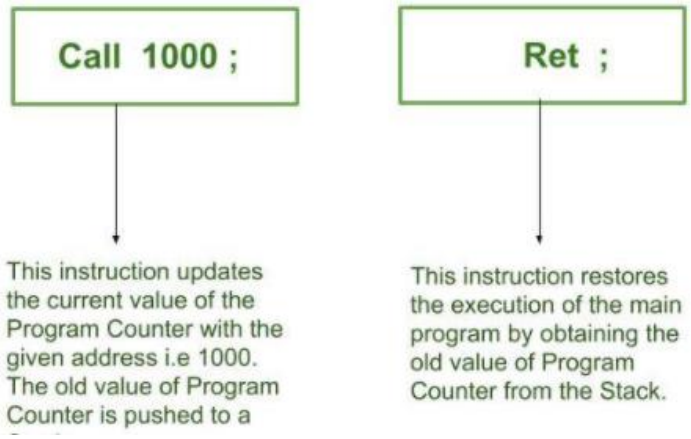
SUB PROGRAMS

A Subprogram is a program inside any larger program that can be reused any number of times.

Characteristics of a Subprogram:

(1) A Subprogram is implemented using the Call & Return instructions in Assembly Language.

(2) The Call Instruction is present in the Main Program and the Return(Ret) Instruction is present in the subprogram itself.Basic Definitions



(3) It is important to note that the Main Program is suspended during the execution of any subprogram. Moreover, after the completion of the subprogram, the main program executes from the next sequential address present in the Program Counter.

(4) For the implementation of any subprogram, a “Stack” is used to store the “Return Address” to the Main Program . Here, Return Address means the immediately next instruction address after the Call Instruction in the Main program.

This Return Address is present inside the Program Counter. Thus during the execution of the Call Instruction, the Program Counter value is first pushed to the Stack as the Return Address and then the Program Counter value is updated to the given address in the Call Instruction. Similarly, during the execution of Return(Ret) Instruction, the value present in the stack is popped and the Program Counter value is restored for further execution of the Main Program.

(5) The Main advantage of Subprogram is that it avoids repetition of Code and allows us to reuse the same code again and again.

DEFINITION

• 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 name, the kind of subprogram, and the formal parameters

• The parameter profile (aka signature) 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 Fundamentals of Subprograms

• 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

Function declarations in C and C++ are often called prototypes

• A subprogram declaration provides the protocol, but not the body, of the subprogram

• A formal parameter is a dummy variable listed in the subprogram header and used in the subprogram

• An actual parameter represents a value or address used in the subprogram call statement

Actual/Formal Parameter Correspondence

Positional

– The binding of actual parameters 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 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

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)

– In C++, default parameters must appear last because parameters are positionally associated

Variable numbers of parameters

– 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

– 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

Ruby Blocks

Ruby includes a number of iterator functions, which are often used to process the elements of arrays

• Iterators are implemented with blocks, which can also be defined by applications

• Blocks are attached methods calls; they can have parameters (in vertical bars); they are executed when the method executes a yield statement

```
def fibonacci(last)
    first, second = 1, 1
    while first <= last
        yield first
        first, second = second, first + second
    end
end
puts "Fibonacci numbers less than 100 are:"
fibonacci(100) {|num| print num, " "}
puts
```

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

DESIGN ISSUES FOR SUBPROGRAMS

Following are the Design Issues for Subprograms:

1. Are local variables statically or dynamically allocated?

2. Can subprogram definitions appear in other subprogram definitions?

3. What parameter-passing method or methods are used?

4. Are the types of the actual parameters checked against the types of the formal parameters?

5. If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?

6. Can subprograms be overloaded?

7. Can subprograms be generic?

8. If the language allows nested subprograms, are closures supported?

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

LOCAL REFERENCING ENVIRONMENTS

Local Variables

Subprograms can define their own variables, thereby defining local referencing environments. Variables that are defined inside subprograms are called local variables, because their scope is usually the body of the subprogram in which they are defined.

Local variables can be either static or stack dynamic. If local variables are stack dynamic, they are bound to storage when the subprogram begins execution and are unbound from storage when that execution terminates. There are several advantages of stack-dynamic local variables, the

primary one being the flexibility they provide to the subprogram. It is essential that recursive subprograms have stack-dynamic local variables. Another advantage of stack-dynamic locals is that the storage for local variables in an active subprogram can be shared with the local variables in all inactive subprograms. This is not as great an advantage as it was when computers had smaller memories.

The main disadvantages of stack-dynamic local variables are the following:

First, there is the cost of the time required to allocate, initialize (when necessary), and deallocate such variables for each call to the subprogram. Second, accesses to stack-dynamic local variables must be indirect, whereas accesses to static variables can be direct.4

This indirectness is required because the place in the stack where a particular local variable will reside can be determined only during execution.

Finally, when all local variables are stack dynamic, subprograms cannot be history sensitive; that is, they cannot retain data values of local variables between calls. It is sometimes convenient to be able to write history-sensitive subprograms.

A common example of a need for a history-sensitive subprogram is one whose task is to generate pseudorandom numbers. Each call to such a subprogram computes one pseudorandom number, using the last one it computed. It must, therefore, store the last one in a static local variable. Coroutines and the subprograms used in iterator loop constructs are other examples of subprograms that need to be history sensitive. The primary advantage of static local variables over stack-dynamic local variables is that they are slightly more efficient—they require no run-time overhead for allocation and deallocation.

Also, if accessed directly, these accesses are obviously more efficient. And, of course, they allow subprograms to be history sensitive. The greatest disadvantage of static local variables is their inability to support recursion. Also, their storage cannot be shared with the local variables of other inactive subprograms.

In most contemporary languages, local variables in a subprogram are by default stack dynamic. In C and C functions, locals are stack dynamic unless specifically declared to be static.

For example, in the following C (or C) function, the variable sum is static and count is stack dynamic.

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

The methods of C , Java, and C# have only stack-dynamic local variables. In Python, the only declarations used in method definitions are for globals. Any variable declared to be global in a method must be a variable defined outside the method. A variable defined outside the method can be referenced in the method without declaring it to be global, but such a variable cannot be assigned in the method. If the name of a global variable is assigned in a method, it is implicitly declared to be a local and the assignment does not disturb the global. All local variables in Python methods are stack dynamic.

Only variables with restricted scope are declared in Lua. Any block, including the body of a function, can declare local variables with the local declaration, as in the following:

```
local sum
```

All nondeclared variables in Lua are global. Access to local variables in Lua are faster than access to global variables according to Ierusalimschy (2006).