

Lecture 11

Syntactic Analysis

Consider the following C++ function. There are a number of syntax errors present.

```
1.    int* foo(int i, int j))
2.    {
3.        for(k=0; i j; )
4.            fi( i > j )
5.        return j;
6.    }
```

Line 1 has extra parenthesis at the end. The boolean expression in the for loop in line 3 is incorrect. Line 4 has a missing semicolon at the end. All such errors are due to the fact the function does not abide by the syntax of the C++ language grammar.

Semantic Analysis

Consider the English language sentence “He wrote the computer”. The sentence is syntactically correct but semantically wrong. The meaning of the sentence is incorrect; one does not “write” a computer. Issues related to meaning fall under the heading of *semantic analysis*. The following C++ function has semantic errors. The type of the local variable **sum** has not been declared. The returned value does not match the return value type of the function (**int***). The function is syntactically correct.

```
int* foo(int i, int j)
{
    for(k=0; i < j; j++ )
        if( i < j-2 )
            sum = sum+i
    return sum;
}
```

Role of the Parser

Not all sequences of tokens are program. Parser must distinguish between valid and invalid sequences of tokens. What we need is an expressive way to describe the syntax of programs and an acceptor mechanism that determines if input token stream satisfies the syntax of the programming language. The acceptor mechanism determines if input token stream satisfies the syntax of a programming language.

Parsing is the process of discovering a *derivation* for some sentence of a language. The mathematical model of syntax is represented by a grammar G . The language generated

by the grammar is indicated by $L(G)$. Syntax of most programming languages can be represented by *Context Free Grammars* (CFG).

A CFG is a four tuple $G=(S,N,T,P)$

1. S is the *start symbol*
2. N is a set of *non-terminals*
3. T is a set of *terminals*
4. P is a set of *productions*

Why aren't Regular Expressions used to represent syntax? The reasons is that regular languages do not have enough power to express syntax of programming languages. Moreover, finite automaton can't remember number of times it has visited a particular state.

Consider the following example of CFG

SheepNoise ? *SheepNoise* baa
| baa

This CFG defines the set of noises sheep make. We can use the *SheepNoise* grammar to create sentences of the language. We use the productions as rewriting rules

Rule	Sentential Form
-	<i>SheepNoise</i>
1	<i>SheepNoise</i> <u>baa</u>
1	<i>SheepNoise</i> <u>baa</u> <u>baa</u>
2	<u>baa</u> <u>baa</u> <u>baa</u>

While it is cute, this example quickly runs out intellectual steam. To explore uses of CFGs, we need a more complex grammar. Consider the grammar for arithmetic expressions:

1	<i>expr</i>	?	<i>expr op expr</i>
2			<u>num</u>
3			<u>id</u>
4	<i>op</i>	?	+
5			-
6			*
7			/

Grammar rules in a similar form were first used in the description of the Algol-60 programming language. The syntax of C, C++ and Java is derived heavily from Algol-60.

The notation was developed by John Backus and adapted by Peter Naur for the Algol-60 language report; thus the term Backus-Naur Form (BNF)

Let us use the expression grammar to derive the sentence

$$x - 2 * y$$

Rule	Sentential Form
-	<i>expr</i>
1	<i>expr op expr</i>
2	<i><id, <u>x</u>> op expr</i>
5	<i><id, <u>x</u>> - expr</i>
1	<i><id, <u>x</u>> - expr op expr</i>
2	<i><id, <u>x</u>> - <num, <u>2</u>> op expr</i>
6	<i><id, <u>x</u>> - <num, <u>2</u>> * expr</i>
3	<i><id, <u>x</u>> - <num, <u>2</u>> * <id, <u>y</u>></i>

Such a process of rewrites is called a *derivation* and the process of discovering a derivation is called *parsing*. At each step, we choose a non-terminal to replace. Different choices can lead to different derivations.

Two derivations are of interest

1. *Leftmost*: replace leftmost non-terminal (NT) at each step
2. *Rightmost*: replace rightmost NT at each step

The example on the preceding slides was *leftmost derivation*. There is also a *rightmost* derivation. In both cases we have

$$expr \quad ? \quad * \quad \underline{id} - \underline{num} * \underline{id}$$

The two derivations produce different *parse trees*. The parse trees imply different evaluation orders!