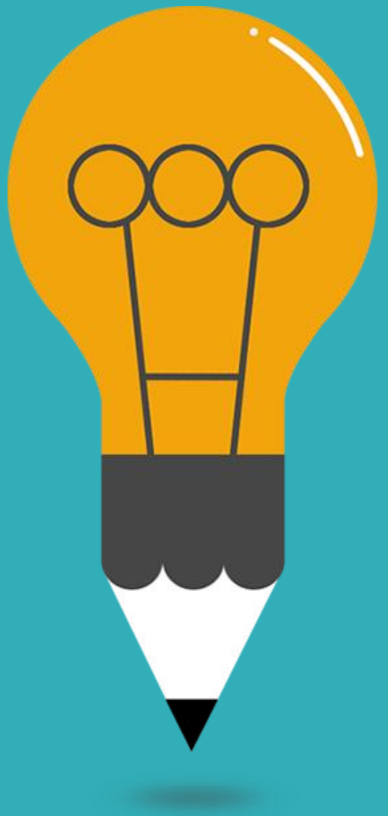
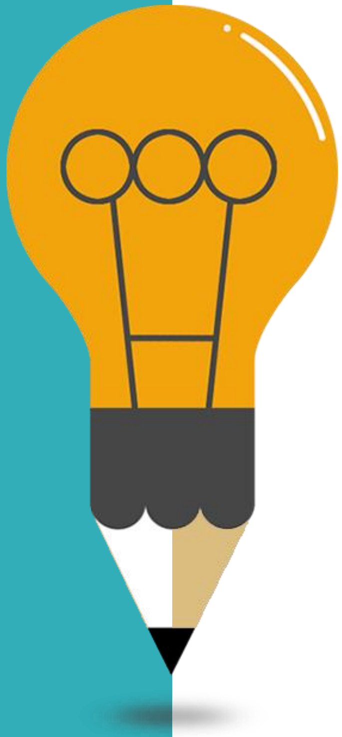


Compiler Design

Course1

By Waseem AlBizreh

Topics



01 **Introduction**
Compiler and Compiler Stages

02 **Lexical Analysis**
Tokens and Lexemes

03 **Syntax Analysis**
Parser type

04 **Parse Tree**
AST and Symbol

Compiler

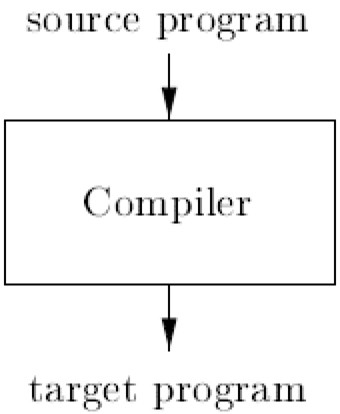


Figure 1.1: A compiler

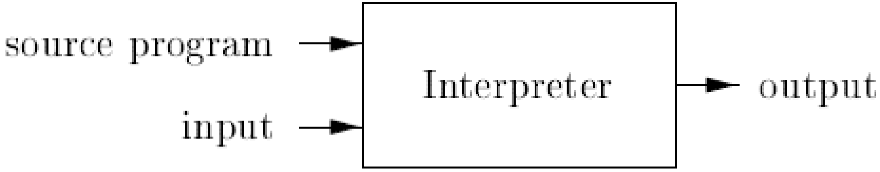
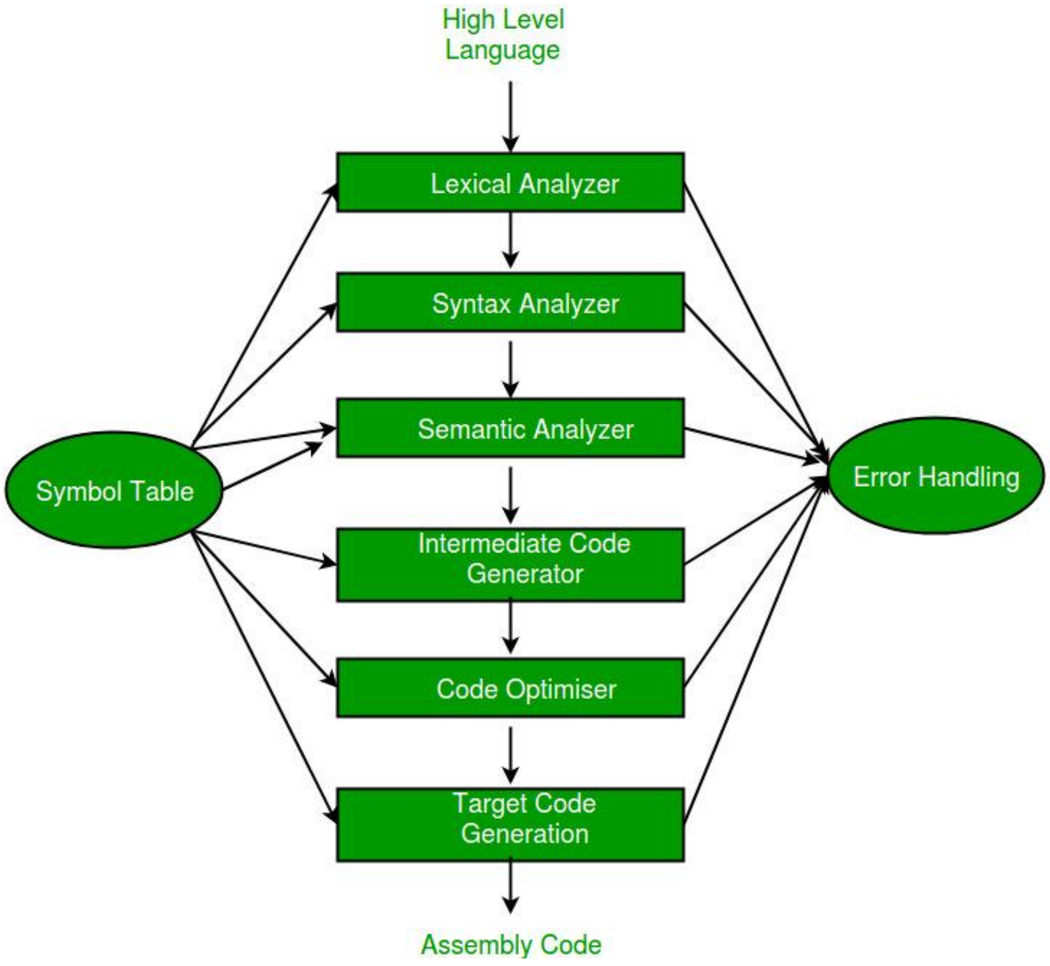


Figure 1.3: An interpreter

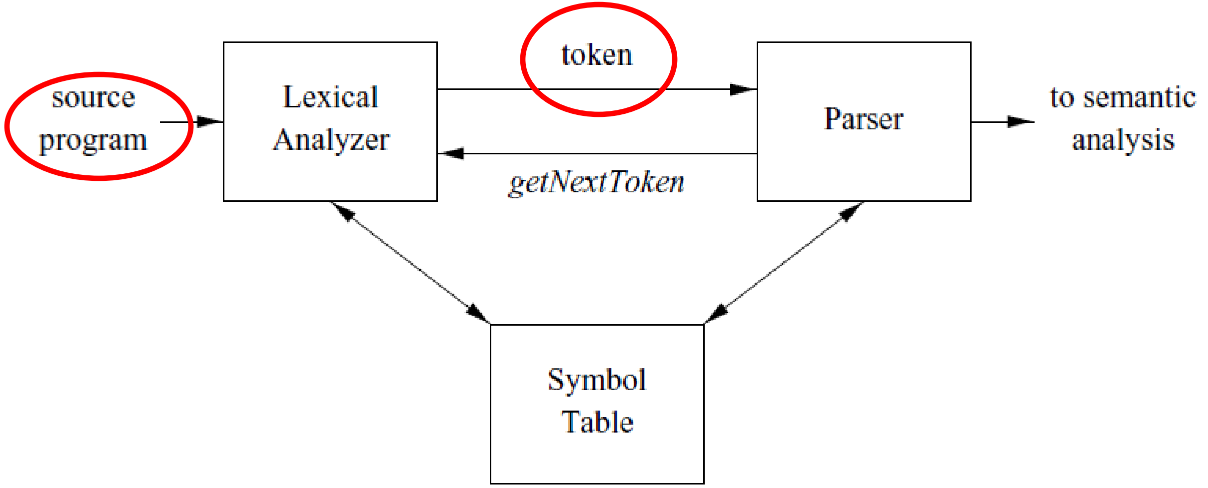
Compiler Stages

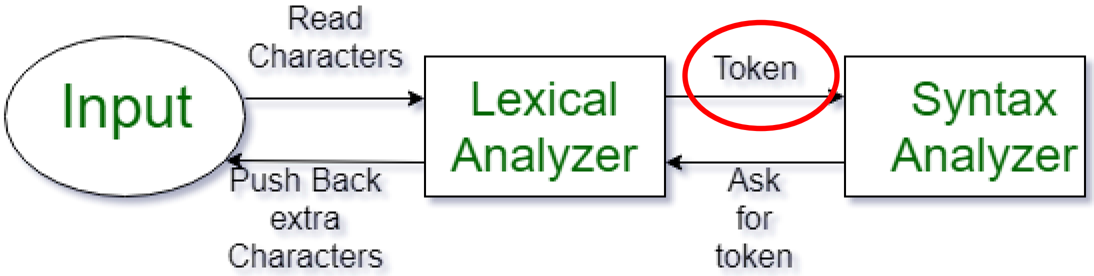
- **Lexical Analyzer**
- **Syntax Analyzer**
- **Semantics Analyzer**
- **Intermediate Code Generator**
- **Code Optimizer**
- **Target Code Generator**



Lexical Analyzer

Lexical Analysis





Lexical Analysis

Example:

Comments in C & C++

// some thing

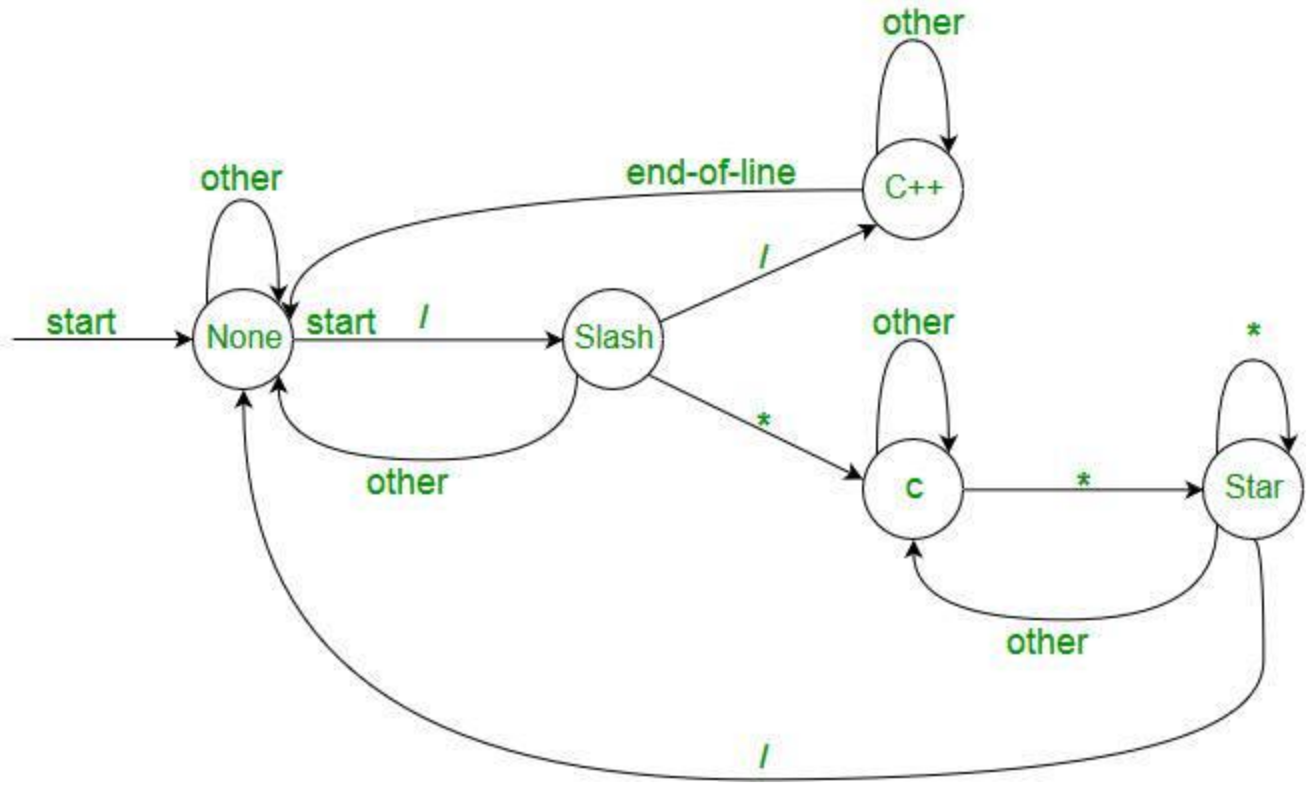
/*
* some thing
 */

Rule

comment: single | multi
single: SPLASH SPLASH
multi: SPLASH STAR NEWLINE STAR SPLASH

Tokens

SPLASH: '/'
NONE: ~[/ *]
STAR: ' * '
NEWLINE: [\n \r]+



Lexical Analysis

Token	Informal Description	Sample Lexemes
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but `"`, surrounded by `"`'s	`"core dumped"`

Figure 3.2: Examples of tokens

Lexical Analysis

Example:

```
int main()
{
  // 2 variables
  int a, b;
  a = 10;
 return 0;
}
```

```
'int'  'main'  '('  ')'  '{'  'int'  'a' ','  'b'  ';'
 'a'  '='  '10'  ';' 'return'  '0'  ';'  '}'
```

Syntax Analyzer

Syntax Analysis

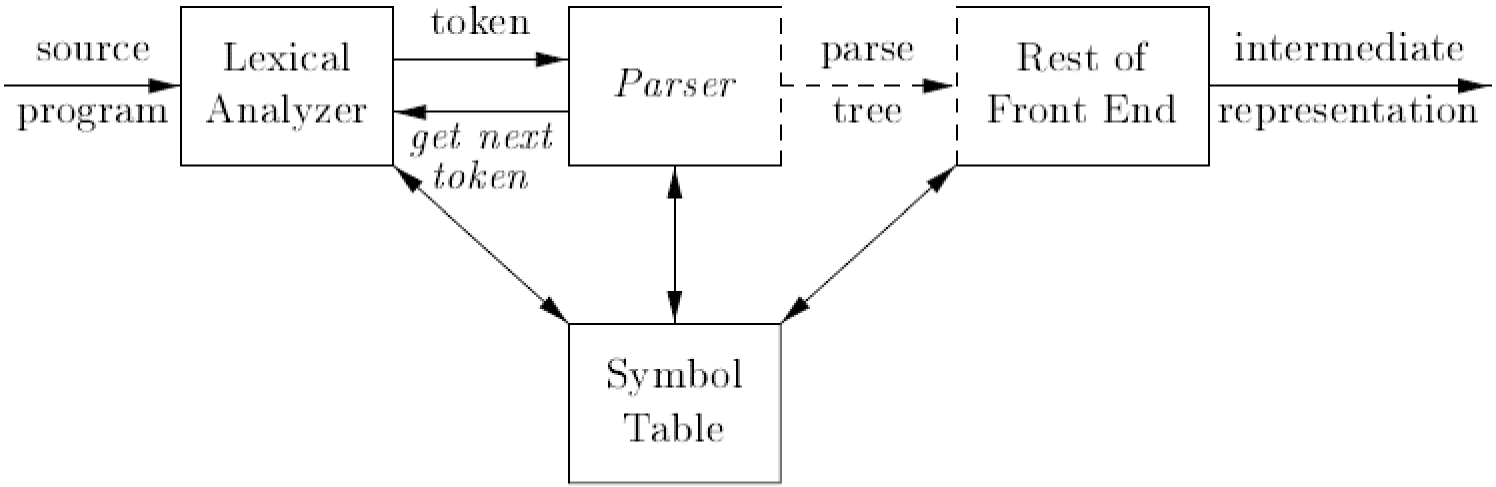


Figure 4.1: Position of parser in compiler model

Syntax Analysis

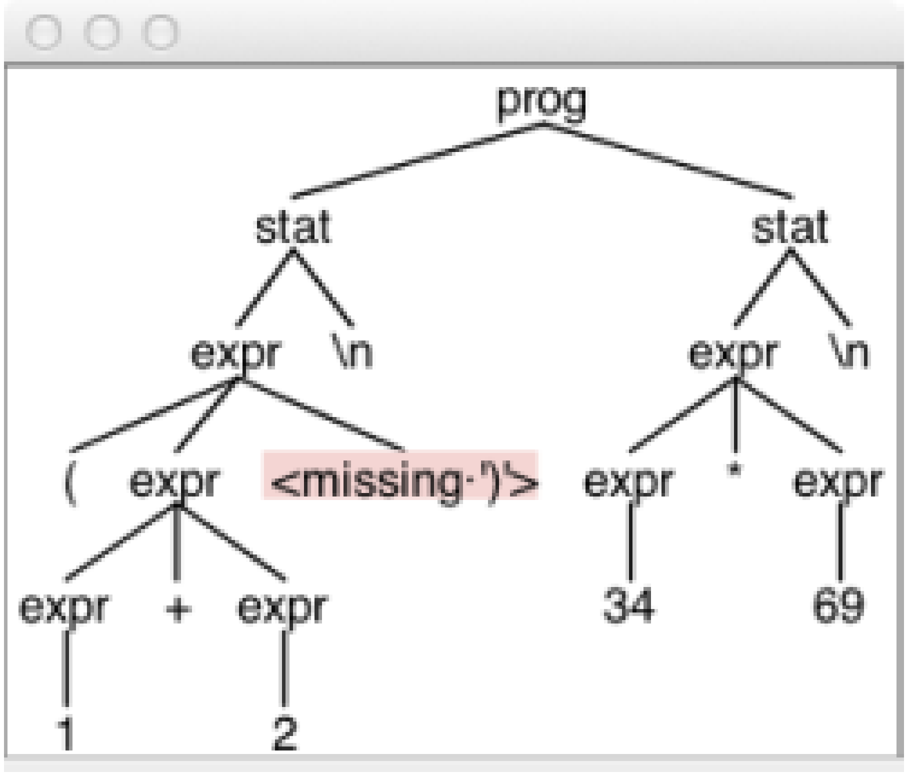
```
stat:   expr NEWLINE
    |   ID '=' expr NEWLINE
    |   NEWLINE
    ;

expr:   expr ('*'|'/') expr
    |   expr ('+'|'-') expr
    |   INT
    |   ID
    |   '(' expr ')'
    ;
```

```
ID  :   [a-zA-Z]+ ;      /.
INT :   [0-9]+ ;         /.
NEWLINE:'\r'? '\n' ;     /.
WS  :   [ \t]+ -> skip ; /.
```

Syntax Analysis

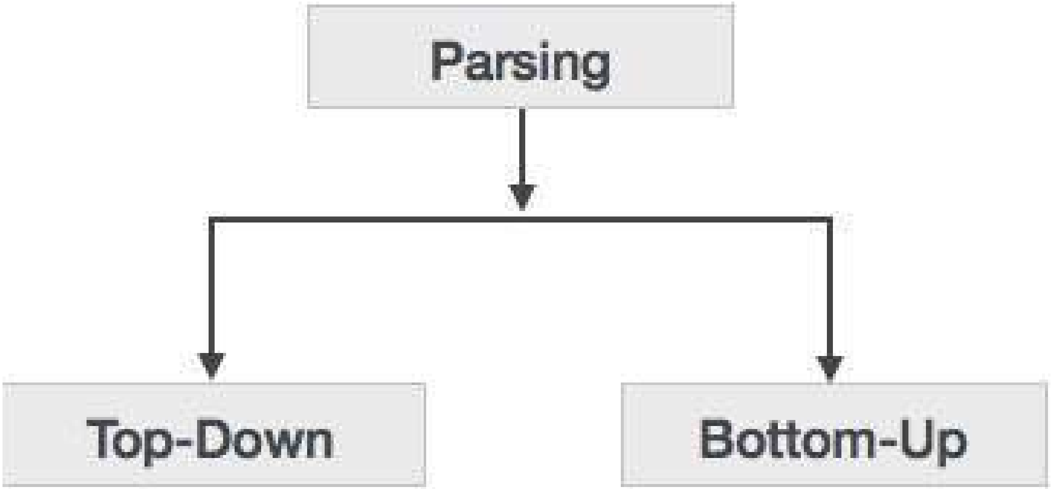
```
⇒ $ grun LibExpr prog -gui
⇒ (1+2
⇒ 34*69
⇒ EOF
```



Syntax Analysis

Type Of Parsing:

- ✓ Top-down Parser
- ✓ Bottom-up Parser



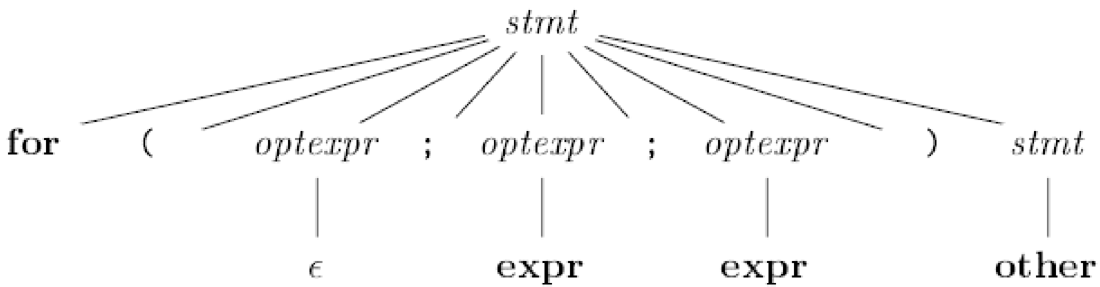
Top-down Parser:

- ✓ Recursive-descent Parser
- ✓ Non Recursive-descent Parser

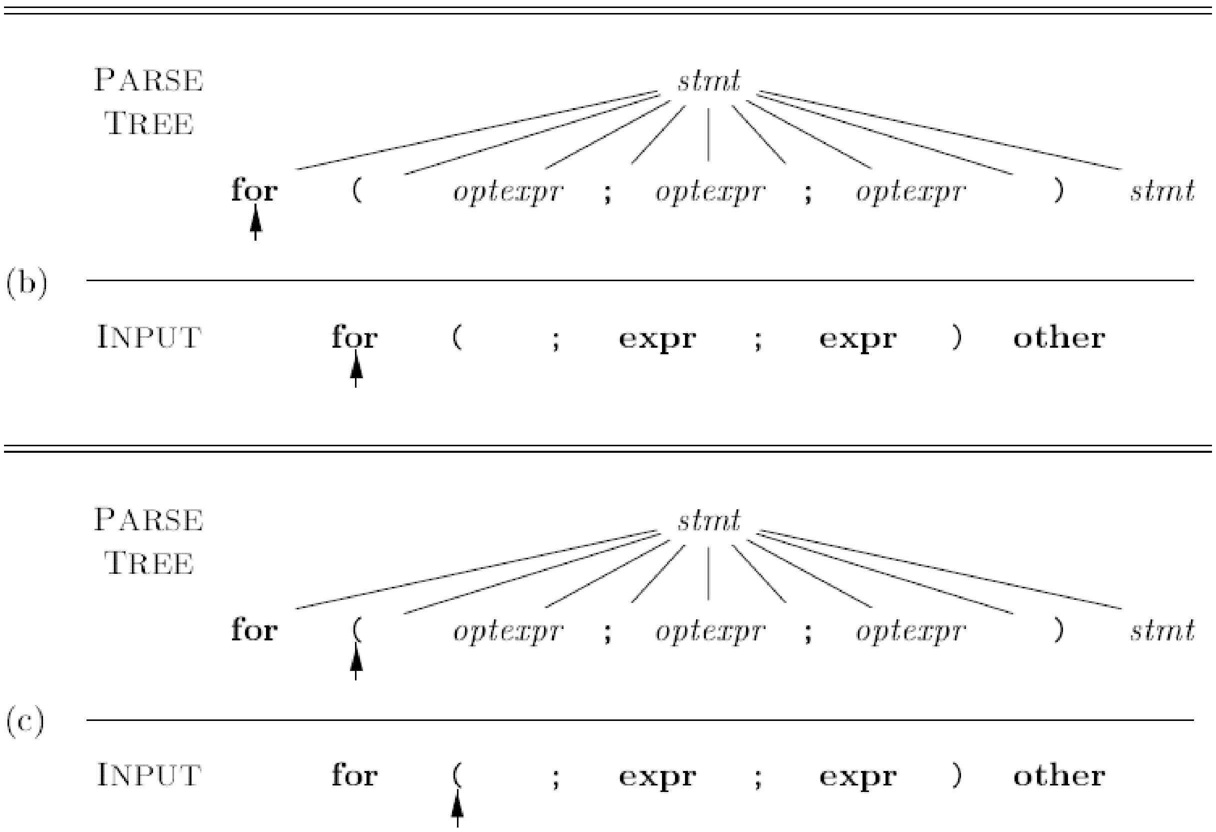
Syntax Analysis

$$\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \textbf{expr} \; ; \\
 & | & \textbf{if} \; (\; \textbf{expr} \;) \; \mathit{stmt} \\
 & | & \textbf{for} \; (\; \mathit{optexpr} \; ; \; \mathit{optexpr} \; ; \; \mathit{optexpr} \;) \; \mathit{stmt} \\
 & | & \textbf{other} \\
 & & \\
\mathit{optexpr} & \rightarrow & \epsilon \\
 & | & \textbf{expr}
\end{array}$$

Figure 2.16: A grammar for some statements in C and Java



Syntax Analysis



Syntax Analysis

Implementation:

```
assign : ID '=' expr ';' ; // match an assignment statement like "sp = 100;"
```

```
// assign : ID '=' expr ';' ;
void assign() {      // method generated from rule assign
    match(ID);       // compare ID to current input symbol then consu
    match('=');
    expr();          // match an expression by calling expr()
    match(';');
}
```

Syntax Analysis

Implementation:

```
/** Match any kind of statement starting at the current input position */
stat: assign          // First alternative ('|' is alternative separator)
    | ifstat          // Second alternative
    | whilestat
   ...
    ;
```

A parsing rule for stat looks like a switch.

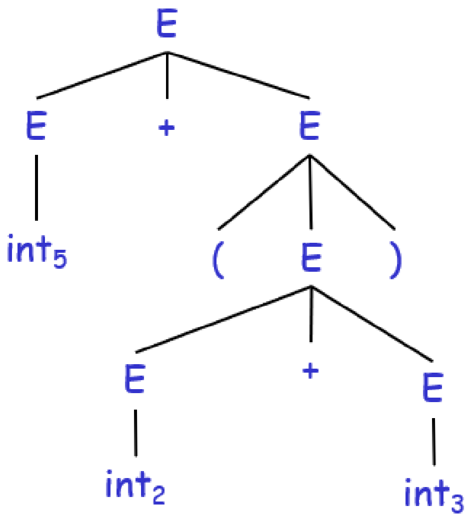
```
void stat() {
    switch ( «current input token» ) {
        CASE ID    : assign(); break;
        CASE IF    : ifstat(); break; // IF is token type for keyword 'if'
        CASE WHILE : whilestat(); break;
        ...
        default    : «raise no viable alternative exception»
    }
}
```

Parse Tree

Parse Tree

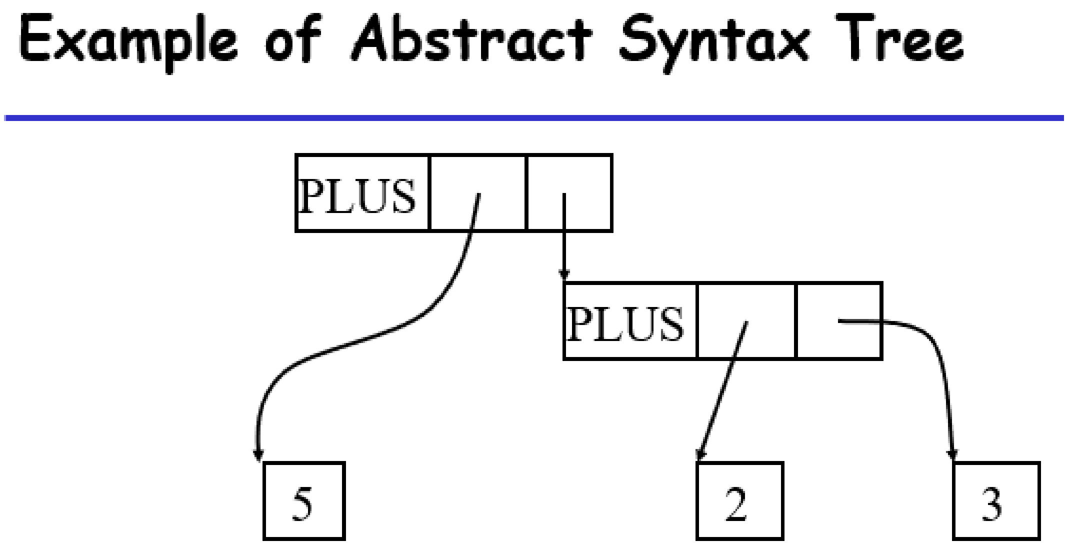
Syntax tree or parse tree This represents the structure of the sentence where each subtree root gives an abstract name to the elements beneath it. The subtree roots correspond to grammar rule names. The leaves of the tree are symbols or tokens of the sentence.

Example of Parse Tree

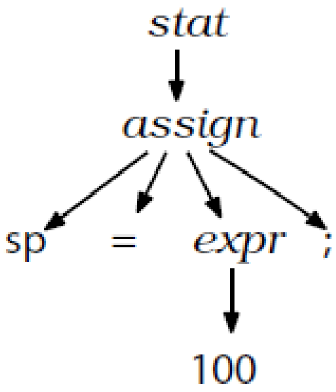


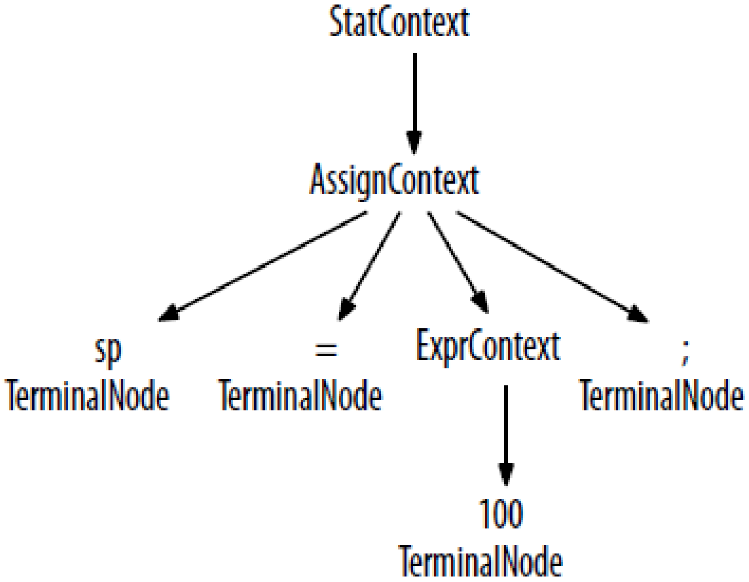
Parse Tree

Abstract Syntax Tree is a kind of tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code.



Parse Tree





These are called *context* objects because they record everything we know about the recognition of a phrase by a rule. Each context object knows the start and stop tokens for the recognized phrase and provides access to all of the elements of that phrase. For example, AssignContext provides methods ID()

Parse Tree

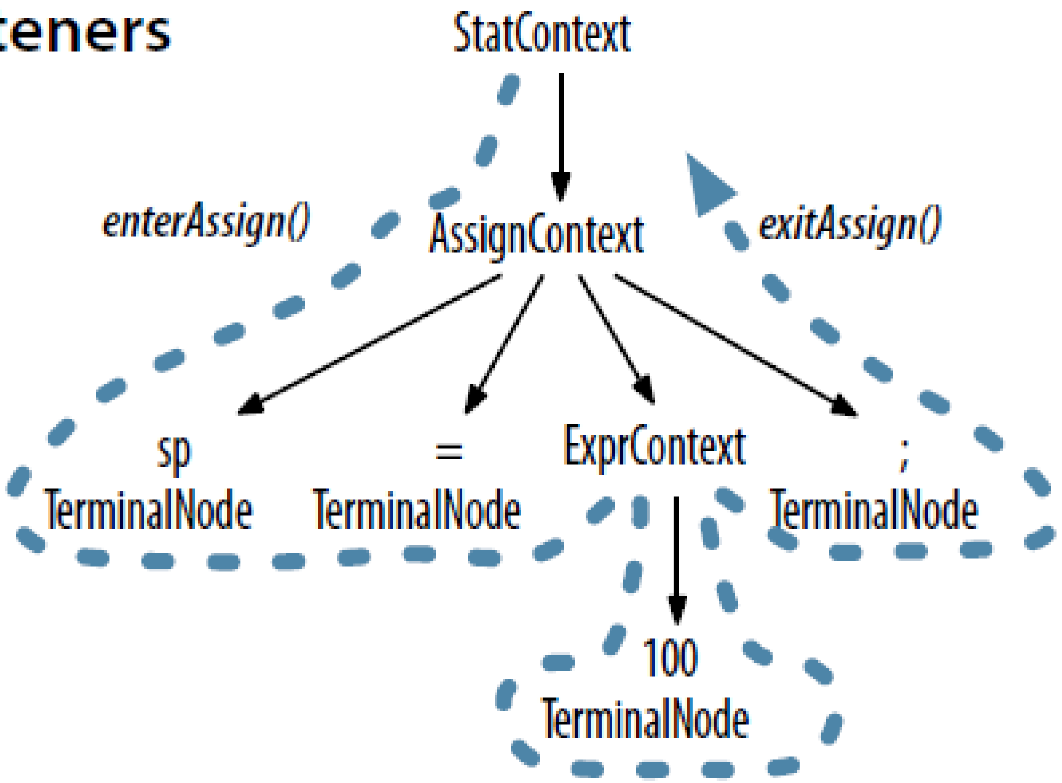
Parse-Tree Listeners and Visitors

ANTLR provides support for two tree-walking mechanisms in its runtime library. By default, ANTLR generates a parse-tree *listener* interface that responds to events triggered by the built-in tree walker. The listeners themselves are exactly like SAX document handler objects for XML parsers. SAX listeners receive notification of events like startDocument() and endDocument(). The

ANTLR generates a ParseTreeListener subclass specific to each grammar with enter and exit methods for each rule. As the walker encounters the node for rule assign, for example, it triggers enterAssign() and passes it the AssignContext parse-tree node. After the walker visits all children of the assign node, it triggers exitAssign(). The tree diagram shown below shows ParseTreeWalker performing a depth-first walk, represented by the thick dashed line.

Parse Tree

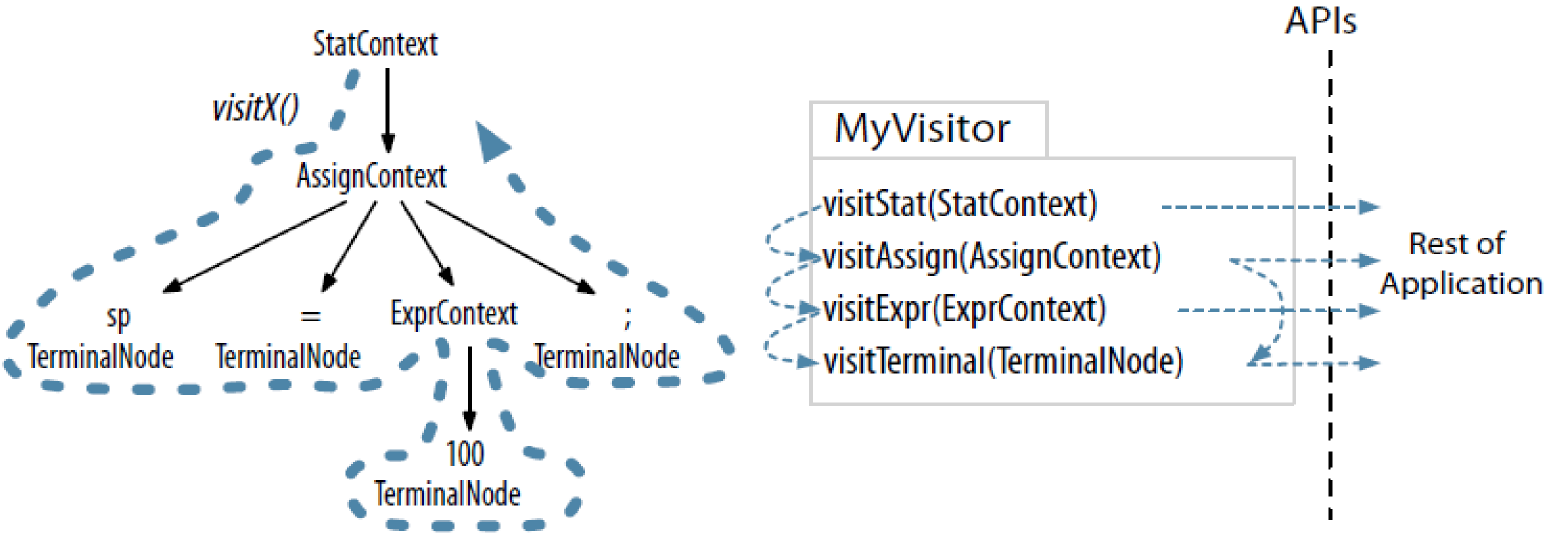
Parse-Tree Listeners



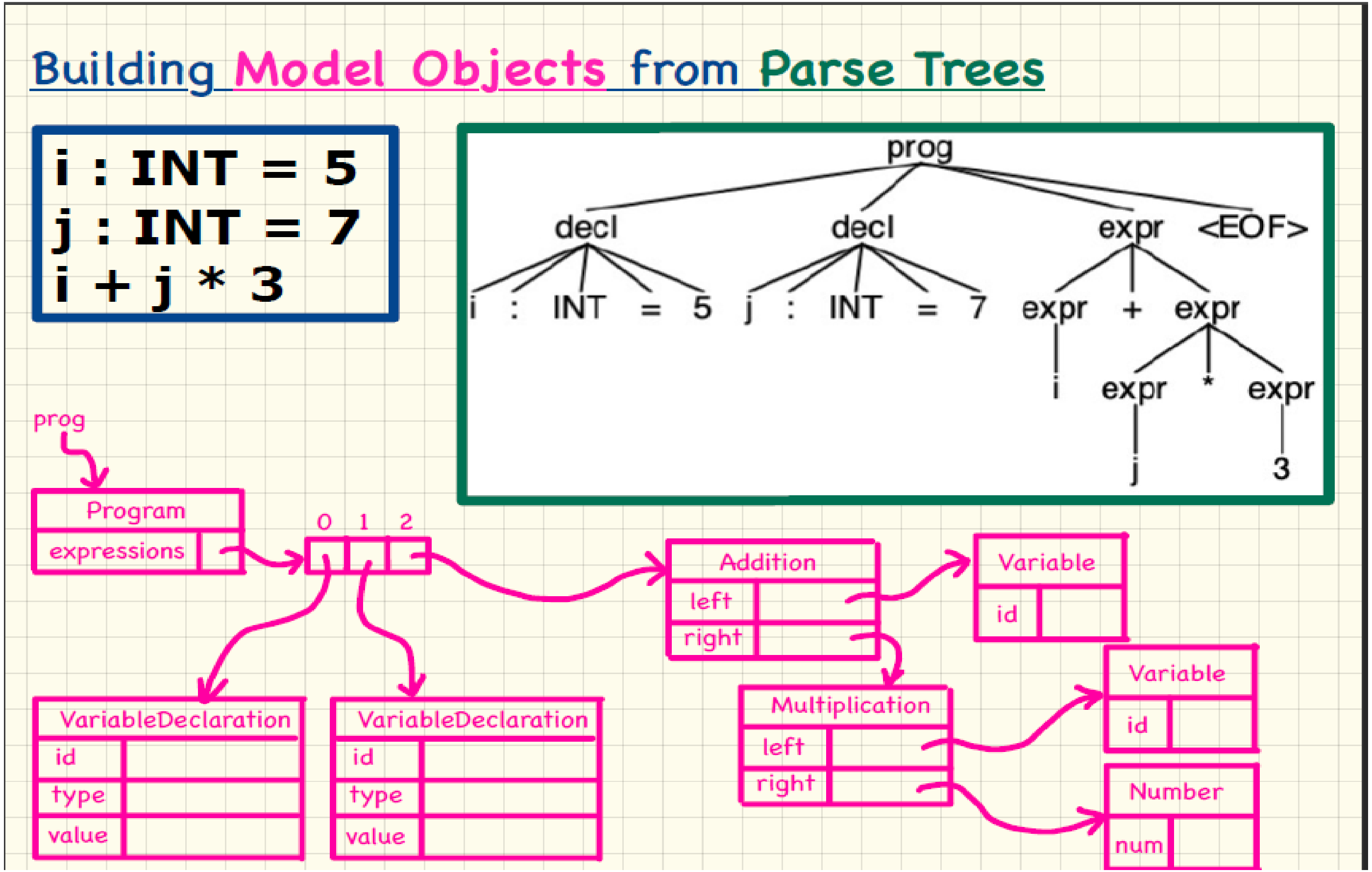
Parse Tree

Parse-Tree Visitors

There are situations, however, where we want to control the walk itself, explicitly calling methods to visit children. Option -visitor asks ANTLR to generate a visitor interface from a grammar with a visit method per rule. Here's the familiar visitor pattern operating on our parse tree:



Parse Tree



Building Model Objects from Parse Trees
i : INT = 5
j : INT = 7
i + j * 3
prog
decl
decl
expr
<EOF>
i : INT = 5
j : INT = 7
expr + expr
i
expr * expr
j
3
prog
Program
expressions
0 1 2
Addition
left
right
Variable
id
Variable
id
Multiplication
left
right
Number
num
VariableDeclaration
id
type
value
VariableDeclaration
id
type
value

Symbol Table

```
int count;
char x[] = "NESO ACADEMY";
```

Entries

Name	Type	Size	Dimension	Line of Declaration	Line of Usage	Address
count	int	2	0	--	--	--
x	char	12	1	--	--	--

03 Compiler Design

Symbol Table

Operation	Function
allocate	to allocate a new empty symbol table
free	to remove all entries and free storage of symbol table
lookup	to search for a name and return pointer to its entry
insert	to insert a name in a symbol table and return a pointer to its entry
set_attribute	to associate an attribute with a given entry
get_attribute	to get an attribute associated with a given entry

Thank you