Compiler Design

Reference Books:

1. Compilers - Principles, Techniques, & Tools, 2nd edition, by Aho Ullman

This PDF contains the notes from the standards books and are only meant for GATE CSE aspirants.

Notes Compiled By-
Manu Thakur
Mtech CSE, IIT Delhi
worstguymanu@gmail.com
https://www.facebook.com/Worstguymanu

Compilers

A **compiler** is a program that can read a program in one language - the **source language** - and translate it into an equivalent program in another language - the **target language**;

Note - An important role of the compiler is **to report any errors in the source program** that it detects during the translation process.

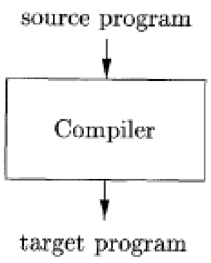


Figure 1.1: A compiler

If the target program is an **executable machine-language program**, it can then be called by the user to process **inputs** and produce **outputs**.

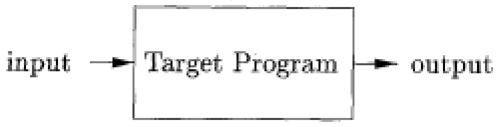


Figure 1.2: Running the target program

Interpreter
An interpreter is another common kind of **language processor**. Instead of producing a target program as a translation, an interpreter appears to **directly execute the operations** specified in the source program on inputs supplied by the user.

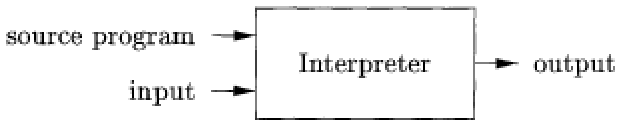


Figure 1.3: An interpreter

Note –
1. The machine-language target program produced by a **compiler** is usually **much faster** than an interpreter at **mapping inputs to outputs**.
2. An **interpreter**, however, can usually **give better error diagnostics** than a compiler, because it executes the source program **statement by statement**.

Hybrid Compiler
Java language processors combine compilation and interpretation, as shown below

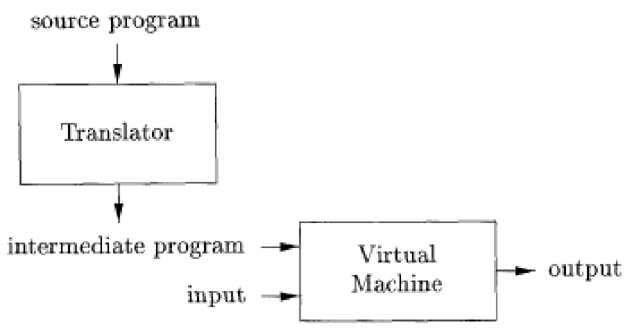


Figure 1.4: A hybrid compiler

A Java source program may first be compiled into an **intermediate form** called **bytecodes**. The bytecodes are then interpreted by a **virtual machine**. A benefit of this arrangement is that **bytecodes compiled on one machine can be interpreted on another machine**.

Language Processing System

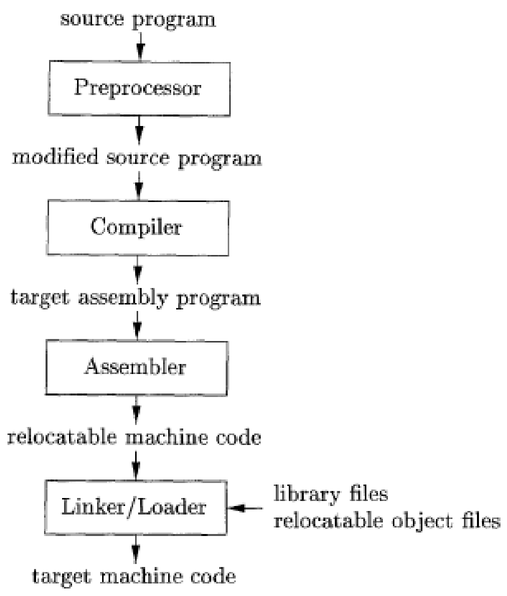


Figure 1.5: A language-processing system

1. A **source program** may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a separate program, called a **preprocessor**.
2. The **preprocessor** may also expand shorthands, **called macros**, into source language Statements.
3. The modified source program is then fed to a **compiler**.
4. The **compiler** may produce an **assembly-language program** as its output because **assembly language is easier to produce as output and is easier to debug**.
5. The assembly language is then processed by a program called an **assembler** that produces **relocatable machine code** as its output.
6. Large programs are often compiled in pieces, so the re**locatable machine code may have to be linked together with other relocatable object files and library files into the code** that actually runs on the machine.
7. The **linker** resolves external memory addresses, where the code in one file may refer to a location in another file.
8. The **loader** then puts together all of the executable object files into memory for execution.

The Structure of a Compiler

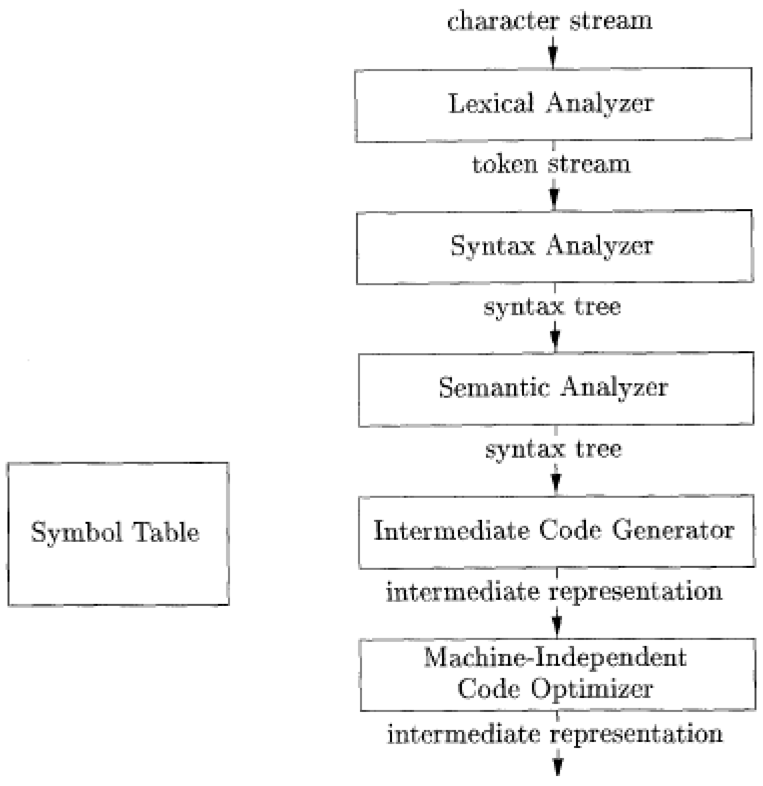
Up to this point we have treated a compiler as a single box that maps a source program into a semantically equivalent target program. If we open up this box a little, we see that there are two parts to this mapping: **analysis** and **synthesis**.

The analysis part

1. It breaks up the source program into **pieces** and **imposes a grammatical structure** on them.
2. It then uses this structure to create an **intermediate representation** of the source program.
3. If the analysis part detects that the source program is either **syntactically ill** formed or **semantically unsound**, then it must provide **informative messages**, so the user can take **corrective action**.
4. The analysis part also collects information about the source program and stores it in a data structure called a **symbol table**, which is passed along with the intermediate representation to the **synthesis part**.
5. The analysis part is often called the **front end** of the compiler.

The synthesis part

1. It constructs the desired **target program** from the **intermediate representation** and the information in the **symbol table**.
2. The synthesis part is the **back end**.


character stream
Lexical Analyzer
token stream
Syntax Analyzer
syntax tree
Semantic Analyzer
syntax tree
Intermediate Code Generator
intermediate representation
Machine-Independent Code Optimizer
intermediate representation
Symbol Table

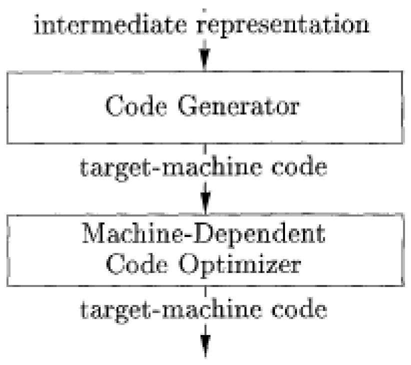


Figure 1.6: Phases of a compiler

Note –
1. **The symbol table, which stores information about the entire source program, is used by all "phases of the compiler"**.
2. Some compilers have a **machine-independent optimization** phase between the front end and the back end.

Lexical Analysis
The first phase of a compiler is called **lexical analysis** or **scanning**. The **lexical analyzer** reads the **stream of characters** making up the source program and groups the characters into meaningful sequences called **lexemes**.

For each lexeme, the lexical analyzer produces as output a **token** of the form

$$\langle \textit{token-name}, \textit{attribute-value} \rangle$$

That it passes on to the subsequent phase, syntax analysis. In the token, the first component token-name is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token.

For example, suppose a source program contains the assignment statement

position = initial + rate * 60

The characters in this assignment could be grouped into the following **lexemes** and mapped into the following **tokens** passed on to the **syntax analyzer**:

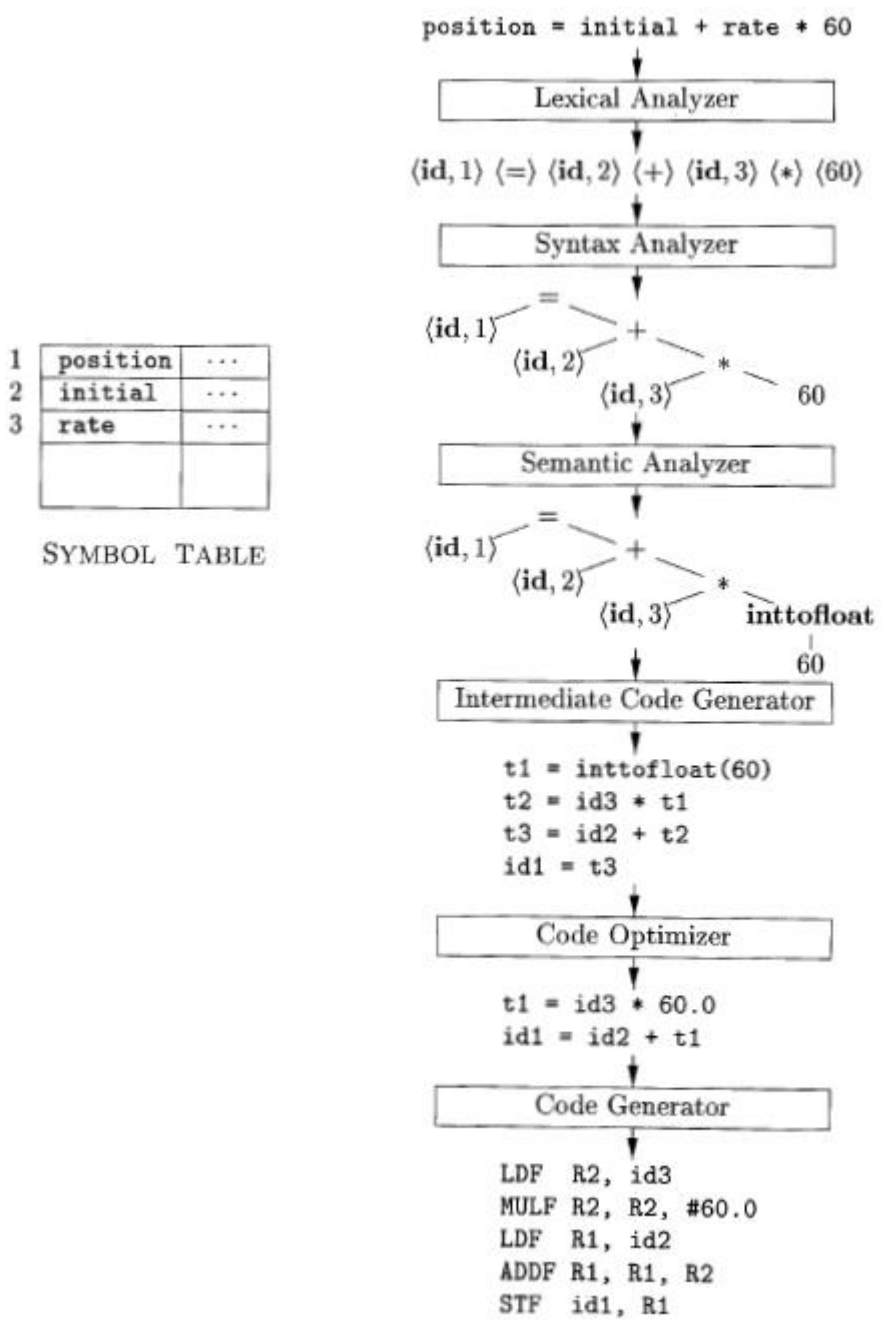
1. Position is a **lexeme** that would be mapped into a token **(id, 1)**, where **id** is an **abstract symbol** standing for **identifier** and 1 points to the symbol **table entry** for position.
2. The assignment symbol = is a lexeme that is mapped into the token (=). Since this token needs no attribute-value, we have omitted the second component. We could have used any abstract symbol such as **assign** for the token-name, but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.
3. **initial** is a lexeme that is mapped into the token **(id, 2)**, where 2 points to the symbol-table entry for initial .

4. + is a **lexeme** that is mapped into the token (+).
5. **rate** is a **lexeme** that is mapped into the token **(id, 3)**, where 3 points to the symbol-table entry for rate .
6. * is a lexeme that is mapped into the token (*).
7. 60 is a lexeme that is mapped into the token (60)

Note - Blanks separating the lexemes would be discarded by the lexical analyzer.

Lexical analysis as the sequence of tokens

$$\langle \mathbf{id},1\rangle \ \langle=\rangle \ \langle \mathbf{id},2\rangle \ \langle+\rangle \ \langle \mathbf{id},3\rangle \ \langle*\rangle \ \langle 60\rangle$$



```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

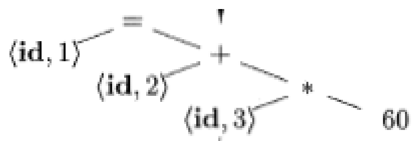
Figure 1.7: Translation of an assignment statement

Syntax Analysis

The second phase of the compiler is **syntax analysis** or **parsing**. The parser uses the **first components** of the tokens produced by the lexical analyzer to create a tree-like **intermediate representation** that depicts the grammatical structure of the token stream.
A typical representation is a syntax tree in which each **interior node represents an operation** and the **children of the node represent the arguments of the operation**.

This tree shows the **order** in which the operations in the assignment are to be performed.

```
position = initial + rate * 60
```



The tree has an interior node labeled * with **(id, 3)** as its left child and the integer 60 as its right child. The node (id, 3) represents the identifier **rate**. The node labeled * makes it explicit that we must first multiply the value of **rate** by **60**.

Semantic Analysis

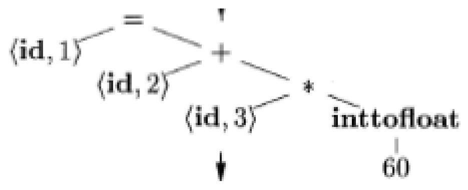
The **semantic analyzer** uses the **syntax tree** and the **information in the symbol table** to check the source program for semantic consistency with the **language definition**.

It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

An important part of semantic analysis is **type checking**, where the compiler checks that each operator has matching operands.
For example, many programming language definitions require an array index to be an **integer**; the compiler must report an error if a floating-point number is used to index an array.

Coercions

The language specification may permit some **type conversions** called **coercions**. For example a **binary arithmetic operator** may be applied to either a pair of integers or to a pair of floating-point numbers. **If the operator is applied to a floating-point number and an integer**, the compiler may **convert** or **coerce** the **integer into a floating-point number**.



Suppose that **position**, **initial**, and **rate** have been declared to be floating-point numbers, and that the lexeme 60 by itself forms an **integer**.
The type checker in the semantic analyzer in discovers that the operator * is applied to a floating-point number rate and an integer 60.
In this case, the **integer may be converted into a floating-point number**.
Notice that the output of the semantic analyzer has an extra node for the operator **inttofloat**, which explicitly converts its integer argument into a floating-point number.

Intermediate Code Generation

In the process of translating a source program into target code, a compiler may construct **one or more intermediate representations**, which can have a variety of forms.
Syntax trees are a form of intermediate representation; they are commonly used during **syntax** and **semantic** analysis.

After syntax and semantic analysis of the source program, many compilers generate an explicit **low-level** or **machine-like intermediate representation**.

We consider an intermediate form called **three-address code**, which consists of a sequence of assembly-like instructions with **three operands per instruction**. **Each operand can act like a register**.

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Each three-address assignment instruction has **at most one operator on the right side**. Thus, **these instructions fix the order in which operations are to be done**;

Code Optimization

The **machine-independent code-optimization** phase attempts to improve the **intermediate code** so that better target code will result.

A simple **intermediate code generation algorithm followed by code optimization** is a reasonable way to generate good target code.

```
t1 = id3 * 60.0
id1 = id2 + t1
```

Note – We have machine dependent optimization also.

Code Generation

The code generator takes as input an intermediate representation of the source program and maps it into the **target language**.
If the **target language is machine code**, **registers** or **memory locations** are **selected for each of the variables used by the program**. Then, the intermediate instructions are translated into sequences of **machine instructions** that perform the same task.

```
LDF  R2,  id3
MULF R2,  R2, #60.0
LDF  R1,  id2
ADDF R1,  R1, R2
STF  id1, R1
```

Symbol-Table Management

The symbol table is a **data structure** containing a **record for each variable name**, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.

Identifiers/ Names/Variables

1. An **identifier** is a string of **characters**, typically **letters** or **digits** that refers to (**identifies**) **an entity**, such as a data object, a procedure, a class, or a type.
   All identifiers are names, but not all names are identifiers.
2. Names can also be **expressions**. For example, the name ***x.y*** might denote the field *y* of a structure denoted by x. Here, x and *y* are identifiers, while *x.y* is a name, but not an identifier.
3. A **variable** **refers to a particular location of the store**. **It is common for the same identifier to be declared more than once**; **each such declaration introduces a new variable**. **Even if each identifier is declared just once, an identifier local to a recursive procedure will refer to different locations of the store at different times**.

Declarations and Definitions

1. **Declarations** tell us about the **types of things**.
2. While **definitions** tell us about **their values**.

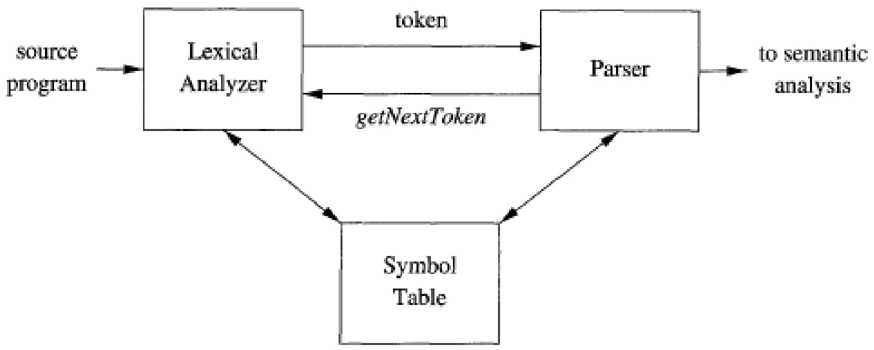
For example, **int i;** **is a declaration of i**, while **i = 1; is a definition of i**.

Lexical Analysis

The Role of the Lexical Analyzer

The main task of the lexical analyzer is to read the input characters of the source program, group them into **lexemes**, and produce as output a **sequence of tokens** for each lexeme in the source program. The **stream of tokens is sent to the parser** for syntax analysis.

1. It is common for the lexical analyzer to interact with the symbol table as well.
2. When the lexical analyzer **discovers a lexeme constituting an identifier**, it needs to enter that lexeme into the symbol table.



Interactions between the lexical analyzer and the parser

Other tasks can be performed by lexical analyser is:
1. Removing out **comments** and **whitespace** (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input).
2. Another task is correlating error messages generated by the compiler with the source program.
3. The lexical analyzer may keep track of the number of newline characters seen, so it can associate **a line number with each error message**.
4. In some compilers, the lexical analyzer makes a copy of the source program with the error messages inserted at the appropriate positions.
5. In some compilers, the **lexical analyzer makes a copy of the source program with the error messages inserted** at the appropriate positions.
6. If the source program uses a **macro-preprocessor**, the **expansion of macros** may also be performed by the lexical analyzer.

Lexical Analysis versus Parsing

There are a number of reasons why the **analysis portion of a compiler is normally separated into lexical analysis and parsing** (syntax analysis) phases.

1. The separation of lexical and syntactic analysis often allows us to **simplify** at least one of these tasks.
2. **Compiler efficiency is improved**. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing.
3. Compiler portability is enhanced. **Input-device-specific typicality** can be r**estricted to the lexical analyzer**.

<u>**Tokens, Patterns, and Lexemes**</u>

- A **token** is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit.
- A **pattern** is a <u>**description of the form**</u> that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword.
- A **lexeme** is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

<u>**Tricky Problems When Recognizing Tokens**</u>

The following example is taken from **Fortran**, in the **fixed-format** still allowed in Fortran 90. In the statement

```
DO 5 I = 1.25
```

It is not clearly understood that <u>**the first lexeme is D051**</u>, an instance of the identifier token, <u>**until we see the dot following the 1**</u>.
Note that blanks in fixed-format Fortran are ignored.

Had we <u>**seen a comma instead of the dot**</u>, we would have had a **do-statement**

```
DO 5 I = 1,25
```

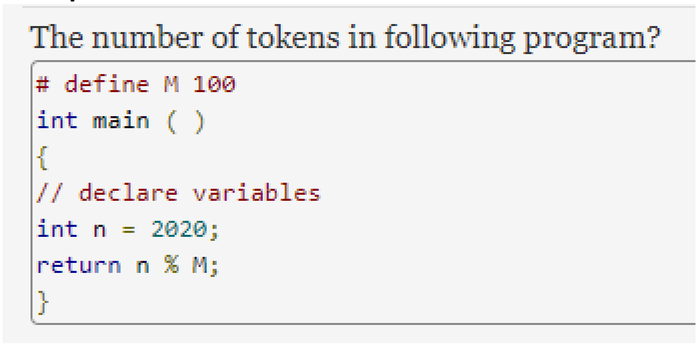
in which the <u>**first lexeme is the keyword DO**</u>.

<u>**Example from GateOverflow**</u>

<u>**Example**</u>
result = x > y ? "x is greater than y" : "x is less than or equal to y";

Solution: there are 10 tokens.

Example

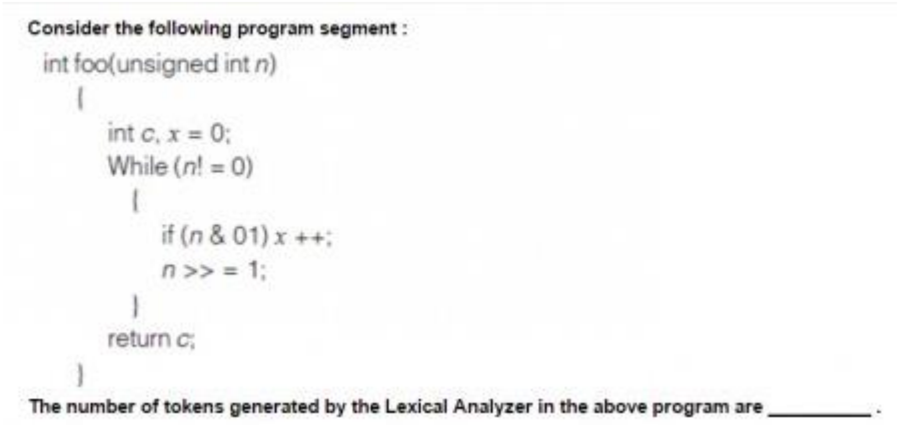

The number of tokens in following program?

```
# define M 100
int main ( )
{
// declare variables
int n = 2020;
return n % M;
}
```

Solution
Number of tokens = 16
Macros and comments are not <u>considered as tokens</u>.

Example:
printf("string %d ",++i++&&&i***a);
return(x?y:z)
Solution there are 24 tokens, *** are three tokens as ** is not a operator. &&, & two tokens.

Example



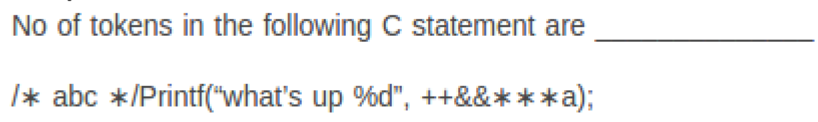
Consider the following program segment :

```
int foo(unsigned int n)
  {
     int c, x = 0;
     While (n! = 0)
       {
         if (n & 01) x ++;
         n >> = 1;
       }
     return c;
  }
```

The number of tokens generated by the Lexical Analyzer in the above program are __________ .

Solution
>>=, !=, ++ will be considered as 1 – 1 token.
Total number of tokens = 40

Example



No of tokens in the following C statement are _______________

/∗ abc ∗/Printf("what's up %d", ++&&∗∗∗a);

Solution: 12

<u>**Examples**</u> Which one is lexical error?

1. ; a = b + c **No**
2. c = a + "Hello" **No**
3. int a = 1, b=2; **No**
4. String S ="Hello **Yes**
5. char a = 'ab **Yes**
6. char b = ab' **Yes**
7. String J = compiler" **Yes**
8. int a = 908; **No**
9. float c = 9.7; **No**
10. float d = 9…8; **Yes**
11. int1 a, b; **No**
12. int 1a, b; **Yes**
13. 1int a, b; **Yes**
14. int a, b; **No**

Syntax Analysis

The Role of the Parser

The parser obtains a string of tokens from the lexical analyzer, and verifies that the string of token names can be generated by the grammar for the source language.

The parser **constructs a parse tree and passes it to the rest of the compiler for further processing**.

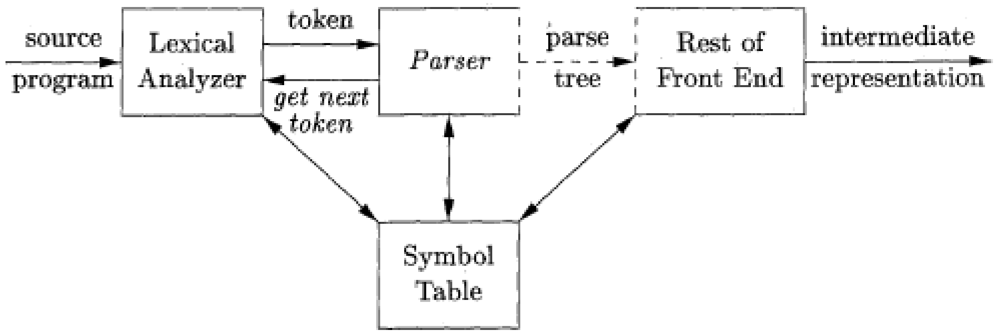


Figure 4.1: Position of parser in compiler model

There are three general types of parsers for grammars:
1. Universal
2. Top-down
3. Bottom-up.

1. **Universal parsing** methods such as the **CYK algorithm** and **Earley's algorithm** can parse any grammar. These general methods are, however, **too “inefficient” to use in production compilers**.
2. **Top-down** methods build parse trees from the **top (root) to the bottom (leaves)**.
3. **Bottom-up methods** start from the **leaves** and work their way up **to the root**.

Note - In either case TDP or BUP, the **input to the parser is scanned from left to right**, one symbol at a time.
1. Lack of tokens to frame a statement.
2. More no. of tokens than actually required for a statement.

Syntax Error Handling

A compiler is expected to assist the programmer in locating and **tracking down** errors.
Common programming errors can occur at many different levels.

- **Lexical errors** include misspellings of identifiers, keywords, or operators - e.g., the use of an identifier elipsesize instead of ellipsesize – and missing quotes around text intended as a string.

- **Syntactic errors** include **misplaced semicolons or extra or missing braces**; that is, '{" or "}" As another example, in C or Java, the appearance of a case statement without an enclosing switch is a syntactic error.

- **Semantic errors** include **type mismatches** between operators and operands. An example is a return statement in a Java method with result type void.

- **Logical errors** can be anything from **incorrect reasoning** on the part of the programmer to the use in a C program of the assignment operator = instead of the comparison operator ==. The program containing = may be well formed; however, it may not reflect the programmer's intent.

Derivations

Beginning with the **start symbol**, **each rewriting step replaces a nonterminal by the body of one of its productions**. This derivational view corresponds to the **top-down construction of a parse tree**.

For example, consider the following grammar, with a single nonterminal E

$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \mathbf{id}$$

We can take a single E and repeatedly apply productions in any order to get a sequence of replacements. For example

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$$

We call such a sequence of replacements a **derivation of -(id) from E**.

1. The symbol $\Rightarrow$ means "**derives in one step**".
2. The symbol $\overset{*}{\Rightarrow}$ means "**derives in zero or more steps**".
3. The symbol $\overset{+}{\Rightarrow}$ means "**derives in one or more steps**".

Sentential form of G

If $S \overset{*}{\Rightarrow} \alpha$ where **S** is the start symbol of a grammar G, we say that α is a **sentential** form of G. Note that a sentential form may contain both **terminals** and **nonterminals**, and may be **empty**.

Sentence of G

A **sentence** of G is a sentential form with **no nonterminals**. The **language generated by a grammar** is its **set of sentences**. Thus, a string of terminals w is in **L(G)**, the language generated by G, if and only if **w is a sentence of G** G (or $S \overset{*}{\Rightarrow} w$).

The string **-(id + id) is a sentence** of the grammar because there is a derivation

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$$

The strings $E, -E, -(E), \ldots, -(\mathbf{id}+\mathbf{id})$ are all sentential forms of this grammar. We write $E \overset{*}{\Rightarrow} -(\mathbf{id}+\mathbf{id})$ to indicate that $-(\mathbf{id}+\mathbf{id})$ can be derived from E.

Leftmost Derivations

In **leftmost derivations**, the **leftmost nonterminal in each sentential** is always chosen. If $\alpha \Rightarrow \beta$ is a step in which the leftmost nonterminal in α is replaced, we write $\alpha \underset{lm}{\Rightarrow} \beta$

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+\mathbf{id})$$

Example of leftmost derivations

Rightmost Derivations

In **rightmost derivations**, the **rightmost nonterminal** is always chosen; we write $\alpha \underset{rm}{\Rightarrow} \beta$ in this case.

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+\mathbf{id}) \Rightarrow -(\mathbf{id}+\mathbf{id})$$

Example of rightmost derivations

Note - Rightmost derivations are sometimes called canonical derivations.

Parse Trees and Derivations

A **parse tree** is a **graphical representation of a derivation** that **filters out the order** in which productions are applied to replace nonterminals.
The interior node is labeled with the **nonterminal A** in the head of the production; **the children of the node are labeled, from left to right, by the symbols in the body of the production by which this A was replaced during the derivation**.

The **leaves** of a parse tree are labeled by **nonterminals or terminals** and, read from **left to right**, constitute a **sentential form**, called the **yield** or **frontier of the tree**.

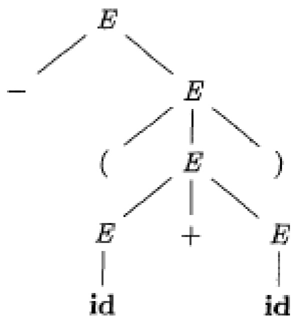


Figure 4.3: Parse tree for $-(\mathbf{id}+\mathbf{id})$

Note - Every parse tree has associated with it a unique leftmost and a unique rightmost derivation.

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id}) \qquad (4.8)$$ **(LMD)**

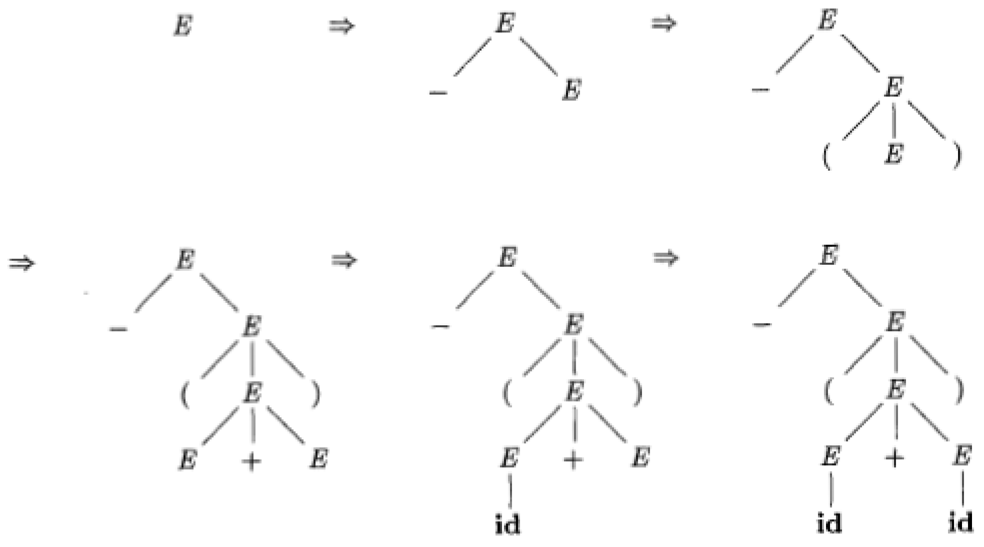


Figure 4.4: Sequence of parse trees for derivation (4.8)

<u>Ambiguity</u>

1. A grammar that produces **<u>more than one parse tree</u>** for <u>some sentence</u> is said to be **ambiguous**.
2. An ambiguous grammar is one that produces **<u>more than one leftmost derivation</u>** or **<u>more than one rightmost derivation</u>** for the same sentence.

The following arithmetic expression grammar permits **<u>two distinct leftmost derivations</u>** for the sentence **id + id * id**.

$$
\begin{array}{lll}
E & \Rightarrow & E + E \\
  & \Rightarrow & \mathbf{id} + E \\
  & \Rightarrow & \mathbf{id} + E * E \\
  & \Rightarrow & \mathbf{id} + \mathbf{id} * E \\
  & \Rightarrow & \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{array}
\qquad
\begin{array}{lll}
E & \Rightarrow & E * E \\
  & \Rightarrow & E + E * E \\
  & \Rightarrow & \mathbf{id} + E * E \\
  & \Rightarrow & \mathbf{id} + \mathbf{id} * E \\
  & \Rightarrow & \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{array}
$$

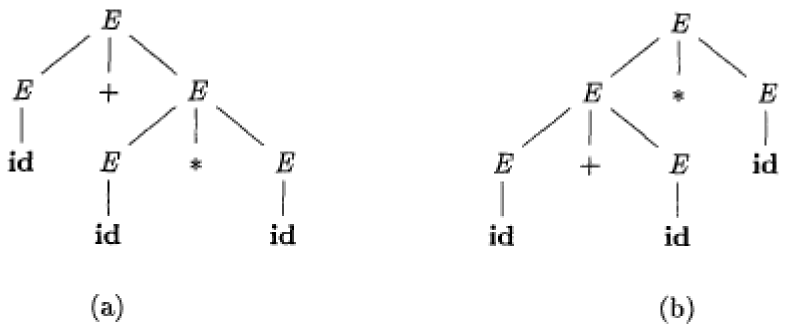


Figure 4.5: Two parse trees for **id+id*id**

Note that the parse tree of Fig. (a) reflects the **commonly assumed precedence of + and *,** while the tree of Fig. (b) **does not**.

<u>Lexical Versus Syntactic Analysis</u>

"**Why use regular expressions to define the lexical syntax of a language?**"
There are several reasons.

1. The **<u>lexical rules of a language are frequently quite simple</u>**, and to describe them we do not need a notation as powerful as grammars.
2. Regular expressions generally provide a more **concise** and **easier-to-understand** notation for tokens than grammars.
3. More efficient lexical analyzers can be **<u>constructed automatically from regular expressions</u>** than from arbitrary grammars.

Grammars, on the other hand, are most useful for describing **<u>nested structures</u>** such as **balanced parentheses**, matching **begin-end's**, corresponding if-then-else's, and so on. **<u>These nested structures cannot be described by regular expressions</u>**.

Eliminating Ambiguity

"Sometimes" an ambiguous grammar can be rewritten to eliminate the ambiguity

Dangling Else Problem

We shall eliminate the ambiguity from the following "**dangling else**" grammar:

$$\begin{array}{rcl} stmt & \rightarrow & \textbf{if}\ expr\ \textbf{then}\ stmt \\ & | & \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt \\ & | & \textbf{other} \end{array}$$

Here "**other**" stands for any other statement. According to this grammar, the compound conditional statement

$$\textbf{if}\ E_1\ \textbf{then}\ S_1\ \textbf{else if}\ E_2\ \textbf{then}\ S_2\ \textbf{else}\ S_3$$

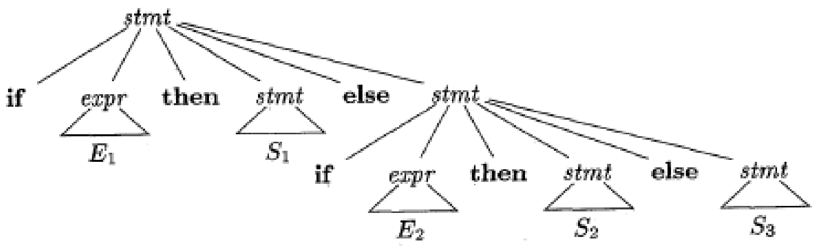


Figure 4.8: Parse tree for a conditional statement

has the above parse tree. But this grammar is ambiguous since the string

$$\textbf{if}\ E_1\ \textbf{then if}\ E_2\ \textbf{then}\ S_1\ \textbf{else}\ S_2$$

has the two parse trees as shown

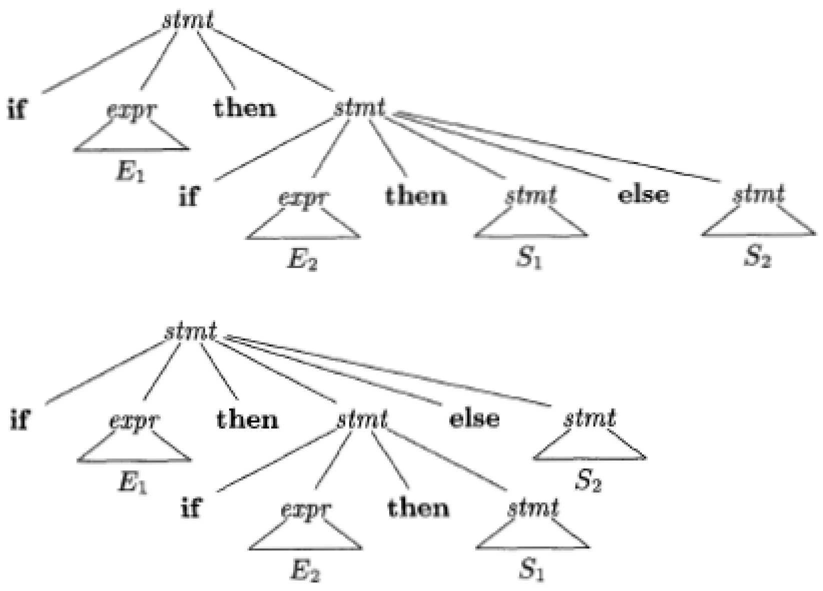


Figure 4.9: Two parse trees for an ambiguous sentence

$$
\begin{array}{rcl}
stmt & \rightarrow & matched\_stmt \\
 & | & open\_stmt \\
matched\_stmt & \rightarrow & \textbf{if } expr \textbf{ then } matched\_stmt \textbf{ else } matched\_stmt \\
 & | & \textbf{other} \\
open\_stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } matched\_stmt \textbf{ else } open\_stmt
\end{array}
$$

Figure 4.10: Unambiguous grammar for if-then-else statements

Elimination of Left Recursion

Where **the leftmost symbol of the body** is the same as the **nonterminal at the head** of the production.

Head of production → body of production

Suppose the procedure for expr decides to apply this production.

1. The body begins with expr so the procedure for expr is called **recursively**.
2. Since the look-ahead symbol changes only when a terminal in the body is matched.
3. No change to the input took place between recursive calls of expr.
4. As a result, the **second call to expr does exactly what the first call did**, which means a third call to expr, and so on, **forever**.

A **left-recursive production** can be eliminated by **rewriting** the offending production.

$A \rightarrow A\alpha \mid \beta$ **(Immediate left recursion)**

It is possible for a **recursive-descent parser to loop forever**. A problem arises with "**left**-**recursive**" productions like

$$expr \rightarrow expr + term$$

A grammar is **left recursive** if it has a **nonterminal A** such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α. **(Indirect left Recursion)**

Removal A left recursive grammar is given as

$$A \rightarrow A\alpha \mid \beta$$

We can rewrite it as

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \epsilon$$

Example remove the left recursion from the following grammar

$$E \rightarrow E + E \mid E * E \mid (E) \mid \textbf{id}$$

Solution

E → (E) E' | id E'
E' → +E E' | *E E' | eps

Example remove the left recursion from the following grammar

$$\begin{array}{lcl} E & \to & E + T \mid T \\ T & \to & T * F \mid F \\ F & \to & (E) \mid \mathbf{id} \end{array}$$

Solution

E→ T E'
E' → +T E' | eps
T → F T'
T' → *F T' | eps
F → (E) | id

is obtained by **eliminating immediate left recursion** from the expression grammar.

Removal of Immediate left Recursion

Immediate left recursion can be **eliminated** by the following technique, which works for any number of A-productions. First, **group the productions as**

$$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

where no β_i begins with an A. Then, replace the A-productions by

$$\begin{array}{l} A \to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\ A' \to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon \end{array}$$

The nonterminal A generates the same strings as before but is **no longer left recursive**.

Note - It does not eliminate left recursion involving derivations of "two or more steps".

For example, consider the grammar

$$\begin{array}{l} S \to A\ a \mid b \\ A \to A\ c \mid S\ d \mid \epsilon \end{array}$$

The **nonterminal S is left recursive** because $S \Rightarrow Aa \Rightarrow Sda$ but it is **not immediately left recursive**.

We have an algorithm that systematically eliminates left recursion from a grammar. **It is guaranteed to work** if the **grammar has no cycles** (derivations of the form $A \overset{+}{\Rightarrow} A$) or **ϵ -productions** (productions of the form A → **ϵ**).

Note – Cycles can be eliminated systematically as can ϵ -productions.(I assume here that they are talking about to convert a CFG into CNF or GNF).

Example Remove the left recursion from the following grammar

S → Sa | S | eps

Solution

S → S'
S' → aS' | S' | eps

But we can remove S' → S', it's a trivial production, hence equivalent grammar will be

S → S'
S' → aS' | eps

Algorithm 4.19: Eliminating left recursion.

INPUT: Grammar G with no cycles or ϵ-productions.

OUTPUT: An equivalent grammar with no left recursion.

METHOD: Apply the algorithm in Fig. 4.11 to G. Note that the resulting non-left-recursive grammar may have ϵ-productions. □

```
1)  arrange the nonterminals in some order $A_1, A_2, \dots, A_n$.
2)  for ( each $i$ from 1 to $n$ ) {
3)          for ( each $j$ from 1 to $i-1$ ) {
4)                  replace each production of the form $A_i \to A_j\gamma$ by the
                        productions $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$, where
                        $A_j \to \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ are all current $A_j$-productions
5)          }
6)          eliminate the immediate left recursion among the $A_i$-productions
7)  }
```

Figure 4.11: Algorithm to eliminate left recursion from a grammar

Example Remove the indirect left recursion from the following grammar

$$\begin{aligned} S &\to A\ a \mid b \\ A &\to A\ c \mid S\ d \mid \epsilon \end{aligned}$$

Solution
We order the non-terminals as S, A.

We **<u>substitute for S in A</u>** → S d to obtain the following A-productions

$$A \to A\ c \mid A\ a\ d \mid b\ d \mid \epsilon$$

Now we can remove direct left recursion

$$\begin{aligned} S &\to A\ a \mid b \\ A &\to b\ d\ A' \mid A' \\ A' &\to c\ A' \mid a\ d\ A' \mid \epsilon \end{aligned}$$

Left Factoring

Left factoring is a grammar transformation that is useful for producing a grammar suitable for **predictive**, or **top-down, parsing**.

When the **choice between two alternative A-productions is not clear**, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice.
For example, if we have the two productions

$$\begin{array}{lll} stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\ & | & \textbf{if } expr \textbf{ then } stmt \end{array}$$

Algorithm: Left factoring a grammar
INPUT: Grammar G.
OUTPUT: An equivalent **left-factored grammar**.

METHOD: For each nonterminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$ — i.e., there is a nontrivial common prefix — replace all of the A-productions $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$, where γ represents all alternatives that do not begin with α, by

$$\begin{array}{l} A \rightarrow \alpha A' \mid \gamma \\ A' \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \end{array}$$

Here **A'** is a new nonterminal. **Repeatedly apply this transformation** until no two alternatives for a nonterminal have a common prefix.

Example The following grammar abstracts the "dangling-else" problem:

$$\begin{array}{l} S \rightarrow i\ E\ t\ S \mid i\ E\ t\ S\ e\ S \mid a \\ E \rightarrow b \end{array}$$

Solution

Left factored grammar

$$\begin{array}{l} S \rightarrow i\ E\ t\ S\ S' \mid a \\ S' \rightarrow e\ S \mid \epsilon \\ E \rightarrow b \end{array}$$

___________________Top-Down Parsing___________________

Top-down parsing can be viewed as the problem of **constructing a parse tree** for the input string, **starting from the root and creating the nodes of the parse tree in <u>preorder</u>** (depth first).

<u>Top-down parsing can be viewed as finding a "leftmost derivation" for an input string</u>

The sequence of parse trees in given below for the input **id+id*id** is a top-down parse according to the following grammar:

$$\begin{array}{lcl} E & \rightarrow & T\ E' \\ E' & \rightarrow & +\ T\ E' \mid \epsilon \\ T & \rightarrow & F\ T' \\ T' & \rightarrow & *\ F\ T' \mid \epsilon \\ F & \rightarrow & (\ E\ ) \mid \mathbf{id} \end{array}$$

Solution
This sequence of trees corresponds to a leftmost derivation of the input.

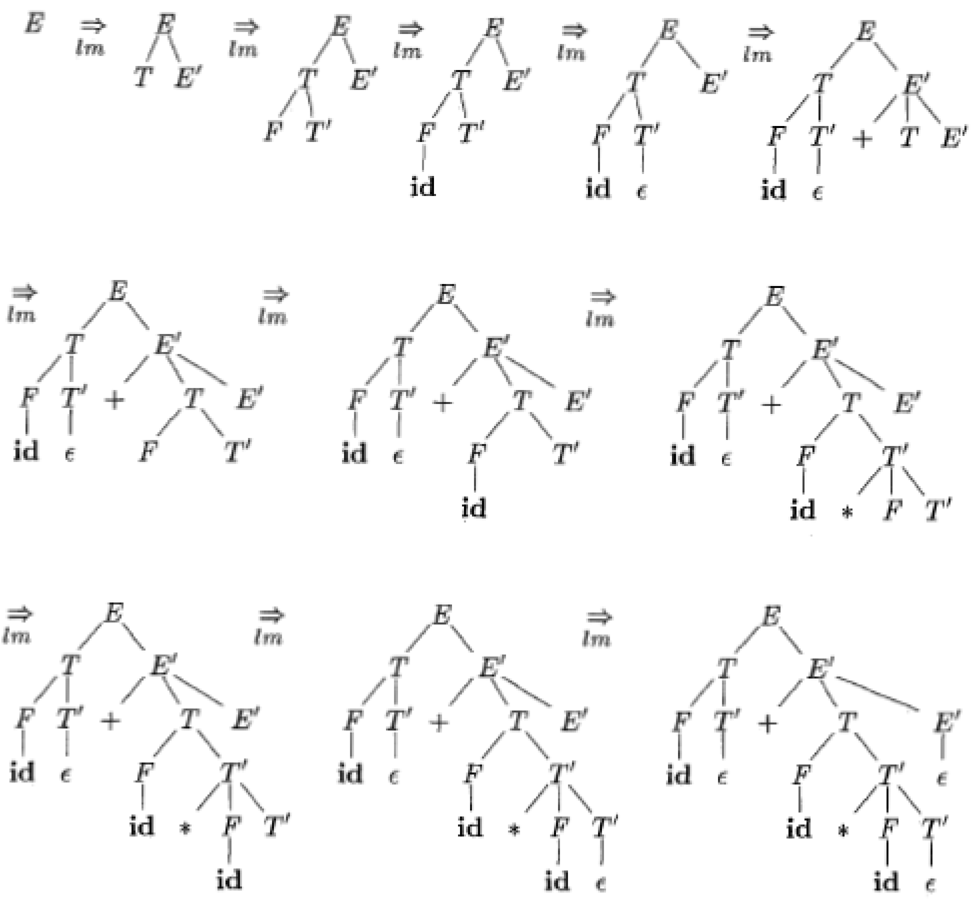


Figure 4.12: Top-down parse for **id** + **id** * **id**

At each step of a top-down parse, **<u>the key problem is that of determining the production to be applied for a nonterminal</u>**, say A. Once an A-production is chosen, <u>the rest of the parsing process consists of "matching" the terminal symbols in the production body with the input string</u>.

The **Top-Down** construction of a parse tree is done by **starting with the root**, labeled with the starting nonterminal, and repeatedly performing the following two steps.

1. At node N, labeled with nonterminal A, **select one of the productions** for A and **construct children at N** for the symbols in the production body.
2. Find the **next node** at which a subtree is to be constructed, typically the **leftmost unexpanded** nonterminal of the tree.

In general, the selection of a production for a nonterminal may involve **trial-and-error**; that is, we may have to try a production and **backtrack** to try another production if the first is found to be unsuitable.
A production is **unsuitable** if, after using the production, **we cannot complete the tree to match the input string**.

Note – "**Backtracking is not needed, however, in an important special case called predictive parsing".**

Predictive Parsing
Recursive-**descent parsing** is a **top-down method** of syntax analysis in which a **set of recursive procedures is used to process the input**.
One procedure is **associated with each nonterminal** of a grammar.

Predictive parsing is a simple form of recursive-descent parsing

In which the **lookahead** symbol **unambiguously determines** the flow of control through the procedure body for **each nonterminal**.

The class of grammars for which we can construct **predictive parsers looking k symbols ahead** in the input is sometimes called the **LL(k) class**.

Recursive-Descent Parsing

```
void A() {
    Choose an A-production, $A \rightarrow X_1 X_2 \cdots X_k$;
    for ( $i = 1$ to $k$ ) {
        if ( $X_i$ is a nonterminal )
            call procedure $X_i()$;
        else if ( $X_i$ equals the current input symbol $a$ )
            advance the input to the next symbol;
        else /* an error has occurred */;
    }
}
```

Figure 4.13: A typical procedure for a nonterminal in a top-down parser

A recursive-descent parsing program consists of a **set of procedures, one for each nonterminal**.
Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body **scans the entire input string**.

Example Consider the grammar

$$\begin{array}{lcl} S & \rightarrow & c\,A\,d \\ A & \rightarrow & a\,b \mid a \end{array}$$

To construct a parse tree top-down for the input string w = **cad**.

Solution

1. S has only one production, so we use it to expand S and obtain the tree.
2. The leftmost leaf, labeled c, matches the first symbol of input w, so we advance the input pointer to a, the second symbol of w, and consider the next leaf, labeled A.
3. Now, we expand A using the first alternative **A → ab** to obtain the tree, we have a match for second input symbol a, so we advance the input pointer to **d**.
4. The third input symbol, and compare d against the next leaf, labeled b. Since b does not match d, we **report failure** and go **back to A** to see whether there is another alternative for A that has not been tried, but that might produce a match.
5. In going back to A, we must reset the input pointer to position 2, the position it had when we first came to A.
6. The second alternative for A produces the tree, and the procedure halts.

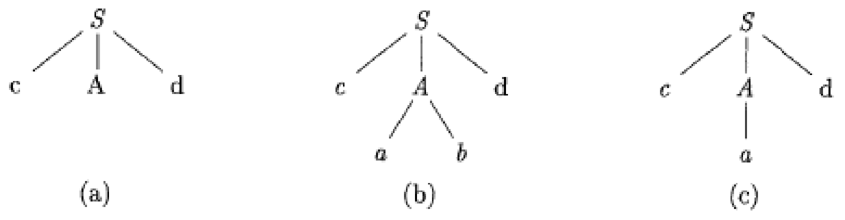


Figure 4.14: Steps in a top-down parse

Note - A **left-recursive grammar** can cause a **recursive-descent parser**, even one with backtracking, to go into an **infinite loop**.

FIRST and FOLLOW

The construction of both **top-down and bottom-up parsers** is **aided** by two functions, **FIRST** and **FOLLOW**, associated with a grammar G.

1. During **top-down parsing**, **FIRST and FOLLOW allow us to choose which production to apply**, based on the **next input symbol**.
2. During **panic-mode error recovery**, sets of tokens produced by **FOLLOW** can be used as synchronizing tokens.

First(X)

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or **ϵ** can be added to any **FIRST** set.

1. A → aα   First (A) = {a} where a is terminal.
2. A → Bα   First(A) = First(B), where A, and B are non-terminals, and B doesn't **ϵ**.
3. A → Bα   {First(B) - ϵ} U {First(α)} where B,α are non-terminals where B contains **ϵ**.

Example

i. S → aAB
A → b
B → c
First(S) = {a}
First(A) = {b}
First(B) = {c}

ii. S → AB
A → a | ϵ
B → b

First(S) = {a} U {b} = {a, b}
First(A) = {a, ϵ}
First(B) = {b}

iii. S → AB
A → a
B → b | ϵ
First(S) = {a}
First(A) = {a}
First(B) = {b, ϵ}

Follow Set

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place $ in FOLLOW(S) where S is the start symbol, and $ is the input right end marker.
2. If there is a production $A \to \alpha B \beta$ then everything in $\text{FIRST}(\beta)$ **except ϵ** is in FOLLOW(B).
3. If there is a production $A \to \alpha B$ or a production $A \to \alpha B \beta$, where $\text{FIRST}(\beta)$ contains **ϵ** then everything in FOLLOW(A) is in FOLLOW(B).

Example Find the First( ) and Follow( ) set

$$\begin{array}{lcl} E & \to & T\ E' \\ E' & \to & +\ T\ E' \mid \epsilon \\ T & \to & F\ T' \\ T' & \to & *\ F\ T' \mid \epsilon \\ F & \to & (\ E\ ) \mid \mathbf{id} \end{array}$$

i. First(E) = First(T) = First(F) = {(, id}
ii. First(E’) = {+, ϵ }
iii. First(T’) = {*, ϵ }

a. Follow(E) = Follow(E’) = {$,)}
b. Follow(T) = Follow(T’)= {+, $,)}
c. Follow(F) = {*, +, $,)}

LL(1) Grammars

Predictive parsers, that is, recursive-descent parsers needing no backtracking, can be constructed for a class of grammars called LL(1).

The first "L" in **LL(1)** stands for scanning the **input from left to right**, the second "L" for producing a **leftmost derivation**, and the **"1" for using one input symbol of lookahead** at each step to make parsing action decisions.

1. The class of LL(1) grammars is rich enough to cover most programming constructs.
2. No **left-recursive** or **ambiguous grammar** can be LL(1).

A grammar G is LL(1) if and only if whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:

1. First(α) ∩ First(β) = Φ, Where, For **no terminal 'a', do both α and β derive strings** beginning with **a**.
2. **At most α and β** can derive the empty string.
3. If $\beta \overset{*}{\Rightarrow} \epsilon$ then **First(α)** ∩ **Follow(β)** = Φ, and if $\alpha \overset{*}{\Rightarrow} \epsilon$ then **First(β)** ∩ **Follow(α)**.

Construction of a predictive parsing table

INPUT: Grammar G

OUTPUT: Parsing table M

METHOD: For each production $A \rightarrow \alpha$ of the grammar, do the following:

1. For each terminal a in FIRST(A), add $A \rightarrow \alpha$ to $M[A, a]$.
2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, b]$. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, \$]$ as well.

If, after performing the above, there is no production at all in $M[A, a]$, then set $M[A, a]$ to **error** (which we normally represent by an empty entry in the table). □

Example construct LL(1) parsing table for the following grammar

$$\begin{array}{lcl} E & \rightarrow & T\ E' \\ E' & \rightarrow & +\ T\ E' \mid \epsilon \\ T & \rightarrow & F\ T' \\ T' & \rightarrow & *\ F\ T' \mid \epsilon \\ F & \rightarrow & (\ E\ ) \mid \mathbf{id} \end{array}$$

i. First(E) = First(T) = First(F) = {(, id}
ii. First(E') = {+, ϵ }
iii. First(T') = {*, ϵ }

a. Follow(E) = Follow(E') = {$,)}
b. Follow(T) = Follow(T')= {+, $,)}
c. Follow(F) = {*, +, $,)}

	id	+	*	(	)	$
E	E->TE'			E->TE'		
E'		E'-> +TE'			E'->eps	E'->eps
T	T->FT'			T->FT'		
T'		T'->eps	T'->*FT'		T'->eps	T'->eps
F	F->id			F->€		

1. For **every LL(1) grammar**, each parsing-table entry **uniquely identifies a production** or **signals an error**.
2. If G is **left-recursive** or **ambiguous**, then M will have at least one **multiple** entries. Although left **recursion elimination** and **left factoring** are easy to do, but

There are some grammars for which "no amount of alteration" will produce an LL(1) grammar

The language in the following example has **no LL(1) grammar** at all.

Example 4.33: The following grammar, which abstracts the dangling-else problem, is repeated here from Example 4.22:

$$\begin{aligned} S &\rightarrow iEtSS' \mid a \\ S' &\rightarrow eS \mid \epsilon \\ E &\rightarrow b \end{aligned}$$

Solution
First(S) = {i, a}
First(S') = {e, eps}
First(E) = b
Follow(S') = Follow(S) = {$, e}
First(S') ∩ Follow(S') != Φ, hence this grammar is **not LL(1)**.

NON - TERMINAL	INPUT SYMBOL					
	a	b	e	i	t	$\$$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

Parsing Table M

This grammar is ambiguous!!

Nonrecursive Predictive Parsing

A **nonrecursive predictive parser** can be built by maintaining **a stack explicitly**, rather than implicitly via recursive calls. **The parser mimics a leftmost derivation**. **If w is the input that has been matched so far,**
then the stack holds a sequence of grammar symbols α such that

$$S \overset{*}{\underset{lm}{\Rightarrow}} w\alpha$$

1. The table-driven parser in has an **input buffer**, a **stack** containing **a sequence of grammar symbols**, a **LL(1) parsing table** and an **output stream**.
2. The input buffer contains the string to be parsed, followed by the end marker **$**.
3. We reuse the symbol **$** to mark the **bottom of the stack**, which **initially contains the start symbol of the grammar on top of $**.

Suppose, **X is the symbol on top of the stack**, and **a**, **the current input symbol**.
1. If X is a **nonterminal**, the parser chooses an X-production by consulting entry **M[X, a]** of the parsing table M.

Algorithm Table-driven predictive parsing.
INPUT: A string w and a parsing table M for grammar G.
OUTPUT: If w is in L(G), **a leftmost derivation of w**; otherwise, an error indication

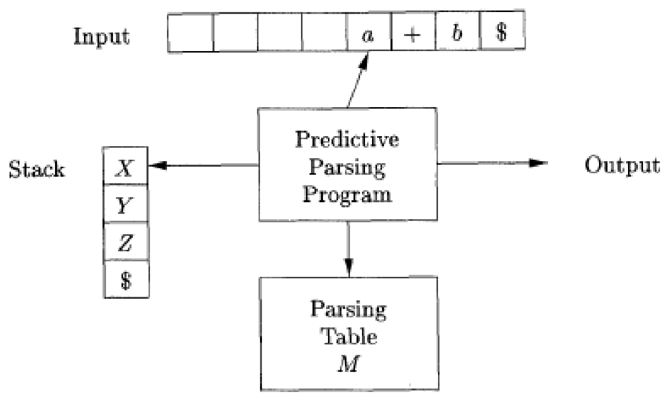


Figure 4.19: Model of a table-driven predictive parser

Method:
Initially, the parser is in a configuration with **w$ in the input buffer** and the start symbol **S of G on top of the stack, above $**.

- Set i/p to point to the first symbol of w.
- Set X to the top stack symbol.

1. If **X = a = $** input string is valid and accepted.
2. If **(X=a!=$)** X is a terminal on the top of the stack and match with input symbol, POP X from stack and advance the i/p.
3. If (**X is a non-terminal**) parser takes decision from parsing table
   If M[X, a] = uvw then PUSH into the stack such a way that **u is on the top**.

EXAMPLE

	id	+	*	(	)	$
E	E->TE'			E->TE'		
E'		E'-> +TE'			E'->eps	E'->eps
T	T->FT'			T->FT'		
T'		T'->eps	T'->*FT'		T'->eps	T'->eps
F	F->id			F->€		

input:	id	+	id	*	id	$

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $T' \to \epsilon$
$\mathbf{id}$	$+ TE'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $E' \to + TE'$
$\mathbf{id} +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id} +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
$\mathbf{id} +$	$\mathbf{id}\ T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$* FT'E'\$$	$* \mathbf{id}\$$	output $T' \to * FT'$
$\mathbf{id} + \mathbf{id} *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id} *$	$\mathbf{id}\ T'E'\$$	$\mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \to \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \to \epsilon$

Figure 4.21: Moves made by a predictive parser on input $\mathbf{id} + \mathbf{id} * \mathbf{id}$

Moves made by a predictive parser on input id + id * id

The nonrecursive predictive parser makes the sequence of moves, these moves correspond to a leftmost derivation

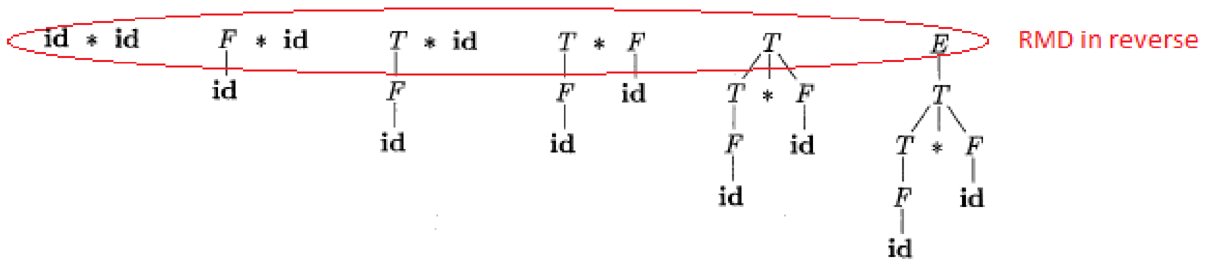
$$E \underset{lm}{\Rightarrow} TE' \underset{lm}{\Rightarrow} FT'E' \underset{lm}{\Rightarrow} \mathbf{id}\,T'E' \underset{lm}{\Rightarrow} \mathbf{id}\,E' \underset{lm}{\Rightarrow} \mathbf{id} + TE' \underset{lm}{\Rightarrow} \cdots$$

(Notice the content of stack from above to down, this is same as LMD)

Note that the **sentential forms** in this derivation correspond to the input that has already been matched (in column MATCHED) followed by the stack contents.

Bottom up Parsing

A bottom-up parser corresponds to the construction of a parse tree for an input string **beginning at the leaves (the bottom) and working up towards the root (the top)**.



A bottom-up parse for id * id

A bottom-up parse of the token stream id * id, with respect to the following expression grammar

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array}$$

The goal of bottom-up parsing is therefore to construct a derivation in reverse.
RMD

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$$

Reductions

We can think of bottom-up parsing as the process of "**reducing**" **a string w to the start symbol of the grammar**. At each reduction step, a specific substring matching the body of a production is replaced by the **nonterminal** at the head of that production.

The sequence starts with the input string **id*id**. The first reduction produces F * id by reducing the leftmost **id to F**, using the production F → id. Second reduction produces T * id by reducing **F to T**.

Now, **we have a choice between reducing the string T**, which is the body of E → T, and the string consisting of the second id, which is the body of F → id. Rather than **reduce T to E**, the second id is reduced to T, resulting in the string T * F. This string then reduces to T. The parse completes with the reduction of T to the start symbol E.

Note- A reduction is the reverse of a step in a derivation.

Handle

Bottom-up parsing during a **left-to-right scan** of the input constructs a **rightmost derivation in reverse**. Informally, a "**handle**" **is a substring that matches the body of a production**, and whose reduction represents one step along the reverse of a rightmost derivation.

Right Sentential Form	Handle	Reducing Production
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \rightarrow \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \rightarrow F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \rightarrow \mathbf{id}$
$T * F$	$T * F$	$E \rightarrow T * F$

Figure 4.26: Handles during a parse of $\mathbf{id}_1 * \mathbf{id}_2$

1. Notice that the **string w to the right of the handle must contain only terminal** symbols.
2. If the grammar is **unambiguous** then **every right-sentential form of the grammar** has **exactly one handle**.
3. A **rightmost derivation in reverse** can be obtained by "handle" reductions.

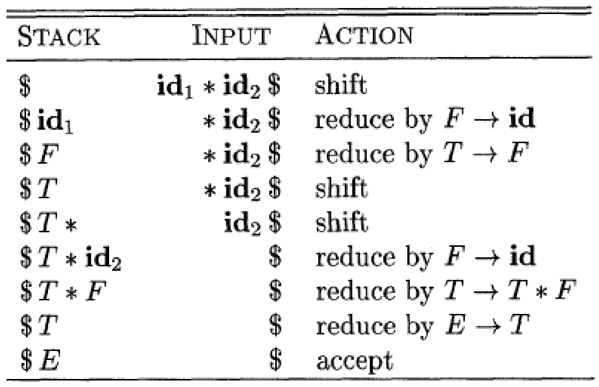
Shift-Reduce Parsing

Shift-reduce parsing is a form of **bottom-up parsing** in which
1. A **stack** holds **grammar symbols**, and
2. An **input buffer** holds the **rest of the string to be parsed**.
3. We use $ to mark the **bottom of the stack** and also the **right end** of the input.

Note - **The handle always appears at the top of the stack just before it is identified as the handle, never inside.**

1. During a **left-to-right scan** of the input string, the **parser shifts zero or more input symbols** onto the **stack**, until it is **ready to reduce** a **string** β of **grammar symbols** on top of the stack.
2. It then reduces β to the head of the appropriate production.
3. The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the **input is empty**.

Following are the steps through the actions a shift-reduce parser might take in parsing the input string **id*id**.



STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2\,\$$	shift
$\$\,\mathbf{id}_1$	$* \mathbf{id}_2\,\$$	reduce by $F \to \mathbf{id}$
$\$\,F$	$* \mathbf{id}_2\,\$$	reduce by $T \to F$
$\$\,T$	$* \mathbf{id}_2\,\$$	shift
$\$\,T\,*$	$\mathbf{id}_2\,\$$	shift
$\$\,T * \mathbf{id}_2$	$\$$	reduce by $F \to \mathbf{id}$
$\$\,T * F$	$\$$	reduce by $T \to T * F$
$\$\,T$	$\$$	reduce by $E \to T$
$\$\,E$	$\$$	accept

3: Configurations of a shift-reduce parser on input $\mathbf{id}_1 * \mathbf{id}_2$

While the primary operations are shift and reduce, there are actually **four possible actions** a shift-reduce parser can make:
1. **Shift**, Shift the next input symbol onto the top of the stack.
2. **Reduce**, the **right end of the string** to be **reduced must be at the top of the stack**. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
3. **Accept**, announce successful completion of parsing.
4. **Error**, discover a syntax error and call an error recovery routine.

Conflicts during Shift-Reduce Parsing

There are context-free grammars for which shift-reduce parsing cannot be used. Parser cannot decide whether to shift or to reduce (a **shift/reduce** conflict), or cannot decide which of several reductions to make (a **reduce/reduce** conflict).
These grammars are not in the LR(k) class of grammars.

Introduction to LR Parsing: Simple LR

LR(K) Parsing:

The "**L**" is for **left-to-right scanning of the input**, the "**R**" for constructing a **rightmost derivation in reverse**, and the **k** for the **number of input symbols of lookahead** that are used in making parsing **decisions**.

Note- LR(K), when K is not given we assume it to be 1.

Why LR Parsers?

1. Why LR Parsers?
2. LR parsers can be constructed to recognize virtually all programming language constructs for which context-free grammars can be written.
3. **Non- LR context-free grammars exist**.
4. An LR parser can detect a **syntactic error as soon as it is possible** to do so on a left-to-right scan of the input.
5. The class of grammars that can be parsed using **LR methods is a proper superset** of the class of grammars that can be parsed **with predictive or LL methods**.
6. For a grammar to be **LR(k)**, we must be able to recognize the **occurrence of the right side of a production in a right-sentential form**, with k input symbols of lookahead.
7. We have many LR parser generators available such a generator **takes a context-free grammar and automatically produces a parser for that grammar**.

LR(0) Parser

How does a shift-reduce parser know when to shift and when to reduce? An LR parser makes **shift-reduce** decisions **by maintaining states** to keep track of where we are in a parse.

1. **States represent sets of "LR(0) items**."
2. An **LR(0) item** (item for short) of a grammar G is **a production of G with a dot** at some position of the body.

Thus, production A → XYZ yields the four LR(0) items or items:

$$A \to \cdot XYZ$$
$$A \to X \cdot YZ$$
$$A \to XY \cdot Z$$
$$A \to XYZ \cdot$$

Note The production $A \to \epsilon$ generates only one item, $A \to \cdot$

Intuitively, an item indicates **how much of a production we have seen** at a given point in the parsing process.

1. The item **A → .XYZ** indicates that we hope to see a string derivable from XYZ next on the input.
2. **A → X.YZ** indicates that we have just seen on the input a string **derivable from X** and that we hope **next to see a string derivable from YZ**.
3. Item **A → XYZ.** indicates that we have seen the body XYZ and that it **may be** time to reduce XYZ to A.

1. Collection of sets of LR(0) items, called the **canonical LR(0) collection**.
2. A **deterministic finite automaton** is used to make parsing decisions.
3. Such an automaton is called an **LR(0) automaton**. Each state of the LR(0) automaton represents **a set of items** in the canonical LR(0) collection.

To construct the canonical **LR(0)** collection for a grammar,

1. We define an **augmented grammar** and two functions, **CLOSURE** and **GOTO**.
2. If **G is a grammar** with **start symbol S**, then **G', the augmented grammar** for G, is G with a **new start symbol S'** and production **S' → S**.
3. Input acceptance occurs when and only when the parser is about to reduce by St → S.

Closure of Item Sets

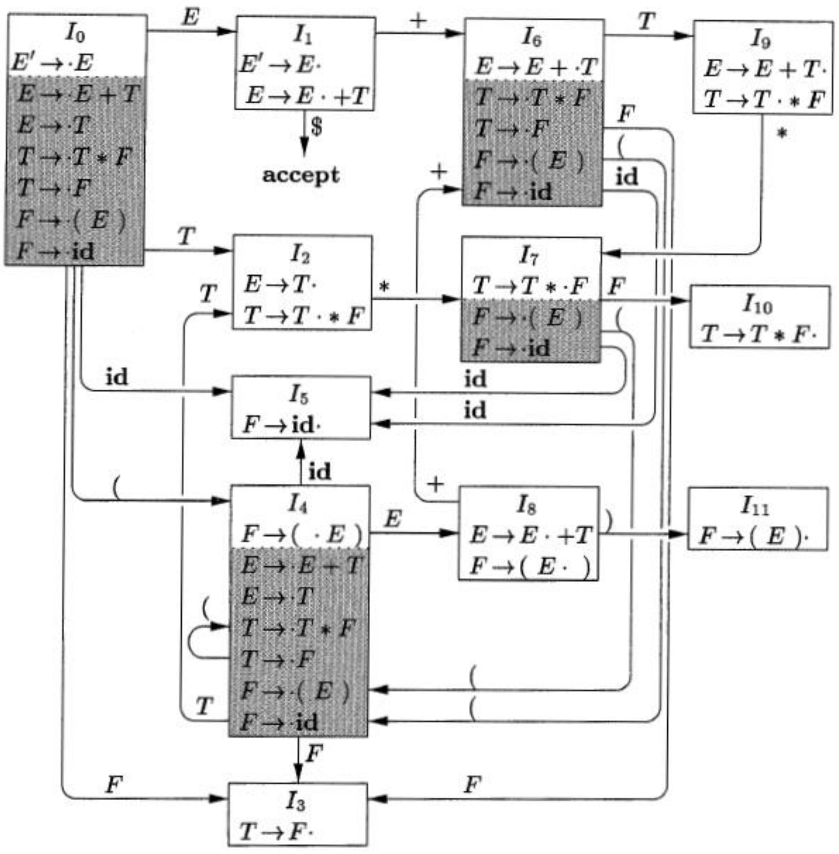
If **I is a set of items for a grammar G**, then **CLOSURE**(I) is the **set of items constructed from** I by the two rules:

1. Initially, add every item in I to CLOSURE(I).
2. If $A \to \alpha{\cdot}B\beta$ is in $\text{CLOSURE}(I)$ and $B \to \gamma$ is a production, then add the item $B \to {\cdot}\gamma$ to $\text{CLOSURE}(I)$ if, it is not already there. Apply this rule until no more new items can be added to CLOSURE(I).

Example Construct **LR(0) automata** for the following augmented grammar

$$\begin{array}{lcl} E' & \to & E \\ E & \to & E + T \mid T \\ T & \to & T * F \mid F \\ E & \to & (E) \mid \textbf{id} \end{array}$$

Solution



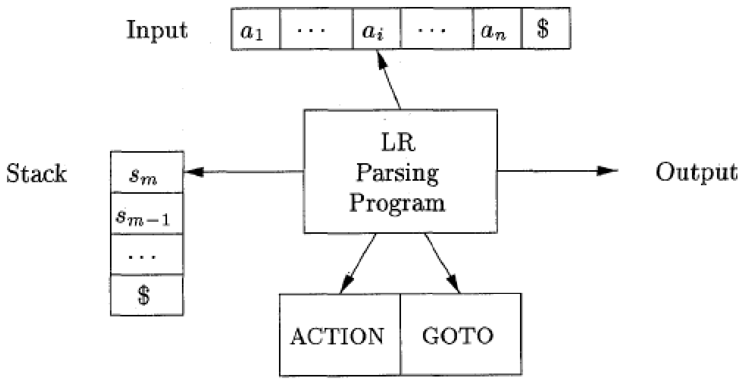
I0: E'→·E, E→·E+T, E→·T, T→·T*F, T→·F, F→·(E), F→·id
I1: E'→E·, E→E·+T
$ accept
I2: E→T·, T→T·*F
I3: T→F·
I4: F→(·E), E→·E+T, E→·T, T→·T*F, T→·F, F→·(E), F→·id
I5: F→id·
I6: E→E+·T, T→·T*F, T→·F, F→·(E), F→·id
I7: T→T*·F, F→·(E), F→·id
I8: E→E·+T, F→(E·)
I9: E→E+T·, T→T·*F
I10: T→T*F·
I11: F→(E)·
E
T
F
id
(
)
+
*

Figure 4.31: LR(0) automaton for the expression grammar (4.1)

- **Kernel items**: the initial item, **S' → .S**, and all items **whose dots are not at the left end**.
- **Non-kernel items**: all items with **their dots at the left end**, **except** for **S' → .S**

The LR-Parsing Algorithm

A **schematic** of **an LR parser** consists of an **input**, an **output**, a **stack**, a **driver program**, and a **parsing table** that has two pasts (ACTION and GOTO).
The driver program is the same for all LR parsers; **only the parsing table changes from one parser to another**.
The parsing program reads characters from an input buffer **one at a time**. Where a shift-reduce parser would shift a symbol, an LR parser shifts a state.



Model of an LR parser

Structure of the LR Parsing Table

The parsing table consists of two parts: a parsing-action function **ACTION** and a goto function **GOTO**

1. The ACTION function takes as arguments **a state i and a terminal a** (or $, the input end marker). The value of ACTION[i, a] can have one of four forms:
   a) **Shift j**, where j is a state. The action taken by the parser effectively shifts **input a** to the stack, but uses state **j to represent a**.
   b) Reduce $A \to \beta$
   c) Accept.
   d) Error.
2. We extend the GOTO function, defined on **sets of items, to states**: if $\text{GOTO}[I_i, A] = I_j$ then GOT0 also maps a **state i** and a **nonterminal A** to **state j**.

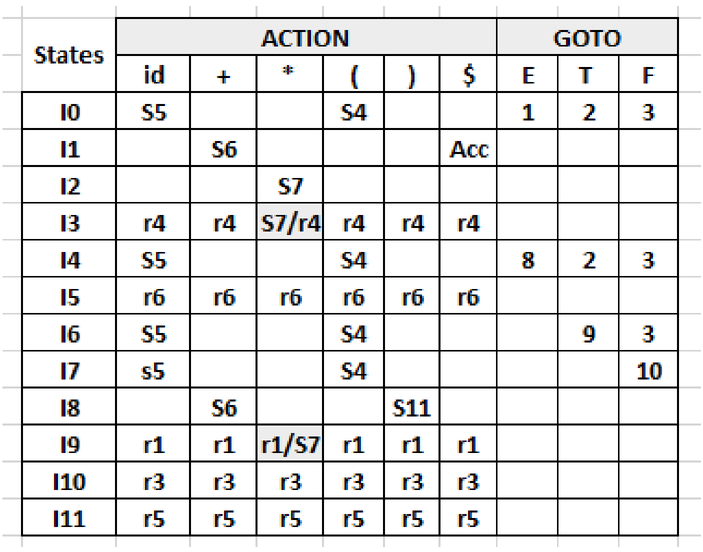
Example construct LR(0) parsing table for the following grammar

(1)	$E \to E + T$	(4)	$T \to F$
(2)	$E \to T$	(5)	$F \to (E)$
(3)	$T \to T * F$	(6)	$F \to \textbf{id}$

Note: We don't give number to extra added transition E'→E.

The codes for the actions are:

1. *si* means shift and stack state *i*,
2. r*j* means reduce by the production numbered *j*,
3. acc means accept,
4. blank means error.



States	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
I0	S5			S4			1	2	3
I1		S6				Acc			
I2			S7						
I3	r4	r4	S7/r4	r4	r4	r4			
I4	S5			S4			8	2	3
I5	r6	r6	r6	r6	r6	r6			
I6	S5			S4				9	3
I7	s5			S4					10
I8		S6			S11				
I9	r1	r1	r1/S7	r1	r1	r1			
I10	r3	r3	r3	r3	r3	r3			
I11	r5	r5	r5	r5	r5	r5			

LR(0) items table

Note:- that state I3 and I9 has shift-reduce conflicts.

Constructing SLR-Parsing Tables

The SLR method begins with **LR(0) items and LR(0) automata**. That is, given a grammar, G, we augment G to produce G’, with a new start symbol S’. From G', we construct C, **the canonical collection of sets of items for G’ together with the GOT0 function**.

Note: - Every SLR(1) grammar is **unambiguous**, but there are many unambiguous grammars that are **not SLR(1)**.

Example 4.48: Every SLR(1) grammar is unambiguous, but there are many unambiguous grammars that are not SLR(1). Consider the grammar with productions

$$\begin{array}{lcl} S & \rightarrow & L = R \mid R \\ L & \rightarrow & *R \mid \mathbf{id} \\ R & \rightarrow & L \end{array} \qquad (4.49)$$

Think of L and R as standing for l-value and r-value, respectively, and $*$ as an operator indicating "contents of."[5] The canonical collection of sets of LR(0) items for grammar (4.49) is shown in Fig. 4.39.

I_0: $S' \rightarrow \cdot S$
$S \rightarrow \cdot L = R$
$S \rightarrow \cdot R$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$
$R \rightarrow \cdot L$

I_1: $S' \rightarrow S \cdot$

I_2: $S \rightarrow L \cdot = R$
$R \rightarrow L \cdot$

I_3: $S \rightarrow R \cdot$

I_4: $L \rightarrow * \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_5: $L \rightarrow \mathbf{id} \cdot$

I_6: $S \rightarrow L = \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_7: $L \rightarrow *R \cdot$

I_8: $R \rightarrow L \cdot$

I_9: $S \rightarrow L = R \cdot$

Canonical collection of sets of LR(0) items

Consider the set of items I2. Since there is both a shift and a reduce entry in ACTION [2, =], **state 2 has a shift/reduce conflict on input symbol =**.

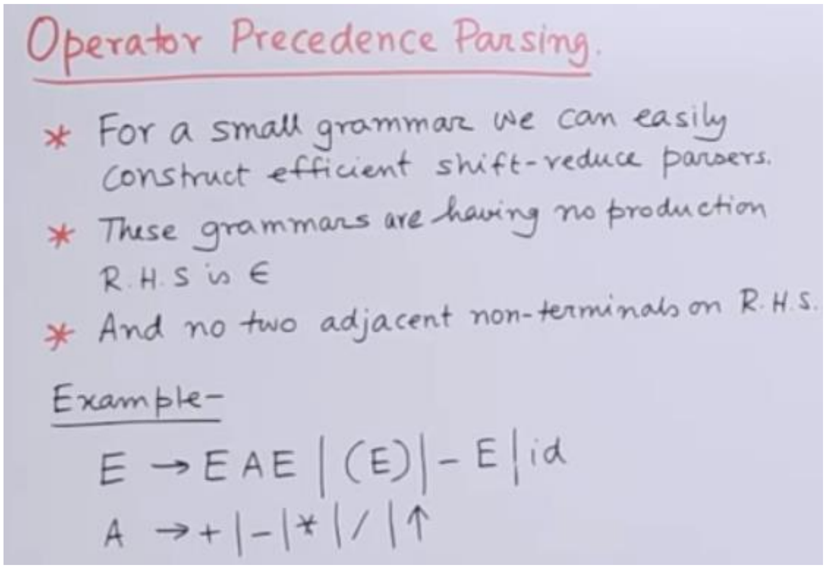
Note:- Grammar (4.9) is not ambiguous still we can't have SLR parser for it. SLR parser construction method is not powerful enough to remember enough left context to decide what action the parser should take on input =.

Note:- There are **unambiguous grammars** for which **every LR parser** construction method will produce a **parsing action table with parsing action conflicts**.

More Powerful LR Parsers

1. The "**canonical-LR**" or just "LR" method, which makes **full use of the lookahead symbol(s)**. This method uses a **large set of items, called the LR(1) items**.
2. The "lookahead-LR" or "LALR" method, which is **based on the LR(0) sets of items**, and has many fewer states than typical parsers based on the LR(1) items. By carefully in**troducing look-aheads into the LR(0) items**, we can handle many more grammars with the LALR method than with the SLR method, and build parsing tables that **are no bigger than the SLR tables**.

Operator Precedence Parsing
It can parse an ambiguous grammar also.



Operator Precedence Parsing.

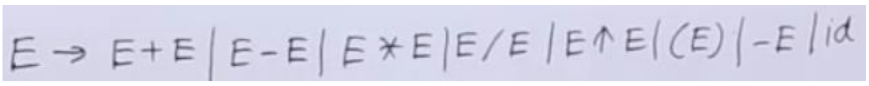
* For a small grammar we can easily construct efficient shift-reduce parsers.
* These grammars are having no production R.H.S is ϵ
* And no two adjacent non-terminals on R.H.S.

Example-

$E \rightarrow EAE \mid (E) \mid -E \mid id$

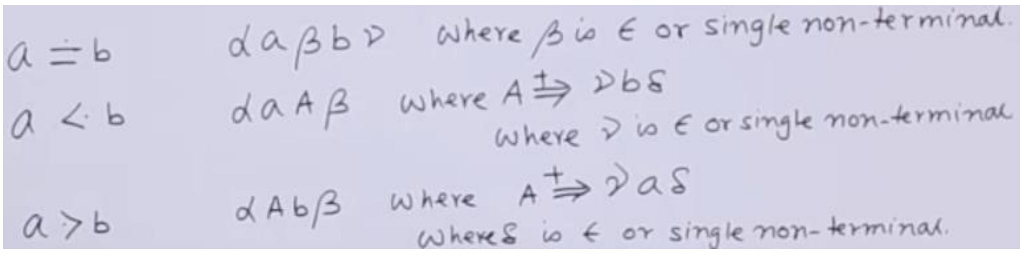
$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$

The given grammar is not Operator grammar, as there are more than 1 consecutive variables (EAE) at RHS. So, modified corresponding operator grammar is:



$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E\uparrow E \mid (E) \mid -E \mid id$

Precedence comparisons b/w two different terminals

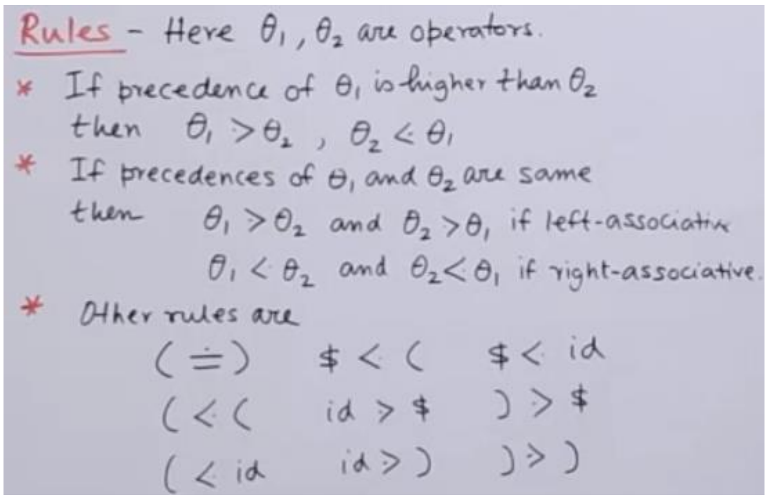


$a \doteq b$ $\qquad \alpha a \beta b \gamma$ where β is ϵ or single non-terminal.

$a \lessdot b$ $\qquad \alpha a A \beta$ where $A \overset{+}{\Rightarrow} \gamma b \delta$ where γ is ϵ or single non-terminal

$a \gtrdot b$ $\qquad \alpha A b \beta$ where $A \overset{+}{\Rightarrow} \gamma a \delta$ where δ is ϵ or single non-terminal.

Precedence comparisons b/w two different operators

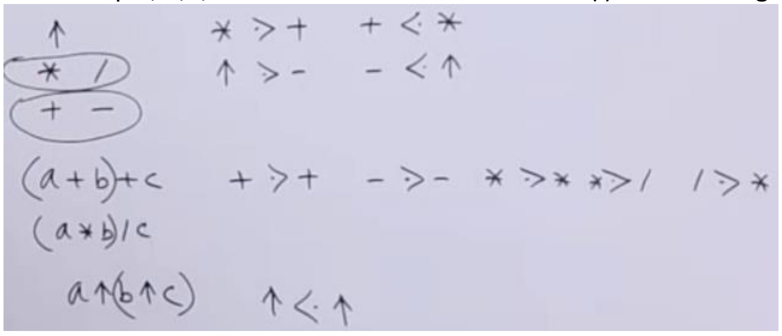


Rules - Here θ_1, θ_2 are operators.

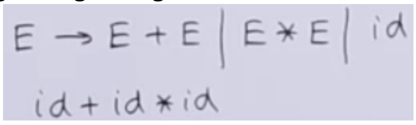
* If precedence of θ_1 is higher than θ_2 then $\theta_1 \gtrdot \theta_2$, $\theta_2 \lessdot \theta_1$
* If precedences of θ_1 and θ_2 are same then $\theta_1 \gtrdot \theta_2$ and $\theta_2 \gtrdot \theta_1$ if left-associative
  $\theta_1 \lessdot \theta_2$ and $\theta_2 \lessdot \theta_1$ if right-associative.
* Other rules are

$(\doteq)$	$\$ \lessdot ($	$\$ \lessdot id$
$(\lessdot ($	$id \gtrdot \$$	$) \gtrdot \$$
$(\lessdot id$	$id \gtrdot)$	$) \gtrdot)$

For example, +,*, and – are left associative while ( ) and ^ are right associative.



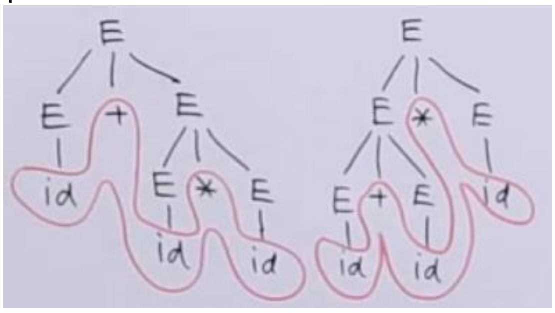
Example – Consider the following ambiguous grammar



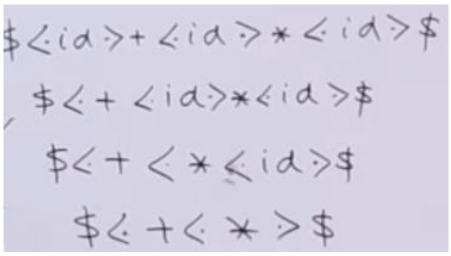
$$E \rightarrow E + E \mid E * E \mid id$$

id + id * id

Solution
Following are two different parse trees:



Before we parse the string, we write it within $, $ is end marker
$ id + id * id $
We will evaluate this expression in the following order from left to right
$ < id > + < id > * < id > $
We should start from left to right and from **less than** precedence symbol until we get **greater than** symbol and then come back till the last visited **less than** precedence, so the terminals included between < > is **handle**.



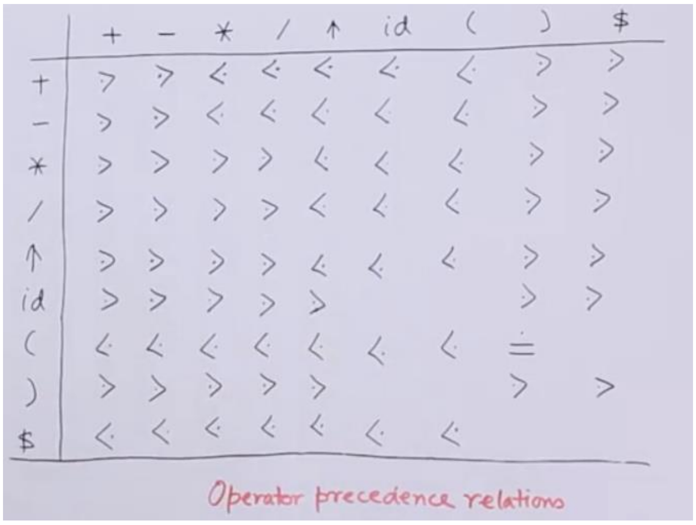
$ ⋖ id ⋗ + ⋖ id ⋗ * ⋖ id ⋗ $

$ ⋖ + ⋖ id ⋗ * ⋖ id ⋗ $

$ ⋖ + ⋖ * ⋖ id ⋗ $

$ ⋖ + ⋖ * ⋗ $

Following is the precedence relation table:



	+	−	*	/	↑	id	(	)	$
+	⋗	⋗	⋖	⋖	⋖	⋖	⋖	⋗	⋗
−	⋗	⋗	⋖	⋖	⋖	⋖	⋖	⋗	⋗
*	⋗	⋗	⋗	⋗	⋖	⋖	⋖	⋗	⋗
/	⋗	⋗	⋗	⋗	⋖	⋖	⋖	⋗	⋗
↑	⋗	⋗	⋗	⋗	⋖	⋖	⋖	⋗	⋗
id	⋗	⋗	⋗	⋗	⋗			⋗	⋗
(	⋖	⋖	⋖	⋖	⋖	⋖	⋖	≐	
)	⋗	⋗	⋗	⋗	⋗			⋗	⋗
$	⋖	⋖	⋖	⋖	⋖	⋖	⋖		

Operator precedence relations

Two id's should be separated by an operator and as id id can be consecutive hence comparisons b/w id and id is not given.
Same for id and (, as it's not possible to have id(, there should be some operator b/w id and (.
)id is not possible.)(not possible. $) not possible and $$ is not also possible.

Syntax-Directed Translation

Syntax-Directed Definitions

A syntax-directed definition specifies the values of attributes by **associating semantic rules with the grammar productions**. For example, an infix-to-postfix translator might have a production and rule

PRODUCTION	SEMANTIC RULE
$E \to E_1 + T$	$E.code = E_1.code \parallel T.code \parallel '+'$

A syntax-directed definition (SDD) is a **context-free grammar together with, attributes and rules**. **Attributes are associated with grammar symbols and rules are associated with productions**. If X is a symbol and a is one of its attributes, then we write X.a to denote the value of a at a particular parse-tree node labeled X.

Note - **Attributes may be of any kind: numbers, types, table references, or strings, for instance**.

Inherited and Synthesized Attributes

1. A synthesized attribute for a nonterminal A at a parse-tree node N is defined by a semantic rule associated with the production at N. **A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself**.

2. An inherited attribute for a nonterminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N. An inherited attribute at node N is defined only in terms of **attribute values at N's parent, N itself, and N's siblings**.

S-Attributed Definitions
An SDD is S-attributed if every attribute is **synthesized**.

L-Attributed Definitions
1. Synthesized, or
2. Inherited, but they can ask from parent or the left side siblings only.