Subprograms are the fundamental building blocks of programs and therefore among the most important concepts in programming language design.

1. **Fundamentals of Subprograms**
   - General Subprogram Characteristics:
     1. Each subprogram has a single entry point.
     2. The calling program unit is suspended during the execution of the called subprogram, which implies that there is only one subprogram in execution at any given time.
     3. Control always returns to the caller when the subprogram execution terminates.
   - Basic Definitions:
     1. Subprogram definition: describes the interface to and the subprogram abstraction.
     2. Subprogram call: explicit request that a specific subprogram be executed.
     3. Subprogram header: first part of the definition.
     5. Parameter profile: contains the number, order, and type of its formal parameters.
     6. Protocol: parameter profile implies, if it is a function, its return type.
     7. Prototype: function declaration.
   - Parameters:
     1. Formal Parameters: parameters in the subprogram header.
     2. Actual Parameters: bound to the formal parameters.
     3. Positional Parameters: first actual parameter is bound to the first formal parameter and so forth.
     4. Keyword Parameters: name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter in a call.
     5. Array formal Parameters: the hash item can be followed by a single parameter preceded by an asterisk.
Procedure and Functions - (Categories of sub-program)

1. Function can return value but procedure do not.
2. Function can be used as procedures but vice versa is not possible.
3. Procedure can produce results in the calling program unit by:
   i. Procedure can change variables of calling program unit that are visible
   ii. Through formal parameters

2. Scope and lifetime of a variable -
   Scope of a variable is from its declaration to the end of the method but the lifetime of a variable is the period of time beginning when the method is entered and ending when execution of the method terminates.

3. Static scope and dynamic scope -
   Static scope is named because the scope of a variable can be statically determined, that is prior to execution.
   Dynamic scope is based on the calling sequence of sub-programs, not their spatial relationship to each other. Thus, this scope can be determined only at run time.

```plaintext
function big() {
  function sub1() {
    var n = 7; //global
    sub2();
  }
  function sub2() {
    var y = n;
    var z = 3;
  }
  return 0;
}
```

```plaintext
function big() {
  function sub1() {
    var n = 7;
    sub2();
  }
  function sub2() {
    var y = n;
    var z = 3;
  }
  return 0;
}
```
Design Issues for Subroutines

1. Are local variables statically or dynamically allocated?
2. Can subroutine definitions appear in other subroutine definitions?
3. What parameter-passing method or methods are used?
4. Are the types of actual parameters checked against the types of the formal parameters?
5. If subroutines can be passed as parameters and subroutines can be nested, what is the referencing environment of a nested subroutine?
6. Can subroutines be overloaded?
7. Can subroutines be generic?
8. If the language allows nested subroutines, are closures supported?

Local Referencing Environment

- Local Variables

Variables that are defined inside subroutines. Local variables can be static or stack, dynamic (they are bound to storage when the subroutine begins execution and are unbound from storage when that execution terminates).

```c
int colder (int list[2], int listener)
{
    static int sum = 0;  // static local variable
    int count;  // start dynamic variable: local variable
    for (count = 0; count < listener; count++)
        sum += list[count];
    return sum;
}
```

- Nested Subroutines

Subroutine within another subroutine.
Parameter Passing Methods -

-**Semantic models of parameter passing** - These models aim to:
  1. **In mode** - They can receive data from the corresponding actual parameter.
  2. **Out mode** - They can transmit data to the actual parameter.
  3. **Input mode** - They can do both.

-**Implementation model of parameter passing** -
  1. **Pass-by-value** - The value of the actual parameter is used to initialize the corresponding formal parameter. (in mode semantic)
  2. **Pass-by-result** - No value is transmitted to the subprogram and it returns a result or variable. (out mode semantic)

Problem 1: void Finer (out int x, out int y)

```plaintext
x = 17;
y = 35;
```

It is the value of a
but it is assigned to 37.

Problem 2: void Finer (out int a, out int b);

- **Pass-by-name** - When parameters are passed by name, the actual parameter is in the subprogram (input mode semantic)

---

### Table: Implementing Parameter Passing Methods

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Function sub</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>At Init</td>
<td>Value of a</td>
</tr>
<tr>
<td>x</td>
<td>At End</td>
<td>Value of b</td>
</tr>
<tr>
<td>y</td>
<td>At Init</td>
<td>Value of c</td>
</tr>
<tr>
<td>z</td>
<td>Code</td>
<td>Address (b)</td>
</tr>
</tbody>
</table>

- **Pass w by value**
- **Pass x by result**
- **Pass y by value-result**
- **Pass z by reference**

Function header: void sub (int a, int b, int c, int d)

Function call in main: sub (w, x, y, z)
7. Overloaded subprograms -
   It is a subprogram that has the same name as another subprogram in the same referencing environment.
   ```
   void func (float b = 0.0);
   void func ();
   ```
   ```
   fun ();// The call is ambiguous & will cause a compilation error.
   ```
8. Generic subprograms -
   Parameterized polymorphism: subprograms are often called generic subprograms that mean it takes generic parameters that are used in type inference that derive the types of the parameters of the subprogram.
   ```
   template <template parameters>
   ```
   ```
   template <class Type>
   ```
   ```
   Type main (Type first, Type second) {
   return first > second ? first : second;
   }
   ```
   ```
   generic methods in Java 5.0 - generic class <T>
   ```
   ```
   public static <T> T doIt (T[] list) {
   ```
   call of to doIt -> doIt <String> (new list);
   ```
   ```
   generic methods in (#2005-
   ```
   class MyClass {
   ```
   ```
   public static <T> doIt <T> (T[] list) {
   ```
   `7` `7` `7`
   ```
   ```
   generic functions in F#-
   ```
   ```
   let getLast (a, b, c) = c;
   ```
   ```
   Design Issues for function -
   The following design issues are specific to functions -
   1. Are side effects allowed?
   2. What types of values can be returned?
   3. How many values can be returned?
(10) Overloaded operator:

\[ \text{class Complex} \]

\[
\begin{array}{l}
\text{operator + (Complex & second) }
\end{array}
\]

return Complex (real + second.real, imag + second.imag);

(11) Co-routines — Special kind of sub-program.

It can have multiple entry points, which are controlled by the co-routines themselves. Co-routines must be history sensitive and thus have static local variables. Co-routines often begin at points other than their beginning. Because of this, the invocation of co-routine is called resume rather than call.

Two possible execution control sequence for two co-routines without lock:

Co-routine execution sequence with lock: