By Dr. Muhammad Nur Yanhaona

Lesson 3: Introduction to Syntax Analysis

Introduction

Syntax analysis is the second stage of the compiler that comes after the lexical analysis. Syntax analysis, as the name indicates, is the analysis of the structure of a program. A syntax analyzer, that does the analysis, checks whether a given input program is structurally valid. As in natural language, structural validity is equivalent to grammatical correctness. For example, the grammar of English verbal sentences requires that sentences have the structure where a subject is followed by the verb then an object. Writing it otherwise is wrong. Similarly, an assignment statement of a Python/C/Java program can only have an identifier on the left followed by the assignment operator (i.e., =) then an expression. We cannot put a number, a function call, or another expression on the left of the assignment operator.

Using grammars for specifying the syntactic rules of a programming language has several advantages. First, it provides and easy to understand specification of the syntax. Second, most often the syntax analyzer can be built automatically from a grammar as we built the lexical analyzer atomically from the regular expressions for tokens. Third, we can easily enhance the capabilities of an existing language by including more grammar rules. Finally, grammar checking of the input program allows it to be translated into an intermediate form that is more convenient for processing in the later stages of the compiler.

As syntax analysis can be thought as taking an entire program as an essay and then parsing its contents into smaller parts then investigating how they incrementally generate larger components from smaller parts and eventually form the program, the process is also called parsing and the syntax analyzer a parser. There is, however, an important distinction between parsing a natural language essay and parsing a program. In case of an essay, the natural language grammar is only concerned about the structure of the sentences. That is why, we never say an essay is grammatically incorrect. Rather, we say it has grammatical errors in its sentences. When parsing a program, on the other hand, we validate both its parts (that is, statements and expressions) and itself as a whole. In other word, the grammar of a programming language defines the rules for the structure of the entire program.

There is a final similarity between parsing a natural language essay and a program. As in the case of an essay, we read the words and at the same time check grammar, lexical analysis and parsing of a program also happen concurrently. In fact, the parser asks the lexical analyzer to send it the next token then the lexical analyzer resumes reading characters of the program file from where it stopped after sending the last token. Later you will learn that some parts of semantic analysis, that is assessing the meaning of the program, are also done concurrently with parsing.

The Grammar for Programming Languages

In fact, the rules for writing functions, loops, conditionals, and building block data structures such as, arrays, and strings are chosen for programming languages in a manner that makes it easier to validate the syntax of the language. For example, consider a Python function definition as follows:

```
def printHello():
    print('Hello there!')
```

The choice of a specific 'def' keyword for functions, the parenthesis after the function name, the colon afterwards, and the indentation of the print statement are all needed to simplify writing a grammar specification for function definitions. So, they are simply not for pretty appearance. You will notice that how you define a class is different from how you define a function. Similarly, dictionary and list are different in syntax. Some other languages, such as C and Java requires a ';' at the end of each program statement and curly braces around loops and functions. These things are like so, again, for grammar simplification. Why is it important to have a simple and clear syntax as opposed to more complex one? The reason is, otherwise, it is difficult to write a syntax analyzer.

The grammar virtually all programming languages use for specifying the syntax is **the context free grammar**. Context free means, when you try to validate a specific rule of the grammar, you do not need to consider what is the context for checking that rule. Which is unlike the grammar for natural languages, which is called a context-full grammar. An example will make the distinction clear. Consider the following natural language speech:

'I heard that Breaking Bad is a wonderful TV series. Have you watched it?'

Now to understand the second sentence of the speech. You need to remember that it refers to Breaking Bad TV series. Without the context provided by the first sentence, the it in the second sentence could be anything, for example, even a dog! However, in a Python program if you have the following:

```
if x > 10:
    y = z + k
```

The assignment statement inside the conditional does not need to understand what encircles it. When you use a context free grammar to specify the syntax of the programming language, the alphabet (called the terminals in grammar terminology) of the language become all the tokens lexical analyzer can returns. The grammar rules, called productions are described using some placeholders, that are the container for those rules, and are called non-terminals. For example, you can write the grammar for assigning the result of an arithmetic expression to a variable as follows:

```
A -> id = E

E -> E + T | E – T

T -> T * F | T – F

F -> id | number | (E)
```

Grammar 1

Here A, E, T, and F are the non-terminals and the rests are terminal (i.e., tokens). A production rule tells how the placeholder non-terminal at the left side of -> can be replaced with a string of terminals and non-terminals, called the body of a production. You learned all about context free grammars in your automata theory course. The full grammar of a programming language is no different from what you have seen there. It is just lengthy as there are so many rules.

Validating the Syntax of a Program

The idea of validating the syntax of a given program is to check whether the grammar of the underlying programming language can generate the program using the available production rules starting from a specific non-terminal, called the start symbol of the grammar. If we can generate the input program by repeated application of production rules then the program is syntactically valid. The process of generating the program from the start symbol of the grammar is called derivation. For example, consider the assignment statement x = (y + 10) * z. It can be derived from the grammar we shared earlier as follows:

A -> id = E -> id = E * T -> id = T * T -> id = F * T -> id = (E) * T -> id = (E + T) * T -> id = (T + T) * T -> id = (F + T) * T -> id = (id + T) * T -> id = (id + F) * T -> id = (id + number) * T -> id = (id + number) * F -> id = (id + number) * id.

Notice that the input was **x = (y + 10) * z** but we derived **id = (id + number) * id**. This is how things work as the lexical analyzer will give the parser id token for x, y and z; and a number token for 10.

It should be clear that a syntactically valid program is a program that has a derivation. Otherwise, it is invalid.

Ambiguous Grammars

We deal with a lot of ambiguities in our daily communication using natural language. For example, it is unclear what animal is referred by the pronoun 'it' in the following speech.

The Jackel swiftly killed the rabbit. It did not make any sound.

We resolve ambiguity from the context of the statement that we construct using what being said earlier and using our other organs such as our vision and hearing. If nothing works to resolve ambiguity then we can ask the speaker questions for clarifications. However, the same cannot be done for programming languages. The compiler must know the exact interpretation to be able to generate a machine executable that behaves exactly same as the programmer intended. Therefore, the grammar we use for the syntax of a programming language cannot be ambiguous. A grammar is ambiguous when two distinct sequence of production rule applications can derive the same program. For example, if we write the grammar for arithmetic expression as follows, instead of the earlier grammar we shared, then the grammar is ambiguous:

E -> E + E | E - E | E * E | E / E | id | number | (E)

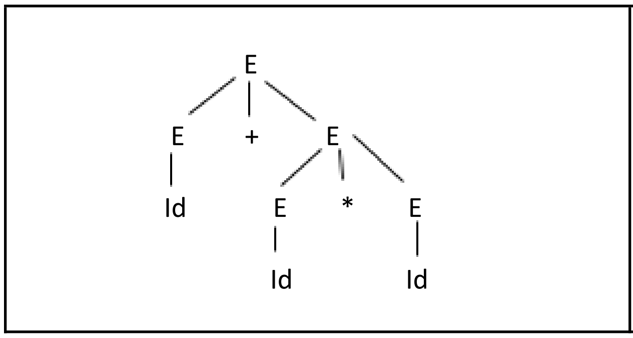
Grammar 2

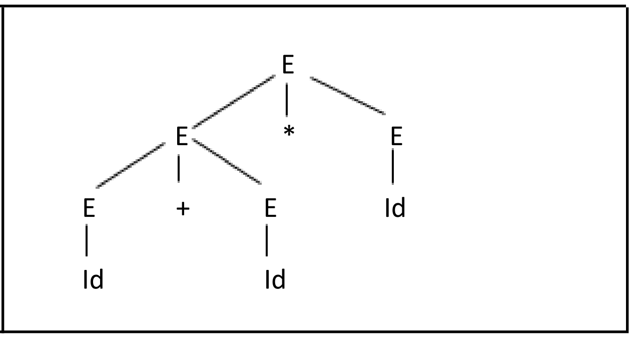
How do you prove that the grammar is ambiguous? By showing an example of two distinct derivations for a single input expression. For example, with this new grammar the expression 'id + id * id' can be derived in the following two different ways:

1. E -> E + E -> id + E -> id + E * E -> id + id * E -> id + id * id
2. E -> E * E -> E + E * E -> id + E * E -> id + id * E -> id + id * id

It is not particularly clear from the derivation sequences that how the above two differ in machine executable generation. Actually, the first sequence means adding a value with the multiplication of two other values while the second means multiplying a value with the addition of two values. Both are definitely not the same. To understand how the compiler is going to interpret a derivation, we draw a tree that makes clear the ordering of production rules and consequent program operations. This tree is called the parse tree.

The parse trees for the first and second derivations are as follows:





The simple strategy to check for ambiguity, is the check if two parse trees are possible for any input. If yes then the grammar is ambiguous, otherwise, it is not.

You will realize that the Grammar 1 we have earlier only allows a parse tree that is similar in appearance to the first parse tree in the above but it does not allow the second one. However, even with Grammar 1, two parse trees are possible for expressions like 'id + id + id.' The solution to deal with this kind of ambiguity is often not in an updated grammar. Rather, explicit instructions are given to the syntax analyzer to prefer one derivation over another in cases like these. Such instructions are called precedence rules in programming language terminology.