Subprograms

Basic Characteristics
- Procedural abstraction
- Characteristics
  - Single entry & exit
  - Suspension of calling (sub)program
  - Return to caller on termination
- Definition - describes interface and procedure
- Call - Request for execution
- Active if called but not yet returned
- Header - that part of the definition that:
  - Declares as subprogram
  - Indicates kind
  - Specifies name
  - Lists parameters
- Body - the statements that define the procedure
- Definition (header) and declaration (body) can be separated
- Profile or signature - return type plus the types of the parameters

Parameters
- Subprogram accesses data via:
  - Nonlocal variables (referencing environment)
    - Reduces readability & reliability
  - Parameters
    - Data
    - Procedure
- Formal parameter (parameter) vs Actual parameter (argument)
- Positonal parameters vs named (keyword) parameters
  - Named useful with long parameter lists or when there are omitted parameters
  - Default arguments
    - w/o named parameters, defaults must occur last
  - Variable number of parameters
    - C#, Java, Ruby, Lua
```java
int num = length + my_method;
list = my_array;
num = my_num;
list = my_array;

get compute_pay(income, exemptions - 1, tax_rate)
pay = compute_pay(20000.0, tax_rate = 0.15)

public void display(int[] list) {
    foreach (int next in list) {
        Console.WriteLine("Next value {}", next);
    }
}
MyClass myObject = new MyClass;
int[] myslist = new int[6] { 2, 4, 6, 8, 10, 12 };  
Display list could be called with either of the following:
myObject.displayList(myslist);
myObject.displayList(2, 4, 3 * x - 1, 17);

list = { 2, 4, 6, 8 }
def test(p1, p2, p3, p4)
...
end

...  
#insert('first', mom = 72, son = 68, wed = 59, list)
inside test, the values of its formal parameters are as follows:
p1 is 'first'
p2 is { mom = 72, son = 68, wed = 59 }
p3 is 2
p4 is 4, 6, 8
```
**Kind**

- **Procedure**
  - Executed for side effect
  - Called as a statement
  - Extends language by adding statements
- **Function**
  - Executed for result
  - Called as an expression
  - Extends language by adding operations (operators)
- **Method**
  - In O-O languages
  - Essentially a procedure or function with a default parameter being the object executing the method body (`this`)
- Some languages have procedures & functions some have only functions

**Design Issues**

- Static or dynamic allocation for local variables
- Nesting of subprogram declarations
- Parameter passing methods
- Type checking of parameters
- Subprograms as parameters
  - Reference environment of passed subprogram
  - Closures?
- Overloading of subprograms
- Generic subprograms
Referencing Environment

- Storage allocation of local variables
  - Static
    - Less overhead
    - History property
    - Cannot support recursion
    - static in C
  - Stack-dynamic
    - Supports recursion
    - Share memory
    - Overhead
    - No history
    - Most languages
- Nested subprograms
  - Access to locals of encompassing subprogram
  - Often exposes too much
  - Better ways to control namespace
  - Modules/classes

Parameter Passing

- Semantic models
  - in parameters
  - out parameters
  - in-out parameters
- Implementation models
  - value, result, value-result, reference, name
Call-by-value

- Actual initializes formal
- Supports in parameters
- Typically implemented by copy
  - Value of actual copied to local storage for formal
- Advantages
  - Efficient for scalars
  - Protects actual
- Disadvantages
  - Expensive for large aggregates
  - Cannot return value via parameter

Call-by-result

- Actual set to value of formal at return
  - Typically actual must be a variable (i.e. lhs)
  - Supports out parameters
  - Typically implemented by copy
  - Value of formal copied to actual at return
- Advantages
  - Efficient for scalars
  - Value can be returned
- Disadvantages
  - Expensive for aggregates
  - Prevention of use of actual?
  - Parameter collision
  - When address of actual computed (e.g. array elements)
Call-by-value-result

- Combination of by-value and by-result
- Supports in-out parameters
- Typically implemented by copy
- Same advantages as by-value and by-result

Call-by-reference

- Formal is indirect reference to actual
  - Typically actual is an l value
- Supports in-out parameters
  - Can be used for in or out
- Typically a pointer is passed
  - Additional reference to actual
- Advantages
  - Efficient for aggregates
  - Can be used to support any model
- Disadvantages
  - Access is marginally slower
  - Creates alias if argument in scope
  - Parameter collisions
Call-by-name

- Actual passed unevaluated, evaluation at reference to formal
  - Like text substitution
- Supports in out parameters
- Typically implemented by replacing actual by a subroutine that computes access path (address) called when formal referenced
  - Subroutine must include referencing environment of actual (closure)
- Advantage
  - Flexibility
- Disadvantage
  - Overhead of computing access path
  - Complex hence readability & reliability problems

Parameter Passing – Implementation

- Run-time (Activation Record) stack
  - Subroutine linkage
  - Parameters
  - Local storage
- By-value
  - Argument copied to stack in storage for formal
- By-result
  - formal copied from stack to storage for argument
- By-value-result
  - Equivalent to combination of by-value & -result
- By-reference
  - Address of actual placed on sack, formal is indirect reference
Language Comparison

- **C**
  - By-value only
  - In-out via pointers
  - User implemented
  - Pass address of argument and formal is pointer
  - Formal as pointer to const
  - Formal as reference (1)
  - Address of argument passed, auto dereference of formal
- **C++**
  - Address of argument passed, auto dereference of formal
- **Java**
  - By-value only
  - Object & array variables are references so reference is passed
  - Cannot change to different object but can change state
- **C#**
  - Default by-value
  - ref for by-reference
  - out for out mode
- **Ada**
  - Specify mode, compiler chooses implementation
  - In mode makes formal a constant
  - Out mode requires an assignment to formal before its value can be referenced
- **Perl**
  - Parameters in an array
  - In-out mode
- **Python, Ruby**
  - Pass-by-assignment
  - Essentially reference passed (like objects in Java)
  - Immutable types don't get changed (since a new object is created)
Type Checking of Parameters

- Match type of actual to type of formal
- Mode
  - In – typically assignment compatible
  - In-out – typically equivalent types
  - Out – typically assignment compatible actual to formal
- C, C++
  - Original C – neither number nor type checked
  - C89 – introduced prototypes & type checking (user choice)
  - C99, C++ - requires prototype, no checking with ellipsis
Multi-dimensional Parameters

- Must be able to implement mapping function for formal without knowing attributes of actual
  - C
    - Mapping function
    - Parameter declaration includes all but first subscript
    - User-implemented via pointer arithmetic
  - Ada
    - Unconstrained array types
    - Essentially descriptor passed
  - Fortran
    - Array parameters must be dimensioned (typically via other parameters)
  - Java, C#
    - Array-of-array
    - Length attribute

```
address(mat[1..3]) = address(mat[0, 0]) + i

void fun(int matrix[10]) {
  ...
}
void main() {
  int mat[5][10];
  ...
  fun(mat);
  ...
}

void fun(float *mat_ptr,
         int num_rows,
         int num_cols);
*(mat_ptr + (row * num_cols) + col) = x;
```

```
type Mat_Type is array(integer range <>, integer range <=) of Float;
Mat_1 : Mat_Type(1..100, 1..20);

function Sumr(Mat : in Mat_Type) return Float is
begin
  Sum := 0.0;
  for Row in Mat.range(1) loop
    for Col in Mat.range(2) loop
      Sum := Sum + Mat(Row, Col);
    end loop;
  end loop;
  return Sum;
end Sumr;
```
Examples

- `swap`
  - C
    - By-value
      - Doesn’t work
    - By reference (pointers)
  - C++
    - By-reference (reference parameters)
  - Ada-style by-value-result
    - With parameter alias
  - By-value-result vs by-reference with non-local alias
```c
void swap1(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

```c
void swap1(int c, int d);
```

```c
a = c; — Move first parameter value in
b = d; — Move second parameter value in
temp = a
a = b
b = temp
```

```c
void swap2(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```c
void swap2(int *c, int *d);
```

```c
a = &c; — Move first parameter address in
b = &d; — Move second parameter address in
temp = *a
*a = *b
*b = temp
```
procedure swap1(a : in out Integer, b : in out Integer) is
  begin
  temp := a;
  a := b;
  b := temp;
  end swap1;

  swap2(c, d);
  addr_c := &c
  addr_d := &d
  a := *addr_c
  b := *addr_d
  temp := a
  a := b
  b := temp
  *addr_c := a
  *addr_d := b

procedure swap2(a : in out Integer, b : in out Integer) is
  begin
  temp := a;
  a := b;
  b := temp;
  end swap2;

  swap3, list[1]);
  addr_i := &i
  addr_list := &list(1)
  a := *addr_i
  b := *addr_list
  temp := a
  a := b
  b := temp
  *addr_i := a
  *addr_list := b

list i := 3; /* i is a global variable */
void fun1(a, list b) {
  i := b;
}
void main() {
  list i := [10];
  list[1] := 5;
  fun1, list[1]);

  addr_i := &i
  addr_list := &list(1)
  a := *addr_i
  b := *addr_list
  i := b
  *addr_i := a
  *addr_list := b

  void fun2(a, list b) {
  i := b;
}
void main() {
  list i := [10];
  list[1] := 5;
  fun2, list[1]);
Subprograms as Parameters

- E.g. integration
  - Sample function at various points
- Issues
  - Type checking of parameters of passed subprogram
    - Prototype included in parameter list
  - Referencing environment
    - Only with nested subprograms
    - Shallow binding vs deep binding vs ad-hoc binding
    - Static scoping vs deep-binding
    - Dynamic scoping – sometimes shallow binding

Calling Subprograms via Indirection

- Dynamic choice of subprogram to call
  - Jump tables, event handling, callbacks
- Pointer to function variable
  - Assigned a value before call
- C++
  - Declaration of pointer includes prototype
  - Function name is pointer to function
- C#
  - Delegates
    - References to functions are objects
    - Instantiation provides actual function
  - Multicast delegate
    - Multiple functions
    - All are called
Overloaded Subprograms

- Multiple subprograms with same name in single referencing environment
- Constructors
- Disambiguation
  - Different protocols
- Coercion
  - Complicates disambiguation
    - Multiple coercions possible?
    - Return type
- Default parameters and variable number of parameters
  - Complicate disambiguation
Generic Subprograms

- Subprograms that differ only in the type of data
  - E.g., sort
- Polymorphic subprogram
  - Subprogram that takes parameters of different types
    - Ad hoc polymorphism
      - Overloading but algorithms may differ
    - Subtype polymorphism
      - Actual parameter may be a subtype of formal
    - Parametric polymorphism
      - Types as "parameters"
      - Algorithm is same
      - aka generic subprograms

Generic functions in C++
- Template functions
  - Lists type "parameters"
  - Used as type names in function
  - Instantiated for any type
    - Implicit instantiation
    - Elaborated for each instantiation
- Generic methods in Java
  - Generic parameters must be classes
  - Instantiation doesn’t create a new subprogram
  - Supports bounded generic parameters
  - Supports wildcards and bounded wildcards
- Generic methods in C#
  - Similar to Java
  - No wildcards
  - Implicit actual type parameters
- Generic functions in F#
  - Automatic generalization when cannot infer an unique type

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

```c
int a, b, c;
char d, e, f;
int x = max(a, b);
f = max(d, e);
```

```c
int max(int first, int second) {
    return first > second ? first : second;
}
```
```java
// template <class Type>
// void generic_sort(Type list[], int len) {
//   int top, bottom;
//   TYPE temp;
//   for (top = 0; top < len; top++)
//     for (bottom = top + 1; bottom < len; bottom++)
//       if (list[top] > list[bottom]) {
//         temp = list[top];
//         list[top] = list[bottom];
//         list[bottom] = temp;
//       } //*** end of generic_sort

float fit_list[100];
...
//generic_sort(fit_list, 100);

public static <T> T doIt(T[] list) {
  ...
}
doitStrings(myList);

public static <T extends Comparable> T doIt(T[] list) {
  ...
}
void printCollection(Collection<? super c> c) {
  for (Object c; c) {
    System.out.println(c);
  }
}
public void drawAll(List<Shape> things)

class MyClass {
  public static T doIt<T>() {
    ...
  }

  String myInt = MyClass.doIt<int>(); // Calls doIt<int>
  String myStr = MyClass.doIt<string>(); // Calls doIt<string>
```
Design Issues for Functions

- Restriction of side effects
  - Parameters restricted to in mode e.g. Ada
  - Subprograms not nested
- Types for return values
  - C
    - Any type but array or function
    - Can return pointer
  - C++
    - Like C but also user-defined types (e.g. class)
  - Functions as first-class objects
    - Can be passed as parameters & returned from functions
  - Python, Ruby, Lua
  - Java, C#
    - Methods are not first-class objects
- Number of return values
  - Usually one
  - Ruby
    - return has 0 expressions, result is nil
    - return has 1 expression, result is single value
    - return has >1 expression, result is array
  - Lua
    - Comma separated values in return
    - Call statement determines how many values used
  - F#
    - Can return a tuple
Overloading Operators

- Additional operations for existing operators
  - Ada, C++, Python, Ruby
- Typically mapping between operator and function
  - Must match protocol for existing operator
- E.g. Python support for complex numbers
  - \(x + y\) mapped to method call \(x._add_(y)\)
  - Self reference not implicit in Python
- C++, Ada
  - Special notation for operator as function

def _add_(self, second):
    return Complex(self.real + second.real, self.imag + second.imag)

Complex operator + (Complex second) {
    return Complex(real + second.real, imag + second.imag);
}
Closures

- Closure - a subprogram and the reference environment where it is defined
- Can be required if the subprogram is executed from an arbitrary place in the code
  - Subprogram parameters
  - Subprogram pointers or variables
- Not really needed if static scoping and no nesting of subprograms
  - Subprogram parameters or variables
- Necessary if nested subprogram could access variable from encompassing subroutines that are not visible or active at the point of call
- Variables in the closure may need lifetimes greater than the usual block-entry to exit (i.e., become static or unlimited extent)
- E.g., JavaScript, C#
Coroutines

- Coroutines are special subprograms that operate co-operatively rather than master/slave
  - No differentiation between caller and callee
  - Transfer control between them
- Multiple entry points
  - \texttt{resume \_r();}  
    "Suspend execution of this coroutine and continue execution of \_r where it left off"
  - Interleaved execution model (quasi-concurrent)
- Only one coroutine executing at any time
- Typical execution model
  - A non-coroutine (master) creates coroutines
  - Each executes initialization code & returns to master
  - Master resumes one
  - Coroutines resume each other until originally resumed one terminates (returns to master)
  - Master continues

```c
void main()
{
    sub \texttt{o1}()
    {
        \ldots
        resume \texttt{o2}();
        \ldots
        resume \texttt{o3}();
        \ldots
    }
}
```

![Diagram of coroutine execution](https://example.com/diagram.png)

© 2012, R. Sebesta