Procedural Programming Model

Loic Niragire
TIM8110 – Programming Languages and Algorithms
March 1st, 2021

ABSTRACT
We examine the procedural programming model and its usages in this article. Moreover, we explore concepts of input parsing and lexical analyzers relative to procedural programming. Additionally, we provide illustrations comparing various basic constructs in C, Fortran, and Lisp. Finally, we conclude the article by identifying applications and solutions best suited for the procedural programming model.

1. INTRODUCTION

This article defines a programming model as a set of programming techniques and design principles made possible by a computational model. By computational model, in this case, we mean a system that defines how computations are carried out. Most notably, we divide computational models into stateful and stateless models. Traditionally, we refer to stateless as a declarative model while stateful refers to imperative. Figure one below illustrates the concept of a programming model.

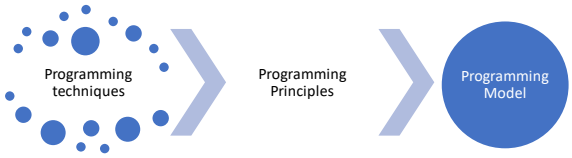

Figure 1 Programming Model

Procedural programming is a stateful computational model where computation's primary means is state mutation (Haridi & Van Roy, 2004). Object orientation is another example of a stateful model. Objects maintain a state, also referred to as private members, and operations that mutate those states. Figure two below gives an illustration of two popular computation models in use today.

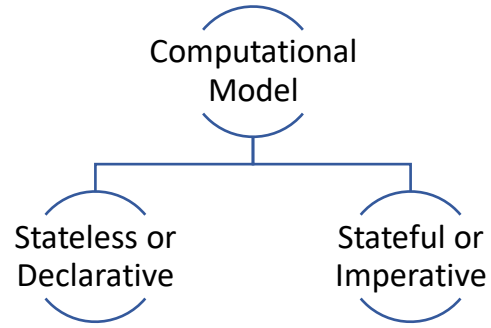

Figure 2 Computational Models

The procedural programming model has strong support for abstraction. As opposed to a declarative model where all system's knowledge is kept outside, stateful models maintain all knowledge inside and provide the ability to manipulate them. Although, without limiting this internal knowledge scope, it becomes much harder to reason about a system under the procedural model. To this end, procedural systems typically follow a component-driven approach where each component encapsulates a set of internal knowledge, increasing modularity.

Composition is a commonly used technique to extend component-driven systems. Components offer new functionalities while delegating sub-tasks to internally contained components. It is worth mentioning that this is a form of dependency.

2. INPUT PARSING

Procedural programming is centered on the idea of grouping a series of computational steps or statements into a referable structure on the stack register. Thus, making it possible to be invoked by the executing thread by referencing its memory address. We refer to these groupings as procedures or routines.

A series of transformations take place before these procedures can be loaded into memory for execution. From a top-down perspective, the input program is encoded and handed off to a lexical analyzer, which asserts that the input stream conforms to the language grammar by generating a series of tokens from the input stream. We refer to this phase as the "lexical analysis" phase. Each language defines what constitutes a valid token pattern through its grammar rules. Lexical analyzers typically rely on regular expressions to match against these rules. After that, the parser validates generated tokens from the lexical analyzer for syntax errors. We refer to this phase as the "syntax analysis" phase. The introduction of a parser, or syntax analyzer, is partially due to regular expressions' limitations. Hence, the lexical analyzer can only perform lexical analysis but not syntax analysis. Upon a successful syntax analysis, the parser generates an abstract syntax tree representing the input program. Figure 3 below illustrates the steps described above.

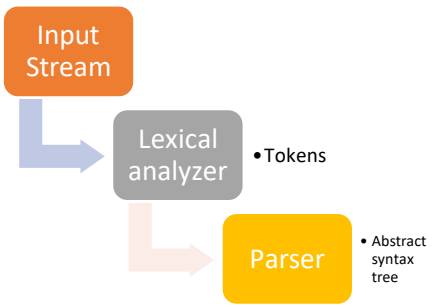

Figure 3 Abstract Syntax Tree generation

With the abstract syntax tree generated, the semantic analysis phase follows. This phase of the process interprets generated symbols, types, and relationships to assert that they carry meaning in the language.

3. CODE GENERATION

To abstract lower-level machine details from the compilation process, the compiler generates intermediate code aimed at an abstract target machine such as the JVM. Following this phase is a machine-independent code optimization process that replaces high-level programming constructs with very efficient low-level code – without altering the intended meaning.

Not all languages, however, pass through intermediate code generation and independent machine optimization. Statically compiled languages, such a C, Fortran, and Lisp, are fully compiled to machine code at compiler time while allowing machine-specific compiler code optimization.

4. EXECUTABLE

Thus far, we have established the following compilation phases: lexical analysis, syntax analysis, and code generation, which results in the creation of an *object file*. With object files, we can increase reusability by grouping a set of routines according to their responsibilities and compiling them into a single object file. A particular program called the *linker* handles the aggregation of object files into an *executable*. Note that the linker can aggregate an arbitrary set of object files. Hence one of the critical responsibilities of the linker is to resolve references across separate object files. It does this by replacing all external references to functions and subroutines with the corresponding memory addresses where the actual corresponding machine code instructions are located on the stack. Finally,

when the operating system is ready to run the generated executable, a program called the *loader* manages the responsibility of loading it into memory. It is important to note here that the loader is part of an operating system, responsible for many tasks such as initializing registers.

5. COMPARISON

This section aims to compare the following procedural languages: C, Fortran and, Lisp by examining their syntax, semantics, structure, and usage. Figure 4 below shows a listing of specific features that we will focus on in this comparison.

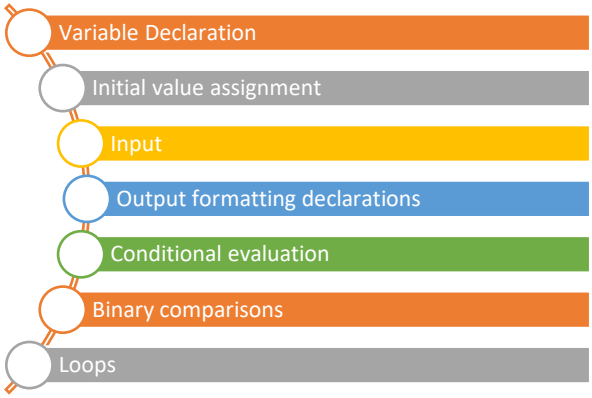

Figure 4 Comparison topics

5.1.1 Declaration and initialization – C

- Syntax:
  - *Type variable_list;*
- Example:
  - *int a, b, c;*
  - *int a = 12;*
- Declares variables a, b, and c as integers
- Type determines occupied space and memory layout (Harbison & Steel, 2002).

5.1.2 Declaration and initialization – Fortran

- Syntax:
  - *Type :: variable_name*
- Example:
  - integer :: age
  - character(len=10)::message
  - logical :: done
- The first one declares an integer variable named age. In comparison, the second one declares a string, message, of length 10. The last example shows support for Boolean type (Oracle Corporation, 2010).

5.1.3 Declaration and initialization – Lisp

- No type declaration for variables (Lisp Community, 2016)
- Global variables are defined using:
  - *defvar* and *defparameter*
- Constant variables are defined using:
  - *defconstant*
- Local variables are defined using:
  - (let (name [initial-value])
- Syntax:
  - *(defvar *name* [initial-value [documentation]])*
  - (setq name expr)
- Example:
  - (defvar *age* 12 "Documentation for variable age")
  - (let ((age 12) (fname "John")))
  - First example declares and initializes variable *age* to 12. While the second example shows declaration and initialization of local variables *age* and *fname*.
  - (setq x 1 y 2 z 3)
  - Initializes:
    - x => 1
    - y => 2
    - z => 3
  - (setq x (1+ y) y (1+ x) z(+ a b)
    - x => 3
    - y => 4
    - z => 7

- sequential variable assignement.

5.2.1 Input and output formatting declarations – C

- Informs the compiler about the data type when using functions that operate on streams, such as *scanf* or *printf* .
- *Scnaf* function is used for reading input off standard input stream while *printf* is used for printing to the standard output stream.
- Syntax:
  - *printf*(char*format, args1, arg2, …);

- Examples:
  - printf("signed integer: %i, decimal floating: %f, character: %c, string: %s", 34, 90.87, 'g', "example");
  - printf("%5d", 24);
  - printf("%05d", 24);
- First example illustrates some of the type specifiers supported in C. They are preceded by the special character '%.' Second and third examples show formatting with desired output length.

5.2.2 Input and output formatting declarations – Fortran

- Fortran supports list-directed input-output formatting.
- Syntax:
  - *read* format, variable_list
  - *print* format, variable_list
  - *write* format, variable_list
- Where *read*, *print* and *write* are standard functions, while *format* indicates the format specification of choice and *variable_list* specifies a list of variables to format.
- A format statement can be defined once and referenced from print function.
- Example:
  - print *, 'My name is', name
  - print "(f12.3)", pi
  - print 1000, x,y,z
    1000 format (1x,3(f5.2))

5.2.3 Input and output formatting declarations – Lisp

- Lisp provides a format function for formatting output.
- Syntax:
  - *format* dest argument-list
  - where *dest* indicates standard output while *argument-list* contains comma-separated output variables.
- Examples:
  - (format t "Distance = ~F~% Speed = ~F" distance speed)
  - Where ~F indicates fixed-format floating-point argument while ~% prints a new line.

5.3.1 Conditional evaluation – C

- Conditional operator syntax:
  - *condition* ? *true_expression*: *false_expression*;
  - Provides a compact if-else statement.
  - If condition evaluates to true, *true_expression* is executed else *false_expression* is executed.
- Example:
  - (4 > 2) ?
    printf("4 is greater than 2")
    :printf("4 is less than 2");

5.3.2 Conditional evaluation – Fortran

- Syntax:
  - IF (condition) THEN
    statements
    .
    .
    END IF
- Fortran supports two logical constants:
  - .TRUE.
  - .FALSE.
- Relational expressions:
  - .LT. (less than)
  - .LE. (less or equal)
  - .EQ. (equal)
  - .GE. (greater or equal)
  - .GT. (greater than)
- Examples:
  - Logical start, done
    start = .TRUE.
    if (.NOT. done) expression
  - if (4 .GT. 2) expression

5.3.3 Conditional evaluation – Lisp

- Lisp supports various forms of conditional evaluation, notably *if*, *when*, and *unless*.
- Syntax:
  - (if (condition)
    (true_expression)
    (false_expression))
- Examples:
  - (if (> z 2)
    (format t "~%z")
    (format t "2"))
  - Prints value of *z* if it is greater than 2, else prints 2)

5.4.1 Binary Comparison – C

- Bitwise operators supported by C:
  - Binary AND operator: &
  - Binary OR operator: |
  - Binary XOR operator: ^
  - Binary one's complement operator: ~
  - Binary left shift operator: <<
  - Binary right shift: >>

5.4.2 Binary Comparison – Fortran

- Bitwise operators:
  - .AND.
  - .OR.
  - .NOT.
  - .EQV. (both true or false)
  - .NEQV.
- Supports three shift operations: logical, arithmetic, and circular.
- ISHFT (j, n)
  - logical shift
  - j: value to be shifted
  - n: number of bits to shift
  - n < 0: right shift performed
  - n > 0: left shift performed
  - n equals to 0: no shift performed

5.4.3 Binary Comparison – Lisp

- Bitwise logical operators:
  - Bitwise AND: logand
  - Bitwise inclusive OR: logior
  - Bitwise exclusive OR: logxor
  - Bitwise NOT: lognor
  - Bitwise Equivalence: logeqv
- Example:
  - (logand x y)
  - (logior x y)
  - (logxor x y)
  - (lognor x y)
  - (logeqv x y)

5.5.1 Loops – C

- Basic loop support:
- *for loop*
  - syntax:
    for (init; condition; increment) {
        statements;
    }

- *while loop*
  - syntax:
    while (condition) {
        statements;
    }

- *do while loop*
  - syntax:
    do {
        statements;
    } while (condition);

5.5.2 Loops – Fortran

- Basic loop support:
- *do loop*:
  - syntax:
    do var = start, stop, [,step]
        statements
    end do
  - Example:
    do n = 1, 10
    ….
    end do

- *do while loop:*
  - syntax:
    do while (logical expr)
        statements
    end do
  - Example:
    do while (n <= 10)
    ….
    end do

5.5.3 Loops – Lisp

- Basic loop support:
- *loop*
  - syntax:
    (loop (expressions))

- *loop for*
  - syntax:
    (loop for loop-variable in <list>
        do (action)
)
    Or
    (loop for loop-variable from v1 to v2
        do (action)
)
  - Example:
    (loop for x in ‘(John Eric Mike)
        do(format t “~s” x)
)

- *do*
  - syntax:
    (do ((variable1 value1 (s-expression)
      (variable2 value2 (s-expression)
      (variable3 value3 (s-expression)
      …)
     (test return-value)
     (s-expressions)
)

- dotimes
  - syntax:
    (dotimes (n count)
        (expressions)
)
  - Example:
        (dotimes (n 10)
            (print n)
)

- *dolist*
  - syntax:
    (dolist (n <list>)
        (expressions)
)

6.0.0 Conclusion

We conclude this article by giving three examples of industry applications predominantly using procedural languages discussed above.

6.0.1 Linux Kernel

The C programming language accounts for 97.5% of the Linux kernel. It is currently an open-source project hosted on GitHub (Linux Foundation, 2021)

6.0.2 OpenBLAS Library

This is an optimized library for linear algebra routines. Fortran language accounts for 48% while C and assemble account for 23% and 26% respectively. It is also an open-source project currently hosted on GitHub (PerfXLab, 2021)

6.0.3 Pgloader Application

This is an application written in Lisp to load data into PostgreSQL from various sources, including other databases. In a single command line, it identifies schema differences and migrates a whole database automatically (Fontaine, 2021)

References

Fontaine, D. (2021, February). Retrieved from https://pgloader.readthedocs.io/: https://pgloader.readthedocs.io/en/latest/#

Harbison, S. P., & Steel, G. L. (2002). Declarations. In S. P. Harbison, & G. L. Steel, A Reference Manual (pp. 74-113). Prentice Hall.

Haridi, S., & Van Roy, P. (2004). Concepts, Techniques, and Models of Computer Programming. The MIT Press.

Linux Foundation. (2021, February). Retrieved from https://github.com: https://github.com/torvalds/linux

Lisp Community. (2016). Style Guide. Retrieved from lisp.lang.org: https://lisp-lang.org/style-guide/

Oracle Corporation. (2010). FORTRAN 77 Language Reference. Retrieved from https://docs.oracle.com: https://docs.oracle.com/cd/E19957-01/805-4939/

PerfXLab. (2021, February). OpenBLAS. Retrieved from https://github.com: https://github.com/xianyi/OpenBLAS