Compiler Lab

Tutorial #1

- The *lexical analyzer* (often called a *scanner* or *tokenizer)* translates the input into a form that's more useable by the rest of the compiler.
- Extract the *words (lexemes)* of the input string and classify them into *token classes* according to their roles.
- The tokens are typically represented internally as unique integers or an enumerated type.
- Both components are always required-the token itself and the lexeme.
- Example: to differentiate the various **name or number** tokens from one another.

- A parser is a group of functions that converts a token stream into a parse tree.
- A parse tree is a structural representation of the sentence being parsed.
- Parse tree represents the sentence in a hierarchical fashion, moving from a general description of the sentence (at the root of the tree) down to the specific sentence being parsed (the actual tokens) at the leaves.

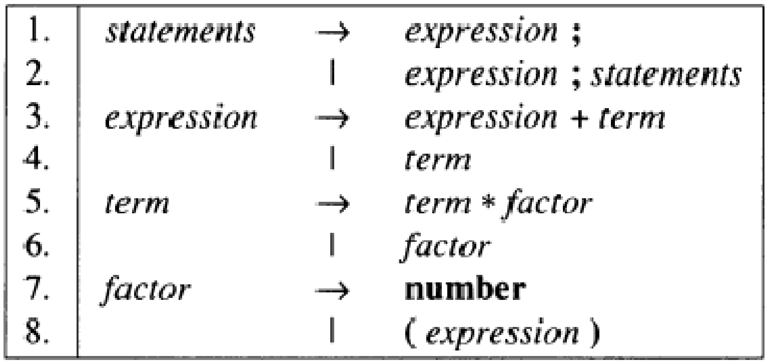
- Code Generation – Code is generated by the same functions that parse the input stream.
- The intermediate language is a subset of C.
- Most instructions translate directly to a small number of assembly-language instructions of a typical machine.

- For example, an expression like “a+b*c+d” is translated into something like this:

```
t0  = _a
tl  = _b
tl *= _c
t0 += tl
t0 += _d
```

- The t0 and t1 in the foregoing code are *temporary variables* that the compiler allocates to hold the result of the partially-evaluated expression.
- The underscores are added to the names of variables declared in the input program (by the compiler) so that they won't be confused with variables generated by the compiler itself, such as t0, and tl (which don't have underscores in their names).

A Simple Expression Grammar

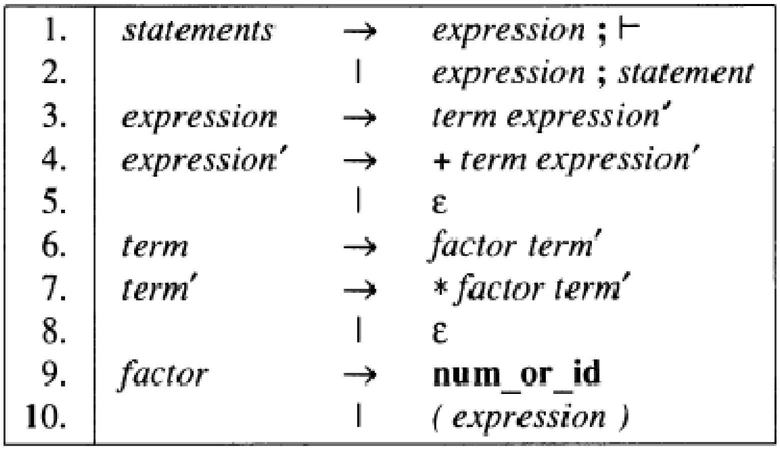


1.	*statements*	→	*expression* ;
2.		|	*expression* ; *statements*
3.	*expression*	→	*expression* + *term*
4.		|	*term*
5.	*term*	→	*term* * *factor*
6.		|	*factor*
7.	*factor*	→	**number**
8.		|	(*expression*)

- *Statements* are made up of a series of semicolon-delimited expressions, each comprising a series of numbers separated either by asterisks (for multiplication) or plus signs (for addition).
- The grammar is recursive. For example, Production 2 has *statements* on both the left- and right-hand sides.
- This grammar has a major problem. The leftmost symbol on the right-hand side of several of the productions is the same symbol that appears on the left-hand side.

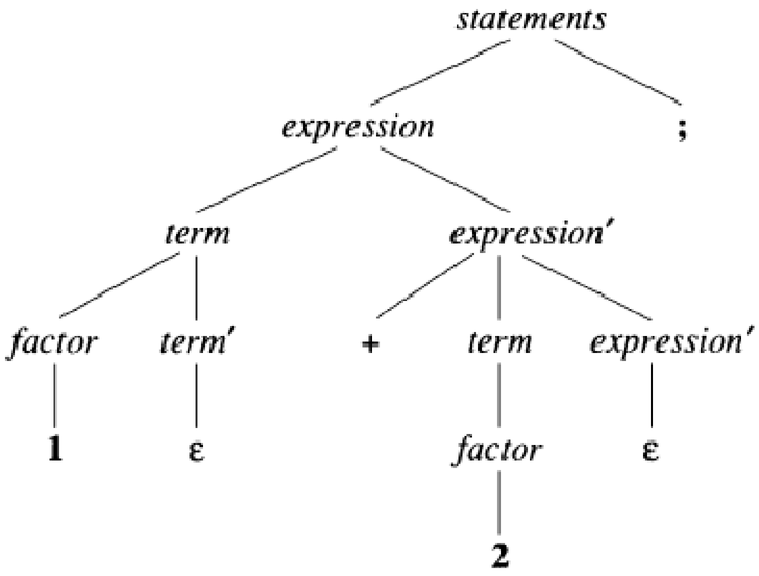
- For example, in Production 3, *expression* appears both on the left-hand side and at the far left of the right-hand side. The property is called *left recursion.*
- We will use *recursive-descent parser* in this assignment. It is a top-down parser that starts from the root symbol and usually progresses by repetitively replacing the leftmost symbol on the right-hand side with the right-hand side of an appropriate production.
- A *predictive parser* is a recursive descent parser that does not require backtracking.
- A *recursive-descent parser* can't handle *left recursion*. They just loop forever, repetitively replacing the leftmost symbol in the right-hand side with the entire right-hand side.

Modified Expression Grammar



1.	*statements*	→	*expression* ; ⊢
2.		|	*expression* ; *statement*
3.	*expression*	→	*term expression′*
4.	*expression′*	→	+ *term expression′*
5.		|	ε
6.	*term*	→	*factor term′*
7.	*term′*	→	* *factor term′*
8.		|	ε
9.	*factor*	→	**num_or_id**
10.		|	(*expression*)

A parse of 1 + 2


statements
expression
;
term
expression′
factor
term′
+
term
expression′
1
ε
factor
ε
2