, the return type of an overloaded function can be used to disambiguate calls (thus two overloaded functions can have the same parameters)
- Ada, Java, C++, and C# allow users to write multiple versions of subprograms with the same name
9.8. Generic Subprograms

- A generic or polymorphic subprogram 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
  - A cheap compile-time substitute for dynamic binding
- Ada
  - Versions of a generic subprogram are created by the compiler when explicitly instantiated by a declaration statement
  - Generic subprograms are preceded by a generic clause that lists the generic variables, which can be types or other subprograms
- C++
  - Versions of a generic subprogram are created implicitly when the subprogram is named in a call or when its address is taken with the & operator
  - Generic subprograms are preceded by a template clause that lists the generic variables, which can be type names or class names
- Java 5.0
  - Differences between generics in Java 5.0 and those of C++ and Ada:
    1. Generic parameters in Java 5.0 must be classes
    2. Java 5.0 generic methods are instantiated just once as truly generic methods
    3. Restrictions can be specified on the range of classes that can be passed to the generic method as generic parameters
    4. Wildcard types of generic parameters
- C# 2005
  - Supports generic methods that are similar to those of Java 5.0
  - One difference: actual type parameters in a call can be omitted if the compiler can infer the unspecified type
- Examples of parametric polymorphism: C++

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

- The above template can be instantiated for any type for which operator > is defined
```cpp
int max (int first, int second) {
    return first > second ? first : second;
}
```
9.9. Design Issues for Functions

• Are side effects allowed?
  – Parameters should always be in-mode to reduce side effect (like Ada)
• What types of return values are allowed?
  – Most imperative languages restrict the return types
  – C allows any type except arrays and functions
  – C++ is like C but also allows user-defined types
  – Ada subprograms can return any type (but Ada subprograms are not types, so they cannot be returned)
  – Java and C# methods can return any type (but because methods are not types, they cannot be returned)
  – Python and Ruby treat methods as first-class objects, so they can be returned, as well as any other class
  – Lua allows functions to return multiple values

9.10. User-Defined Overloaded Operators

• Operators can be overloaded in Ada, C++, Python, and Ruby
• An Ada example

```ada
function "*" (A, B: in Vec_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 "*";
...
c := a * b; -- a, b, and c are of type Vec_Type
```
9.11. 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-concurrent execution of program units (the coroutines); their execution is interleaved, but not overlapped

```lua
sub co1()
{
    ... resume co2();
    ... resume co3();
    ...
}
```

**Coroutines Illustrated: Possible Execution Controls**

(a)
Coroutines Illustrated: Possible Execution Controls with Loops

A

```
resume B
resume B
```

B

```
resume A
```

Resume from master
9.12. Summary

- A subprogram definition describes the actions represented by the subprogram
- Subprograms can be either functions or procedures
- Local variables in subprograms can be stack-dynamic or static
- Three models of parameter passing: in mode, out mode, and inout mode
- Some languages allow operator overloading
- Subprograms can be generic
- A coroutine is a special subprogram with multiple entries