6

LINGUISTICS FOR PROGRAMMING LANGUAGES

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. Chapter 1.

Web Sites

Wikipedia: search "metalanguage," "surface structure," "deep structure," "transformational grammar."

docs.sun.com

6.1 METALANGUAGE VERSUS OBJECT LANGUAGE

Speaking about languages adds a strange twist to the conversation. How do you distinguish between the language you are talking about and the language you are using to describe features? For example, when you learned a programming language such as C, you probably used a natural language such as English to describe a feature of the language C. In this case, we use the natural language as a metalanguage while C is the object language. The distinction is important because during the project, there are several languages in use: the production language for the parser, the type definition language for types, Forth, and Gforth. We must be very clear as to which (if any) is the object language and which is the metalanguage.

A metalanguage describes an object language. The prefix "meta-" in the case at hand denotes a language that deals with issues such as behavior, methods, procedures, or assumptions about the object language. The object language is the language system being described.

This distinction becomes evident when reading programming language specification. This chapter uses the Java specification as an exemplar for language specifications. The specification is available at `java.sun.com`.*

6.2 SURFACE STRUCTURE AND DEEP STRUCTURE

When you read natural language, there are certain features of the language, such as word order or spelling, that we recognize as belonging to that language. These features we call *surface structure*. For example, native English speakers would probably question the word "al Asqa" as being an English word. But there is another structure: a *deep structure*. Deep structure deals with the manner in which lingustic elements are put together in a sentence. In Chapter 5 we developed the syntax tree of a sentence and we discussed the idea of sentence diagrams; both of these concepts are deep structures.

CASE 39. WHAT IS THE PROPER WAY TO FORM JAVA SYMBOLS?

In Chapter 5, Section 5.4, we introduced the concept of morphology: the structure of words. A synonym for morphology is **lexical structure**, which is also related to the concept of surface structure. Certainly, programming languages have many sorts of words, including numbers and quasi-mathematical symbology. How should we define the spelling rules for a language. We can study a well-developed language such as Java to find out.

For this case, refer to Chapter 3, "Lexical Structure of the Java Language Specification," in (name a book?). Please note that you may first have to read Chapter 2, "Grammars," to familiarize yourself with the manner in which production rules are portrayed. Answer the following questions:

1. What is **unicode** and why is it important? Is unicode just arbitrary numbers? Could a Russian programmer use a cyrillic keyboard for Java?
2. Read the chapter and develop a lexicon of lexical concepts in programming languages. For example, **floating point numbers** and **identifiers** are two such types. In this table, you should have major headings (such as "Numbers") and subheadings (such as "Integer").

* Accessed in April 2005.

CASE 40. WHAT ARE THE PARTS OF SPEECH IN JAVA?

Read Chapter 18, "The Grammar of the Java Programming Language," in (name of book?) What is the equivalent to the parts of speech in natural language? **Hint:** Link the lexical categories to parts of speech as terminals for the grammar.

CASE 41. HOW IS THE SYNTAX OF JAVA PRESENTED TO DEVELOPERS

It is time to consider how we communicate the technical aspects of the syntax of a programming language. Again consider Chapter 2 and Chapter 18 of the Java specification.

Starting with the nonterminal *statement* develop a parse for the statement

```
a = b+c;
```

CASE 42. GRAPH MODELS OF SYNTAX

One of the major reasons for using the context-free type of grammar is that we can develop a very usable data structure: *an* n-ary tree or a directed acyclic graph (DAG). The procedure for developing such a graph is the following.

Suppose we have the following context-free production

$$N \rightarrow X_1 \; X_2 \; \ldots \; X_n$$

where N is a nonterminal and the X_i are either terminal or nonterminal. We can form a *rooted, ordered n-ary tree* by making the root of the tree, and the results of the parse of X_1 is the first child, X_2 is the second child, and so on.

Using the parse of the previous case, draw the tree.

6.3 STRUCTURAL INDUCTION

The fundamental approach to designing semantics is called *structural induction.* Consider once again Old Faithful:

$$\begin{aligned} E &\rightarrow T + E \mid T - E \mid T \\ T &\rightarrow F * T \mid F/T \mid F\%T \mid F \\ F &\rightarrow I \in \text{int} \mid (E) \end{aligned}$$

How many different *forms* can be generated from this grammar? The short answer is "There are an infinite number of possible forms." The naïve answer would say that $1 + 2 * 3$ and $7 + 5 * 12$ are different forms: surface forms!

How many forms from the deep structure? There are just nine: one for each right-hand side of the productions. Thus, we can build up, or more properly *induce*, the structure from the productions. For this reason, the technical name for this method is called *structural induction* and a companion approach for recursion, structural recursion.

CASE 43. STRUCTURAL INDUCTION

This method is central to our understanding of many applied and theoretical algorithms. Write a research paper on structural induction and structural recursion. For cases, a research paper is intended to be one or two pages long. The focus questions in this exact instance are, "What is a definition of each term?" and "Demonstrate that you understand by producing one or more examples." In this paper, give a concrete problem (such as string generation) and illustrate the use of structural induction to generate possible answers. Similarly, develop a recursive definition of the same problem and demonstrate structural recursion.

6.4 INTERPRETATION SEMANTICS

The purpose of a program is to produce *values*. In order to have the computer do useful work we must, in a stepwise fashion, instruct the computer on exactly how values are to be computed. You are somewhat familiar with this idea from computation in mathematics. In the early grades, we work with numbers and operations only. We are the computer in this circumstance and we are the ones who must learn to add, subtract, multiply, and divide. As we become more advanced, we learn to compute values of trigonometric functions and logarithms. When we learn calculus, we learn how to compute with symbols using the rules of differential and integral calculus.

In each case, we learn three things: (1) how to formulate problems, so that we can (2) write down computations so that they are understood by everyone, and (3) how to convert the symbols into values. We use mathematical notation in, for example, calculus class, that has evolved over a 400-year time span. Programming languages are much younger (1950s), but they fulfill the same purpose as mathematical notation: a system of symbols (language) that express algorithms (programming). In the case of programming languages we can write algorithms that compute symbols as well as numbers. One such class of programs is *compilers and interpreters* that read programs and cause the algorithms to be executed. In order to do this, we must agree on various words and constructs are converted to computer instructions. We call this agreement *semantics*.

We have discussed semantics in Chapter 5, where we defined it as the study of meaning of words, phrases, sentences, and texts. The same general definition applies here, except we are thinking about a *formal, mathematical* meaning. Here's an example: the mathematical statement

$$3 + sin(2\pi)$$

can be carried out by the Forth statement

3 2 PI * fsin + if PI has been set equal to a proper approximation to π:

3.14159265358979323846 according to the IEEE standard.

What then is the *semantics* of `3 2 PI * fsin +`? We will take the view in this text that the semantics of programming language is the value computed by the expression, or the *denotational semantics*. Denotational semantics are based on the λ-calculus. The λ-calculus is discussed in Chapter 9.

There are several approaches to language, denotational being one. *Axiomatic semantics* is developed in the same manner as any axiomatic (mathematical) system. Axiomatic semantics are closely associated with Edgers Dijkstra, David Gries, and Sir C. A. R. Hoare. This approach is perhaps informally known to you because it introduces the concepts of *precondition* and *postcondition*. The fundamental inference rules are formulated by

$$\{P\}S\{Q\},$$

where P is the precondition, Q is the postcondition, and S is the command. In axiomatic semantics, the semantics name not the value (as in operational and denotational), but the conditions under which the values exist, the expression's connotation. An informal use of this concept is seen in testing because test cases require the tester to understand the three parts of the formula.

An *operational semantics* is a mathematical model of programming language execution that utilizes a mathematically defined *interpreter*. This mathematically defined interpreter is actually in the spirit of the definition of Gforth. The first operational semantics was based on the λ-calculus. Abstract machines in the tradition of the SECD machine are closely related. SECD was introduced by Peter Landin in 1963. His definition was very abstract and left many implementation decisions to the implementors. This situation gave rise to research by Plotkin (1981) and later investigators.

6.5 WHAT ABOUT PRAGMATICS?

Finally, then, how do programming languages display pragmatics? Actually, the answer is simple: the pragmatics are seen in the structure of programs. Recall that "pragmatics" means the manner in which the language is used. Simply, then, programming language pragmatics are the elements of programs and the learned idioms. For example, the English language statement "increment `i` by one" is pragmatically displayed in C in several ways: `i++`, `i+=1`, and `i=i+1`. These are semantically equivalent but may still be pragmatically different because they are used in different circumstances.