7

WHAT IS A GENERAL STRUCTURE FOR COMPILING?

Reference Materials

Terrence W. Pratt and Marvin V. Zelkowitz. *Programming Languages: Design and Implementation*. 4th Ed. Upper Saddle River, NJ: Prentice Hall. 1995. Chapters 1 and 3.

Kenneth C. Louden. *Programming Languages: Principles and Practice*. 2d Ed. Belmont, CA: Thompson, Brooks/Cole. 2003. Chapters 1 and 3.

7.1 A GENERAL FRAMEWORK

We have been studying language, both natural and formal. Formal languages are languages that have completely specified syntax and semantics. What we believe at this point is that programming languages are not all that different from natural languages in their overall concepts. We should expect that programming languages have the same parts (Figure 7.2): although syntax deals with the overall appearance of the language, the meaning of the words is the purview of semantics; informally semantics has taken on the meaning of "meaning of a phrase or sentence." Regardless, there must be a lexicon and a vocabulary. The lexicon is the entire set of words, whereas the vocabulary is words and definitions. We also have studied semantics in terms of its relationship to the deep structure of a statement.

From a project management standpoint there are some questions: who, what, when, where, why, and how are each of these different parts defined?

CASE 44. MERGING LINGUISTICS AND COMPILING

Write a story about how the sentence, "The cat ate the hat." is understood using the phases listed in Figure 7.1. In linguistics and logic, we speak of the binding time of a word. The binding time is the moment when the definition of a word is assigned. In an abuse of language, we can extend this concept to relate to the assignment of meaning for any element: word, phrase, sentence, paragraph, ... When are the meanings of the five words bound? When is the meaning of the sentence bound?

Meaning is not something that just happens: it occurs as we read the sentence. We can only understand the meaning of the sentence when we understand the meaning of the underlying words, phrases, etc. The same must happen in the compiler.

CASE 45. PROCESS ORDERING

Develop a time line representation of the translation of a C statement `x = 1*2+3` into the Forth statement `1 2 3 * +`. Annotate the time line with the phases in Figure 7.1. When is the lexicon consulted? How is the vocabulary used?

7.2 TYPES

When you study the assembly language for a computer (not just the chip) it is apparent that the vast majority of instructions manipulate various data types: integers, floating point numbers, information to and from devices, among others. It would be difficult to move a program from one computer to another if there were not some organization to this data and some

(1) Syntax
    (a) Morphology
        (i) Spelling rules
        (ii) Lexicon
        (iii) Vocabulary
    (b) Grammar and deep structure
(2) Semantics
(3) Pragmatics

Figure 7.1 Phases in Compiling

agreement on what various operations are called. This organization is called *data types* or just *types*.

In order to explain types, let's look at the quintessential type: integers. Everyone knows about integers; everyone knows that "+" is read "plus" and stands for the operation "take two integers and sum them giving an integer as a result." Everyone also knows that "<" is read "is less than" and stands for the condition "one number is less than another number, giving a true or false answer." Everyone also knows that "if a is less than b and b is less than c, then a is less than c."

For our purposes, we can start by defining a *type* as a triple of the form $\langle S, F, R\rangle$, where S is a set of constants, F is a set of function symbols, and R is a set of relation symbols.

$$\text{Integers} = \langle integers, \{+, -, \ldots\}, \{=, \neq, <, \ldots\}\rangle$$

It should be clear that S, F, and R determine the lexicon for integer expressions (without variables). The harder part is to develop algorithms for each element in F and R.

CASE 46. PRIMITIVE DATA TYPES

Develop a type definition for each of the standard types: booleans, integer, floating point, and strings.

7.3 RECAP

In this chapter we have laid the groundwork for the project. We have a time line of who, what, when, where, and why elements of a statement are processed and given meaning. This is the fundamental step in any project. We are missing the *how* for many of these steps. The purpose of the Milestone chapters in Part I is to fill in the *how* and to implement a simple compiler that takes simple statements to Forth.

7.4 DEFINING A LANGUAGE

In order to understand the requirements in the milestones, we need to understand the overall issues in a programming language.

1. Architectural
   - Registers
   - Control word
   - Control flow
   - Input/output
2. Primitive data
   - Arithmetic—integer and floating point
   - Logical (bit level)
   - Character
   - Array support
3. Pseudo-instructions
   - Value naming
   - Relative addressing
   - Static storage allocation

Figure 7.2 Classes of Statements in Assembler Language

7.4.1 Algorithms and Data Representation

Fundamentally, a programming language is a vehicle for describing algorithms and data representations. In the early days (1950s and 1960s), the major implementation vehicle was assembly language. Assemblers make algorithms and data representations quite clear because only primitive machine data types are available. The major categories of statements are shown in Figure 7.2.

You can see that we still have those same classes of statements. While hardware is faster and with more memory, the same classes of statements are present in today's chips.

7.4.2 Evolution

Why did programming languages evolve? The simple answer is that programming large systems in assembler language is very costly and very error prone. One of the first steps was to eliminate the deep connections between the program and the architecture. Programming at the assembler level is by definition tied to a specific computer on which the implementation is done. This causes a major problem: the code must be changed if there is any change in the underlying machine. The ramifications were far reaching in the 1960s: manufacturer dominance was equivalent to software system dominance. This stifled innovation in the hardware arena because the cost

of recoding a system rarely could be justified by the advantages of new hardware except for pure speed.

By 1960 there were several higher-level languages that abstracted the computer from the algorithm; the most influential were ALGOL, Cobol, Fortran, Lisp, and Snobol. Fortran remains heavily used in scientific programming and Lisp continues to be important in artificial intelligence applications. Object-oriented concepts emerged in 1967 with Simula67. Therefore, by 1970, the stage was set for evolution in many directions.

Higher-level languages effectively removed the architectural class by having variables and data controlled by the compiler. Some languages continue to have input/output defined in the language, but most have now eliminated that from the language definitions and moved to using a library of subprograms instead.

Looking at the pragmatics of programming language development, there has been a steady movement from complex languages (PL/I) and smaller libraries to smaller languages with much larger library support (JAVA and its API). The one major exception has been Fortran, which has always had extensive library support due to its use in numerical processing. For example, C was develop almost directly in revolt against the complexity of PL/I and Pascal. The C/Unix pair, developed at Bell Laboratories in the late 1960s and early 1970s, became the standard in computer science education; C's minimalistic nature influenced many language designers.

Newer languages such as ML that have significant complexity introduce expanded function declaration mechanisms, such as *modules*. Modules encapsulate operations for entire data types, for example. Objects, on the other hand, encapsulate data types along with memory allocation.

7.4.3 Implications for the Project

Regardless of the exact design of a language, we will be able to develop and categorize various saspects of the language in a manner similar to that shown in Figure 7.3. Each and every aspect will have an impact on all the possible phases of the compiler.

CASE 47. DESIGNING MODULES

Use the form given in Figure 7.4 to develop the details needed. Some language elements may have entries in all columns, some may not. **Note:** The semantics column must reference a Forth word in the table and that word must be defined below the table.

1. Architectural
   - Control flow
     a. if-then-else
     b. for / do
     c. while
     d. subprograms: functions and subroutines
   - Input/output subprograms
2. Primitive data
   - Arithmetic—integer and floating point
   - Logical (bit level)
   - Character and strings
   - Arrays
   - Pointers and dynamic allocation
   - Structures and objects
3. Pseudo instructions
   - Variables and assignment statements

Figure 7.3 General Classes of Statements in Higher-Level Languages

	Syntax		Type	Semantics	Pragmatics
Lexical	Grammar	Vocabulary			

Figure 7.4 Design Table for Language Features

Each symbol with *semantic* meaning must be in the table. From a practical standpoint, it is probably best to start with the various primitive data types and operations.