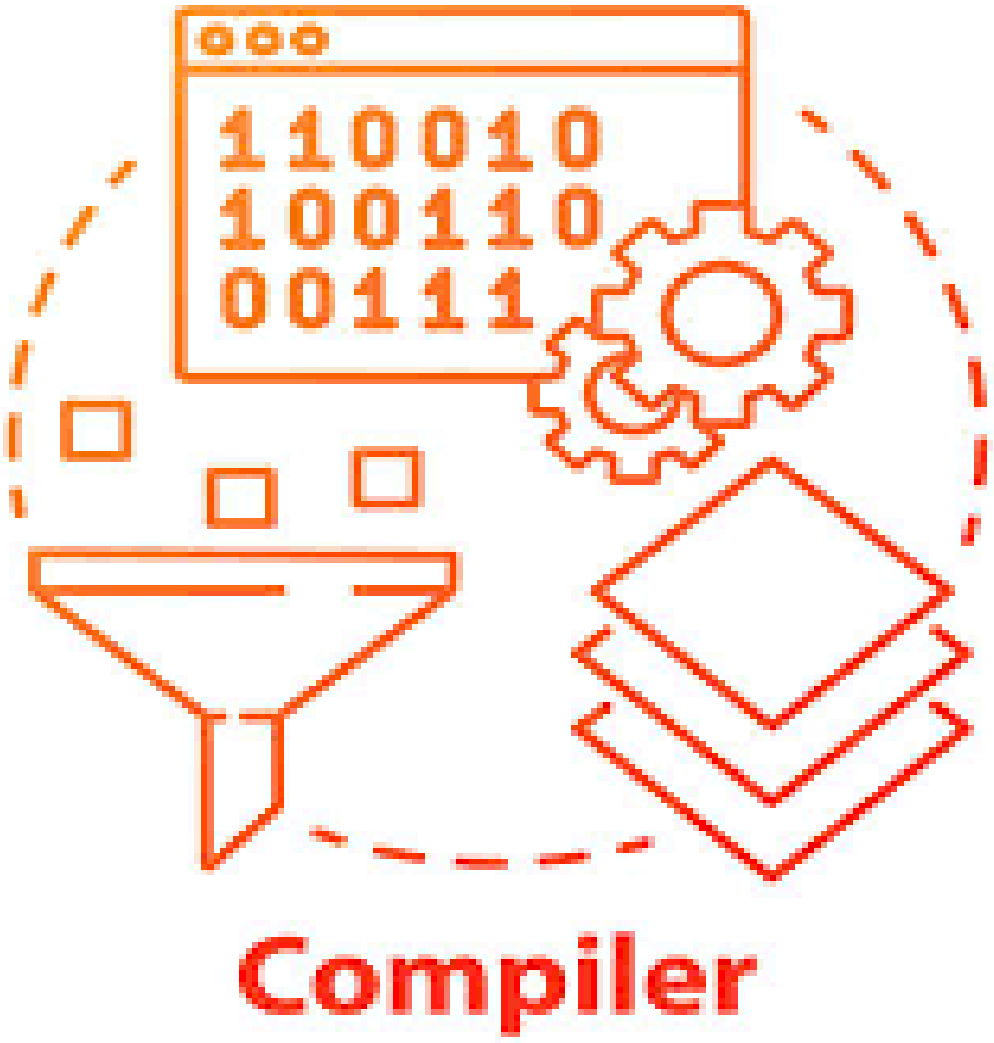


Compiler Course

Handout

Faculty of Science
Dept. of Mathematics

Compiler Course

Prepared by

Dr. Mai Kamal

For

Fourth Level

Benha University

2022

Table of Contents

Chapter 1

Introduction and Overview on Compiler Concepts

1.1 Introduction to Compiler

The programming languages are notations for describing computations to people and machines. The world as we know it depends on programming languages because all the software running on all the computers was written in some programming language. But, before a program can be run, it first must be translated into a form in which it can be executed by a computer. The software systems that do this translation are called ***compilers***.

1.2 Kinds of Languages Processors

Most programs are written in a high-level language such as C++, Java, or Python. These languages are designed more for people, rather than machines, by hiding some hardware details of a specific computer from the programmer.

Simply put, high-level languages simplify the job of telling a computer what to do. However, since computers only understand instructions in machine code (in the form of 1's and 0's), we cannot properly communicate with them without some sort of a translator. This explains the reason behind the existence of a "*language processor"*.

The language processor is a special translator system used to turn a program written in a high-level language, which we call "source code", into machine code, which we call "object program" or "object code". To design a language processor, a very precise description of lexicon and syntax, as well as the semantics of a high-level language, is needed.

There are three types of language processors:

1. Assemblers.
2. Compilers.
3. Interpreters.

1.3 Translation and Interpretation

The term *compilation* denotes the conversion of an algorithm expressed in a human-oriented *source language* to an equivalent algorithm expressed in a hardware-oriented *target language*.

Programming languages are tools used to construct formal descriptions of finite computations (algorithms). Each computation consists of operations that transform a given *initial state* into some *final state*.

1.4 What are the Compiler and Interpreter?

A *compiler* is a program that accepts as input a program text in a certain language and produces as output a program text in another language while preserving the meaning of that text. This process is called *translation*, as it would be if the texts were in natural languages. Almost all compilers translate from one input language, the *source language*, to one output language, the *target language*, only.

Figure 1.1 A Compiler

Figure1. 2. Running the target program.

To obtain the translated program, we run a compiler, which is just another program whose input is a file with the format of a program source text and whose output is a file with the format of executable code.

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.

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.

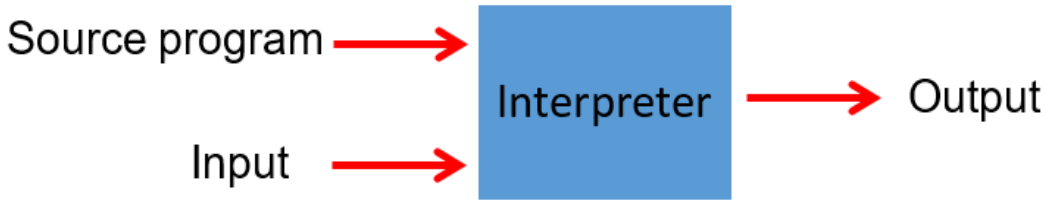


Figure1.3. An Interpreter.

Hybrid Compiler appears in some language processors that *combine* the compilation and interpretation such as *Java* Programming Language.

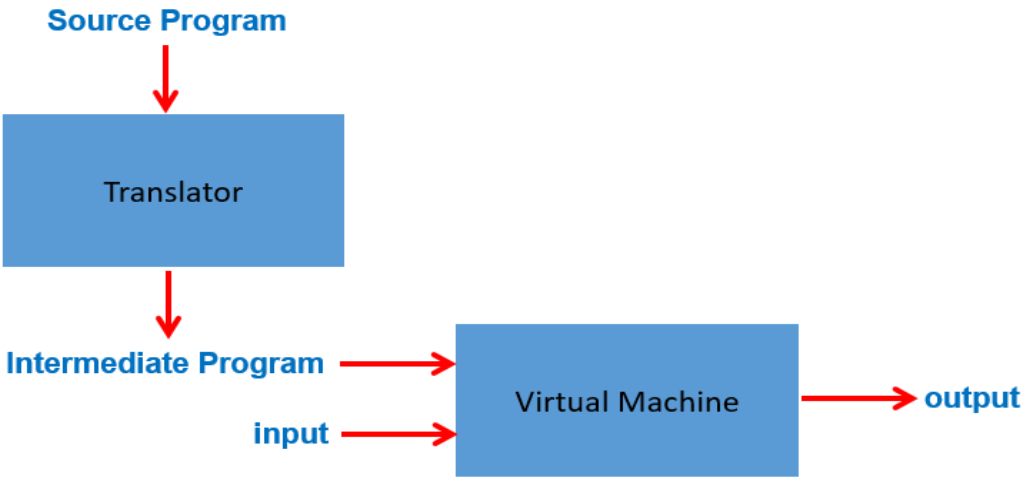


Figure 1.4. A Hybrid Compiler.

1.5 Language-processing System

A source program may be divided into modules stored in separate files. The preprocessor task collects the source program and also expands shorthands, called macros, into source language statements. The preprocessor task generates a “modified source program ” as output.

The modified source program is then fed to a compiler. The compiler produces an assembly-language program as its output. The assembly language is then processed by a program called an assembler that produces relocatable machine code as its output. Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that runs on the machine. The linker resolves external memory addresses, where the code in one file may refer to a location in another file. The loader then puts together all of the executable object files into memory for execution.

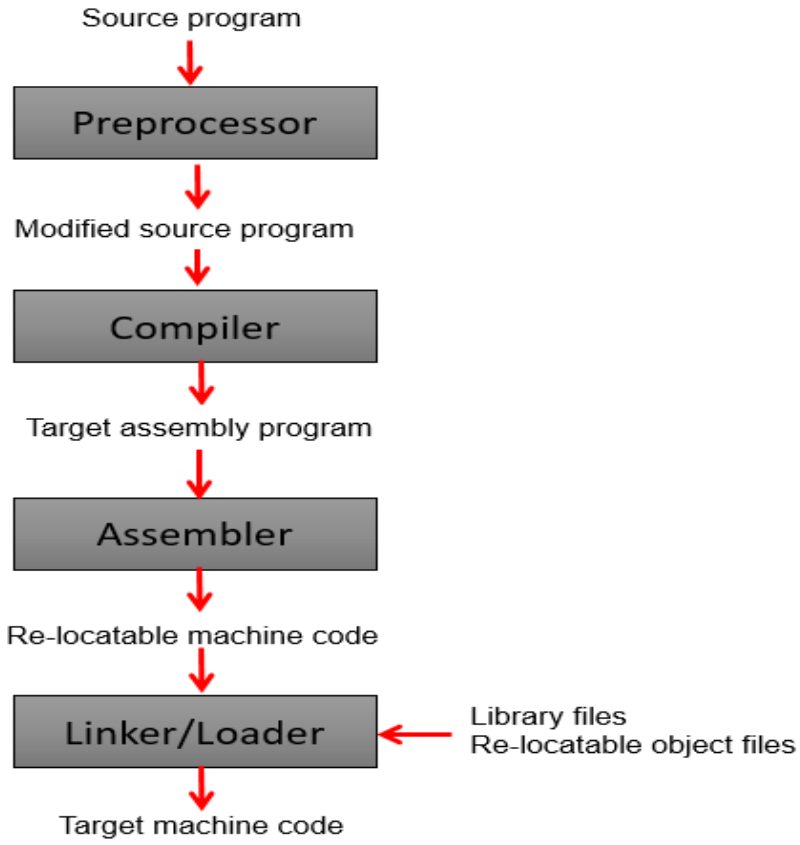


Figure 1.5 A Language Processing System.

1.6 The Tasks of a Compiler

A compilation is usually implemented as a sequence of transformations (SL, L1), (L1, L2) ……. (Lk, T L), where SL is the source language and

TL is the target language. Each language Li is called an intermediate language. Intermediate languages are conceptual tools used in decomposing the task of compiling from the source language to the target language. The design of a particular compiler determines any intermediate language programs can be broken down into two major tasks:

1. Analysis: Discover the structure and primitives of the source program, determining its meaning.
2. Synthesis: Create a target program equivalent to the source program.

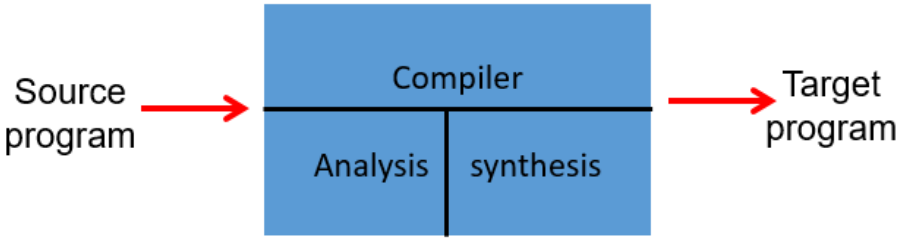


Figure1.6 The Compiler Structure.

The *analysis part* breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. 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. 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. The symbol table stores information about the entire source program, it is used by all phases of the compiler. The analysis part is often called the front end of the compiler.

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

The process of the symbol table :

1. Record the variable names used in the source program and collect information about various attributes of each name like a variable name, its type, and its scope.
2. Record the information about functions like function name, it passes by value or passes by reference.

1.7 The Structure of Compiler

The structure of the compiler is divided into seven phases. The first four phases are collectively called the frontend of the compiler and the last three phases are collectively called the backend.

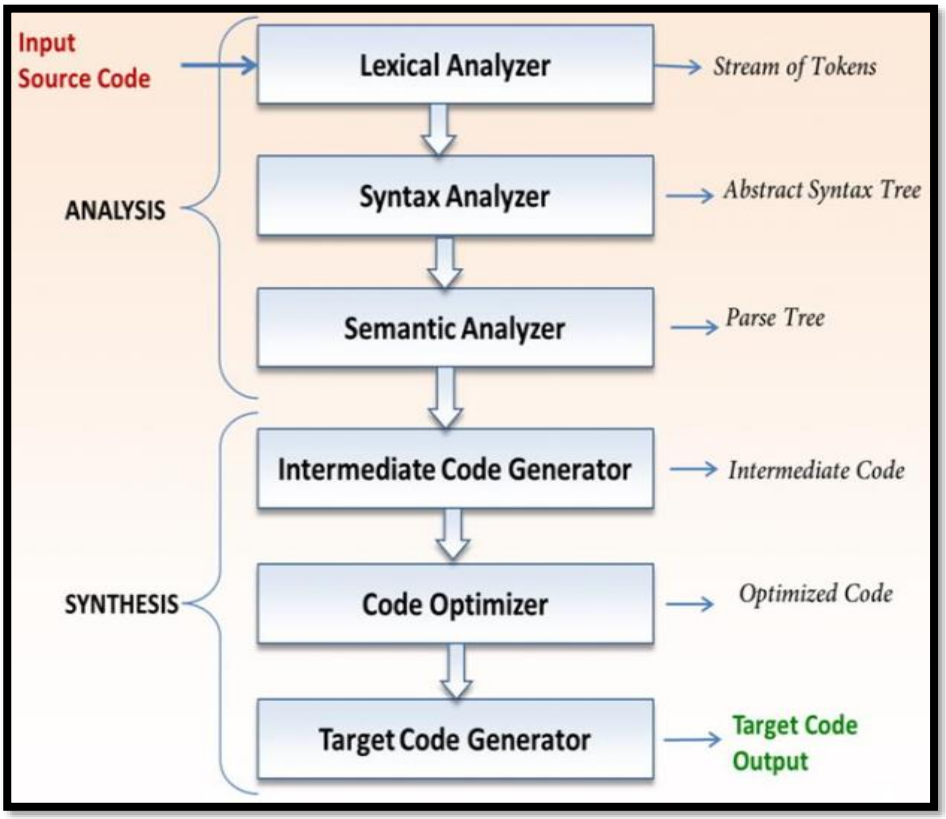


Figure 1.7.Compiler Complete Phases.

Example: Find the Translation of the Following Assignment Statement:

position = initial + rate * 60

Answer:

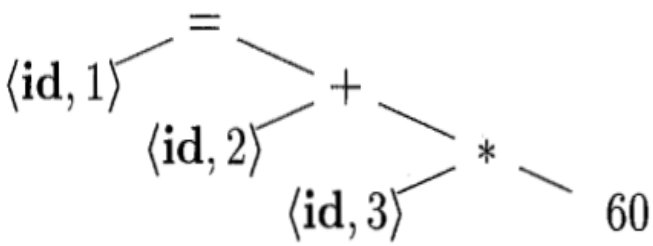
The first phase of a compiler is called *lexical analysis* or scanning. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token of the form: **<token-name, attribute-value>,** 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.

The Answer to the Lexical Phase:

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

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.

The Answer to the Syntax Phase:

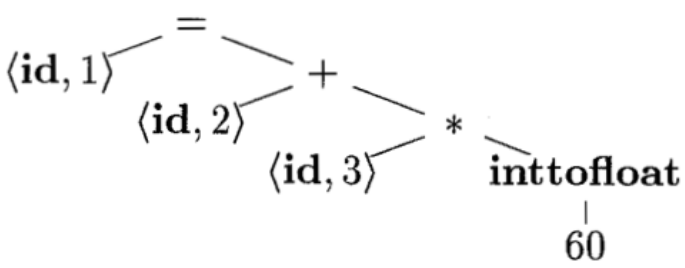


Note:

- Context-Free Grammars are used to specify the grammatical structure of programming languages and discuss algorithms for constructing the efficient syntax analyzers automatically from certain classes of grammar.

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.

The Answer to the Semantic Phase:



After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation. The intermediate form is called “three-address code”, which consists of a sequence of assembly-like instructions with three operands per instruction. Each operand can act as a register.

This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine.

The Answer to the Intermediate Code Generation Phase:

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

The code optimization phase is optional. The code-optimization phase attempts to improve the intermediate code so that better target code will result.

The Answer to the Code Optimization Phase:

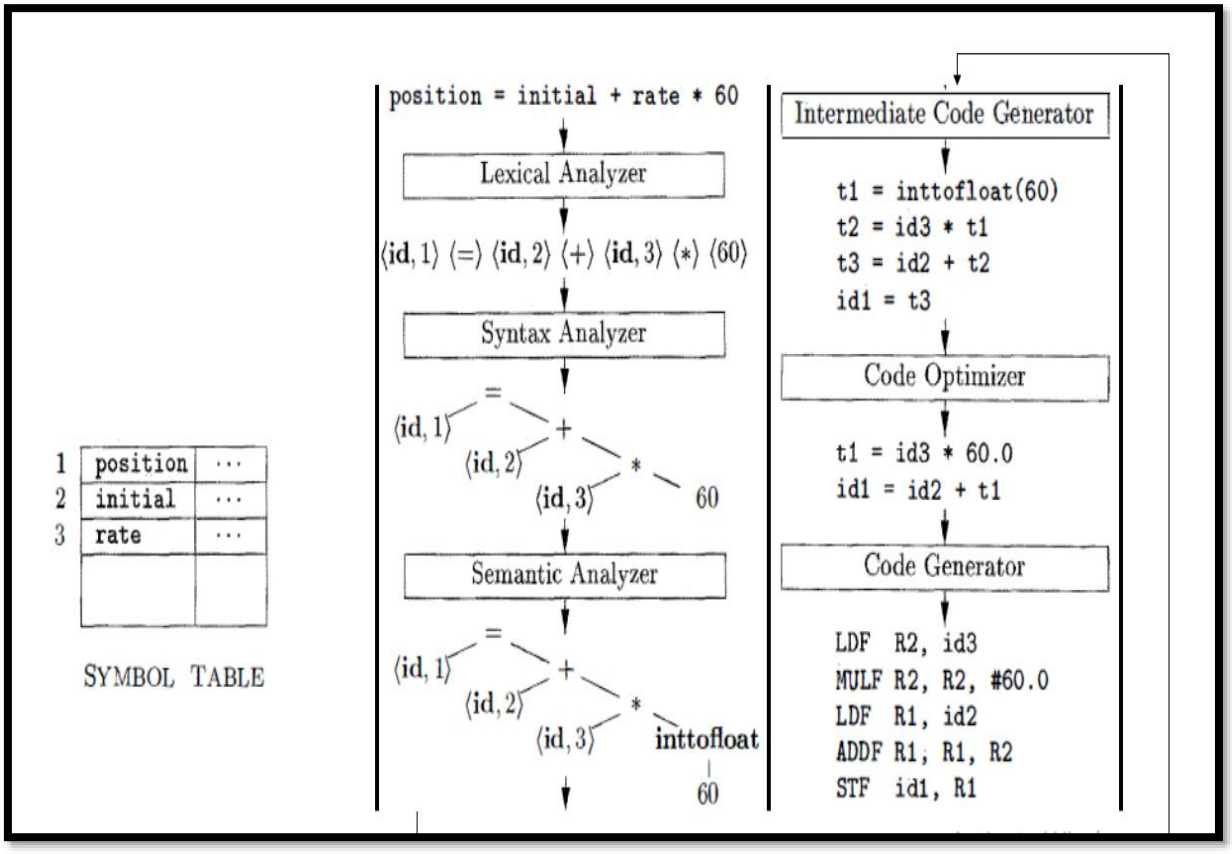
```
t1 = id3 * 60.0
id1 = id2 + t1
```

The code generation takes as input an intermediate representation of the source program and maps it into the target language.

The Answer to the Code Generation Phase:

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

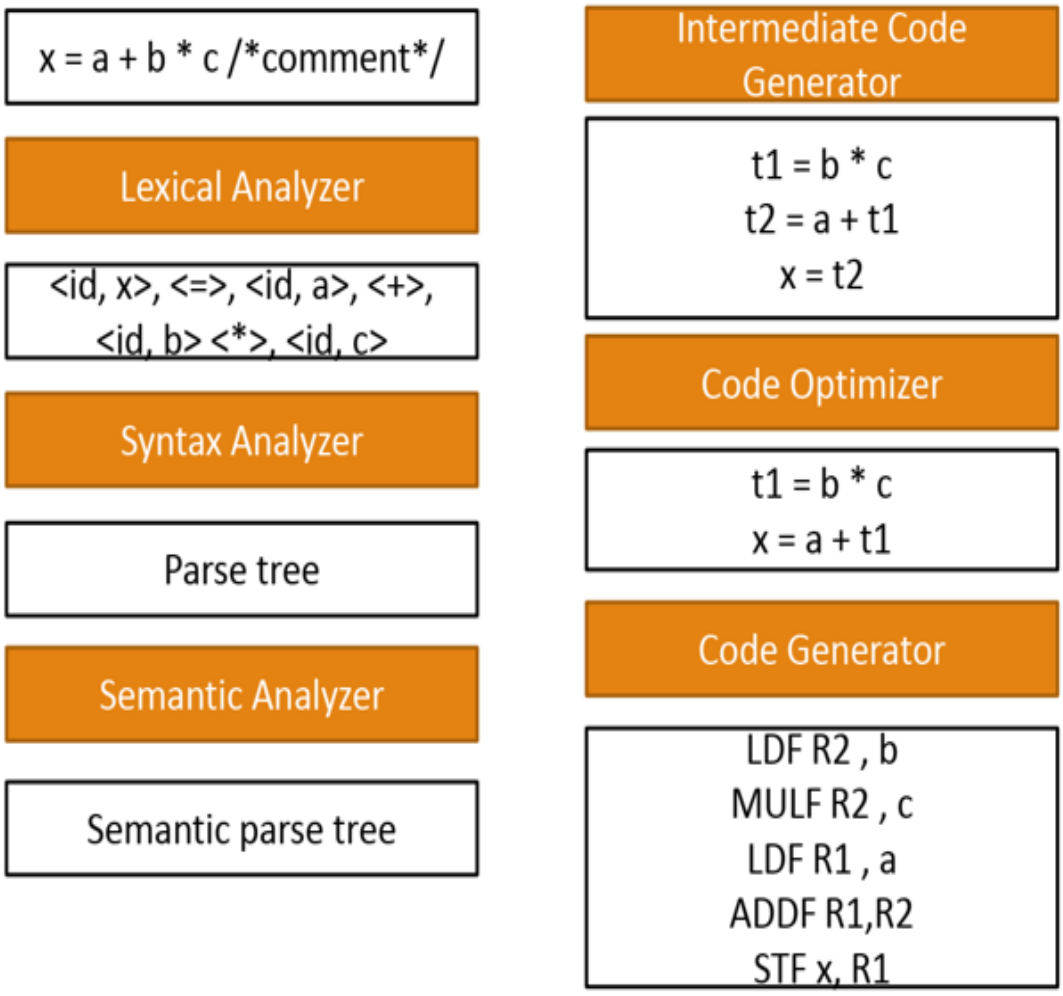
Full Answer:


position = initial + rate * 60
Lexical Analyzer
⟨id, 1⟩ ⟨=⟩ ⟨id, 2⟩ ⟨+⟩ ⟨id, 3⟩ ⟨*⟩ ⟨60⟩
Syntax Analyzer
Semantic Analyzer
inttofloat
60
1 position ···
2 initial ···
3 rate ···
SYMBOL TABLE
Intermediate Code Generator
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
Code Optimizer
t1 = id3 * 60.0
id1 = id2 + t1
Code Generator
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1

Another Example: Find the Translation of the Following Assignment Statement:

x = a + b * c

Answer:



1.8 The Chapter Exercises

1. **What is the difference between a compiler and an interpreter?**
2. **What are the advantages of (a) a compiler over an interpreter (b) an interpreter over a compiler?**
3. **What advantages are there to a language-processing system in which the compiler produces assembly language rather than machine language?**
4. **Find the Translation of the Following Statement:**

```
while (IP < z)
++ IP;
```

Chapter 2

Lexical Analysis

2.1 The Phases of a Compiler

This chapter shows and explains how to construct a lexical analyzer.

Lexical Analysis: this is the first phase of the compiler it reads and analyses the program text. The text is read and divided into tokens, each of which corresponds to a symbol in the programming language, e.g., a variable name, keyword, or number.

Syntax Analysis: is the second phase. it takes the list of tokens produced by the lexical analysis and arranges these in a tree structure (called the syntax tree) that reflects the structure of the program. This phase is often called parsing.

Intermediate Code Generation: The program is translated to a simple machine-independent intermediate language.

Machine Code Generation: The intermediate language is translated to assembly language (a textual representation of machine code) for a specific machine architecture.

2.2 Lexical Analysis

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.

Lexical Analyzers are divided into :

A. ***Scanning*** consists of simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.

B. ***Lexical analysis*** proper is the more complex portion, where the scanner produces the sequence of tokens as output.

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.

A ***token*** is a pair consisting of a **token name** and an optional **attribute value**.

Token, Their Patterns, and Attribute Values:

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
`<`	**relop**	LT
`<=`	**relop**	LE
`=`	**relop**	EQ
`<>`	**relop**	NE
`>`	**relop**	GT
`>=`	**relop**	GE

Examples of Tokens used in Most Programming Languages:

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"`

Example : **Find Tokens in this C Statement**

printf ("Total = %d\n " , score) ;

Answer:

Lexeme	Token
printf	keyword
(	symbol
"Total = %d\n "	literal
,	comma
score	id
)	symbol
;	semicolon

Another Example: **Find the token names for the Fortran statement:**

```
E = M * C ** 2
```

Answer:

<id, 1>

<assign-op >

<id, 2 >

<mult -op>

<id, 3>

<exp-op>

<number , integer value 2>

Note:

- The lexical analyzer cannot distinguish between keywords and misspelling as shown in this example:

```
fi ( a == f(x)) ...
```

- A lexical analyzer cannot tell whether **f i** is a misspelling of the keyword if or an undeclared function identifier. Since **f i** is a valid **lexeme** for the **token id**,
- The simplest recovery strategy is "***panic mode***" recovery. We delete successive characters from the remaining input until the lexical analyzer can find a well-formed token at the beginning of what input is left.

The lexical phase is based on:

1. **Regular Expressions,**
2. **Non-deterministic Automata(NFA),**
3. **Deterministic Automata(DFA).**

2.3 Operations on Languages

- **Alphabet** is any finite set of symbols.
- **String:** is a finite sequence of symbols.
- **Length of string: |S|** is the number of occurrences of the symbols in the string.
- **Language:** is a set of strings over an alphabet.

In lexical analysis, the most important operations on languages are (as shown in the below Figure):

1. union,
2. concatenation, and
3. closure.

OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

Lexical Analysis groups a stream of input characters into tokens by:

- **Specification** (Regular Expression).
- **Recognition** (Finite automata).

2.4 Regular Expressions

A regular expression is a sequence of characters that specifies a pattern in the text.

Regular Expressions Rules are :

1. **"ε" is a regular expression, and L(ε) = { ε }.**
2. **"a" is a symbol in the *alphabet* Σ, then a is a regular expression, and L(a) = {a}.**
3. **(r) is a regular expression denoting L(r) .**
4. **(r+s) = (r) | (s) is a regular expression denoting the language L(r) U L(s).**
5. **(r.s) = (r).(s) is a regular expression denoting the language L(r) . L(s).**
6. **(r)* is a regular expression denoting (L(r))***
7. **(?) is operator means zero or one occurrence.**
   - **Ex: r? = r | ε .**
8. **(+) is operator means zero or more occurrences.**
   - **Ex: r^+ = r . r* .**
9. **Shorthands**
   - **Ex: (0|1|2|3|4|5|6|7|8|9) $(0|1|2|3|4|5|6|7|8|9)^*$ = [0-9][0-9] * .**
   - **Ex: [a-z A-Z] denotes all alphabetic letters in both lower and upper case.**
   - **Ex: Variable names or identifiers : [a-z A-Z_][a-zA-Z_0-9] *.**
   - **Ex: Integers : [+ -]?[0-9]$^+$.**
   - **Ex: Float Numbers: [+-]?((([0-9]+.[0-9]∗.[0-9]+)([eE][+-]?[0-9]+)?)[0-9]+[eE][+-]?[0-9]+)**
   - **This regular expression can be simplified to : [+-]?(([0-9]$^+$(.[0-9]*)?.[0-9]$^+$)([eE][+-]?[0-9]$^+$)?)**

- **Ex: String constants: ([a-z A-Z 0-9] | [a-z A-Z])***

Examples: Describe the languages denoted by the following regular expressions:

a (a | b) * a

Answer:

L(r) ={aa, aaa, aba, aaba, abba, …….}

This language must starts and end with the symbol "a" and contain between them all the combinations of "a" and "b" strings.

Examples: Find the regular expressions for all strings of a and b of even length.

Answer:

(aa| bb| ab| ba)*

2.5 Finite State Machine

A finite automaton is a graph that comes in two forms:

- **Non-deterministic Finite Automata (NFA) and**
- **Deterministic Finite Automata (DFA).**

Finite automata say "yes" or "no" about each possible input string. Both deterministic and nondeterministic finite automata are capable of recognizing the

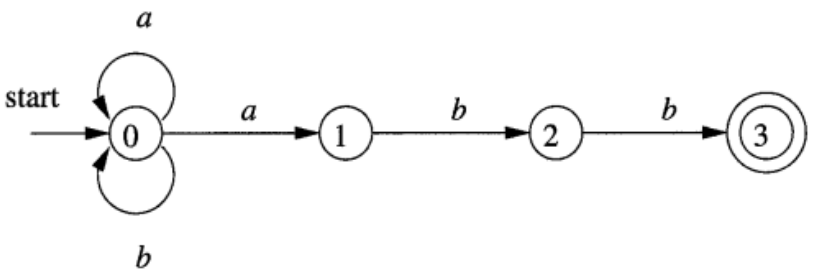
same languages. We can represent either an NFA or DFA by a transition graph, where the nodes are states and the labeled edges represent the transition function.

2.5.1 Nondeterministic Finite Automata

Properties of NFA: Have no restrictions on labels of their edges.

NFA consists of :

1. A finite set of states S.
2. A set of input symbols Σ, the input alphabet.
3. ε stands for the empty string.
4. A state s_0 from S is distinguished as the start state (or initial state).
5. A set of states F, a subset of S, that is distinguished as the accepting states (or final states).
6. A transition function that gives, for each state, and each symbol in Σ U { ε} a set of next states.



Transition Tables

We can also represent an NFA by a transition table, whose rows correspond to states, and whose columns correspond to the input symbols and ε. The entry for a given state and input is the value of the transition function applied to those arguments. If the transition function has no information about that state-input pair, we put ø in the table for the pair.

Examples: Find the transition table for the previous NFA figure

STATE	a	b	ϵ
0	$\{0,1\}$	$\{0\}$	$\emptyset$
1	$\emptyset$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

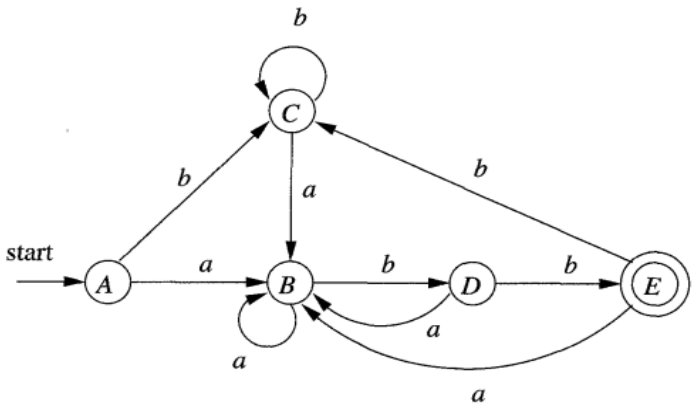
2.5.2 Deterministic Automata

The deterministic finite automaton or DFA for short obey many additional restrictions:

- There are no epsilon-transitions.
- There may not be two identically labeled transitions out of the same state.

This means that we never have a choice of several next-states: The state and the next input symbol uniquely determine the transition.

Example:



2.6 Convert R.E. into NFA

The McNaughton-Yamada-Thompson Algorithm for converting any regular expression to an NFA that defines the same language.

- It works recursively up the parse tree for the regular expression. For each sub-expression, the algorithm constructs an NFA with a single accepting state.

Input:

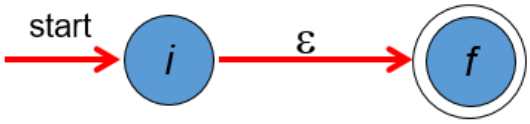
A regular expression *r* over alphabet Σ.

Output:

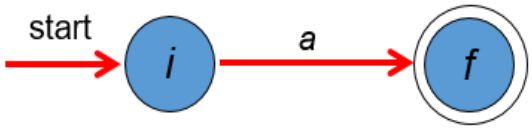
An NFA accepting *L*(*r*).

Method:

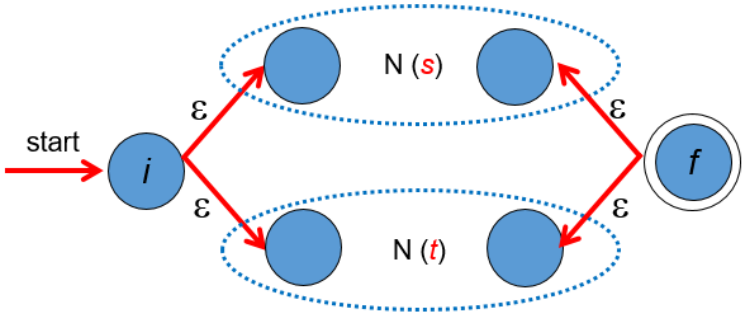
1. For expression ε construct the NFA



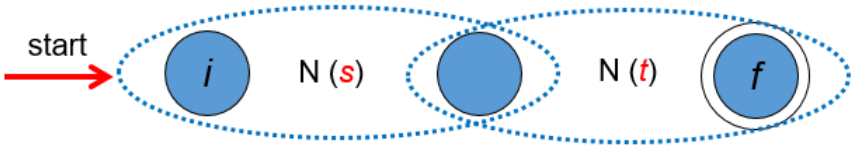
2. For any sub-expression, "*a*" in Σ, construct the NFA



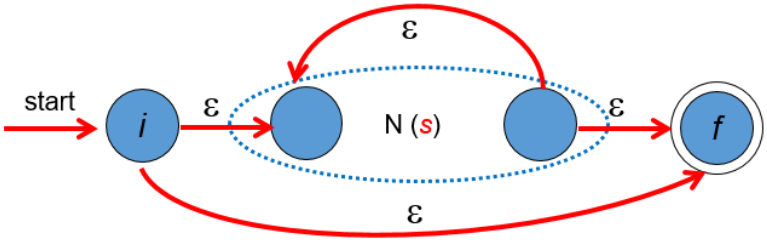
3. If r = s | t. Then N (r), the NFA for r, is constructed as: (Union operation).



4. For any regular expression r = st. Then N (r), the NFA for r, is constructed as: (concatenation operation).

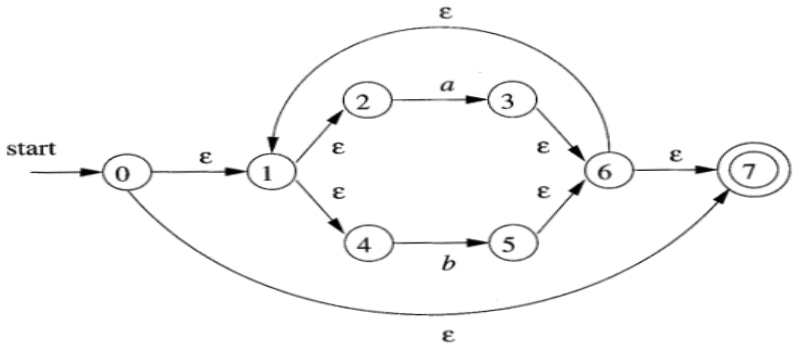


5. If r = s*. Then N (r), the NFA for r, is constructed as: (Closure operation).



Example : **Convert the following regular expression r = (a|b)* into its equivalent NFA.**

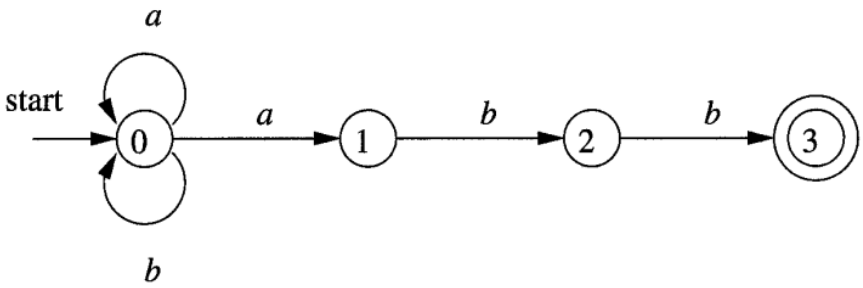
Answer:



2.7 Acceptance of Input Strings by Automata

An NFA *accepts* input string *x* if and only if there is some path in the transition graph from the start state to one of the accepting states.

Example:



Answer:

The string "aabb" is accepted by this NFA because there is a path labeled by "aabb" from state 0 to state 3.

Note that:

- There are several paths labeled by the same string that may lead to different states or the same state.
- Several paths make the string accepted.

2.8 Simulating DFA

```
s = s0;
c = nextChar();
while ( c != eof ) {
        s = move(s, c);
        c = nextChar();
}
if ( s is in F ) return "yes";
else return "no";
```

2.9 Simulating NFA

```
1)  S = ε-closure(s0);
2)  c = nextChar();
3)  while ( c != eof ) {
4)          S = ε-closure(move(S, c));
5)          c = nextChar();
6)  }
7)  if ( S ∩ F != ∅ ) return "yes";
8)  else return "no";
```

2.10 Conversion of an NFA to DFA

The subset construction algorithm based on each state of the constructed DFA corresponds to a set of NFA states.

Input:

An NFA *N*.

Output:

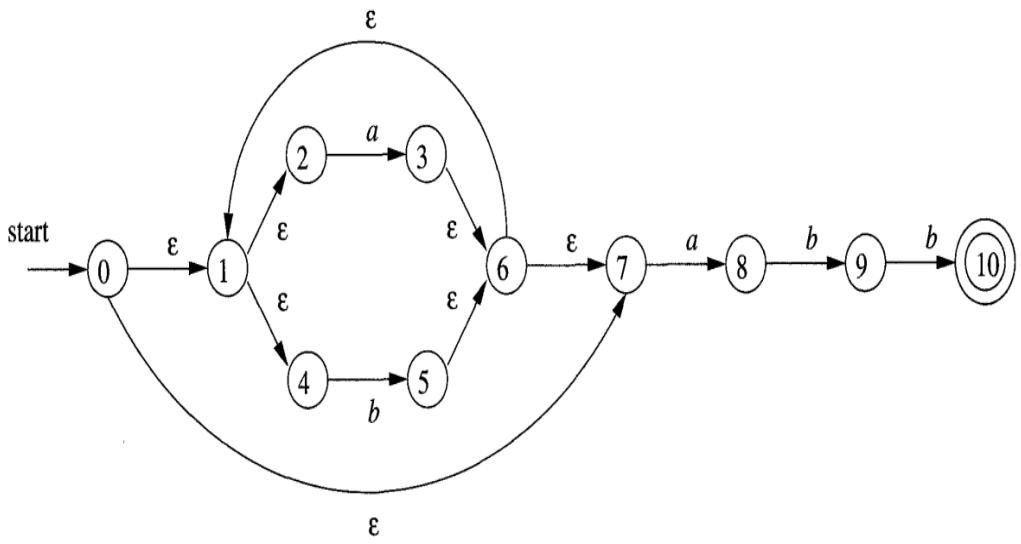
A DFA *D* accepting the same language as *N*.

Method:

This algorithm constructs a transition table D*tran* for D. Each state of D*tran* D is a set of NFA states, and through constructing D will simulate "in parallel" all possible moves N can make on a given input string. The first problem is to deal with ε- transitions of N probably. The definition of several functions is, where s is a single state, and T is a set of states in N:

OPERATION	DESCRIPTION
ϵ-*closure*(s)	Set of NFA states reachable from NFA state s on ϵ-transitions alone.
ϵ-*closure*(T)	Set of NFA states reachable from some NFA state s in set T on ϵ-transitions alone; $= \cup_{s \text{ in } T}$ ϵ-*closure*(s).
move(T, a)	Set of NFA states to which there is a transition on input symbol a from some state s in T.

Example: **Converts from NFA to DFA** by using the subset construction Algorithm.

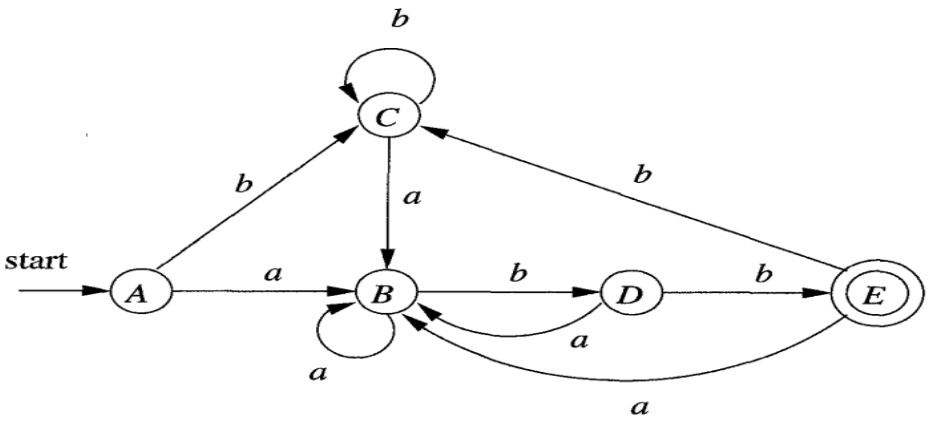


Answer:

State	a	b
0	ø	ø
1	ø	ø
2	{3}	ø
3	ø	ø
4	ø	{5}
5	ø	ø
6	ø	ø
7	{8}	ø
8	ø	{9}
9	ø	{10}
10	ø	ø

State	a	b	Ɛ
0	ø	ø	{1, 7}
1	ø	ø	{2, 4}
2	{3}	ø	ø
3	ø	ø	{6}
4	ø	{5}	ø
5	ø	ø	{6}
6	ø	ø	{1, 7}
7	{8}	ø	ø
8	ø	{9}	ø
9	ø	{10}	ø
10	ø	ø	ø

DFA Set		a		b	
{0,1,2,4,7}	A	{1,2,3,4,6,7,8}	B	{1,2,4,5,6,7}	C
{1,2,3,4,6,7,8}	B	{1,2,3,4,6,7,8}	B	{1,2,4,5,6,7,9}	D
{1,2,4,5,6,7}	C	{1,2,3,4,6,7,8}	B	{1,2,4,5,6,7}	C
{1,2,4,5,6,7,9}	D	{1,2,3,4,6,7,8}	B	{1,2,4,5,6,7,10}	E
{1,2,4,5,6,7,10}	E F	{1,2,3,4,6,7,8}	B	{1,2,4,5,6,7}	C



2.11 Minimizing DFA

The Minimization DFA Algorithm

- **Input:**

  A DFA Machine.

- **Output:**

  A new DFA after minimization.

- **Method:**

1. Start with an initial partition II with two groups, F and S - F, the accepting and non-accepting states of D.
2. Apply these steps to construct a new partition anew.
3. initially, let II new= II;

   for *(* each group G of II)

   {

   partition G into subgroups such that two states s and t are in the same subgroup if and only if for all input symbols a, states s and t have transitions on a to states in the same group of II;

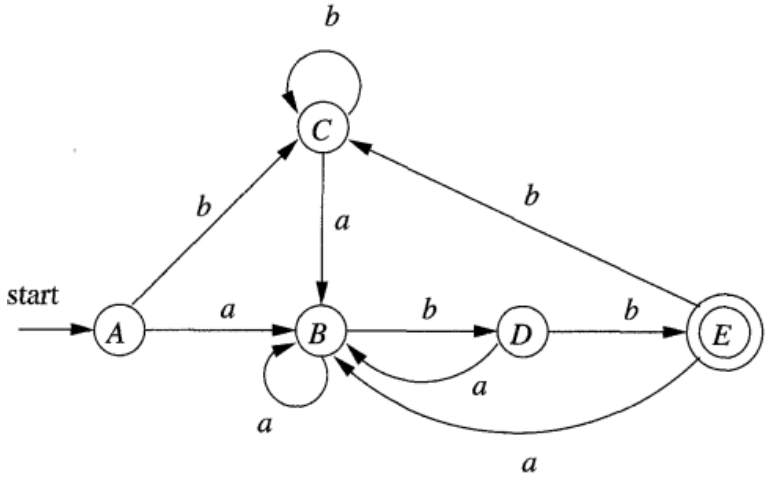
   /* at worst, a state will be in a subgroup by itself */

   replace G in IInew by the set of all subgroups formed;

   }
4. If II new, = II, let IIfinal= II and continue with step (4). Otherwise, repeat step (2) with IInew in place of II.
5. Choose one state in each group of IIfinal as the representative for that group. The representatives will be the states of the minimum-state DFA *D'*.

Example: Minimizing the following DFA:

(a)



Answer:

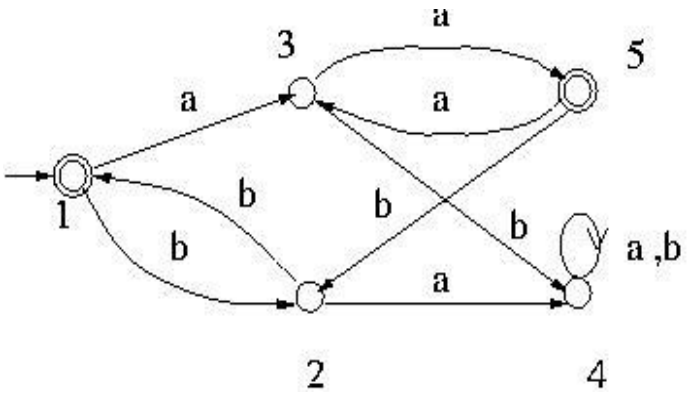
IInew = {E}, {A,B,C,D}

{A,B,C,D} group split into {A,C}, {D}, {B}

II final = {A,C}, {D}, {B}, {E}

STATE	*a*	*b*
A	*B*	*A*
B	*B*	*D*
D	*B*	*E*
E	*B*	*A*

(b)



Answer:

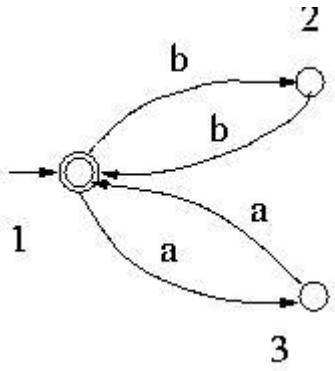
II new = {1,5} , {2,3,4}

{2,3,4} group split into {2}, {3}, {4}

II final = {1,5} , {2}, {3}, {4}

Note:

We remove the dead states.



2.12 Chapter Exercises

1. **Divide the following C++ program into appropriate lexemes:**

```
float limitedSquare(x) float x {
    /* returns x-squared, but never more than 100 */
    return (x<=-10.0||x>=10.0)?100:x*x;
}
```

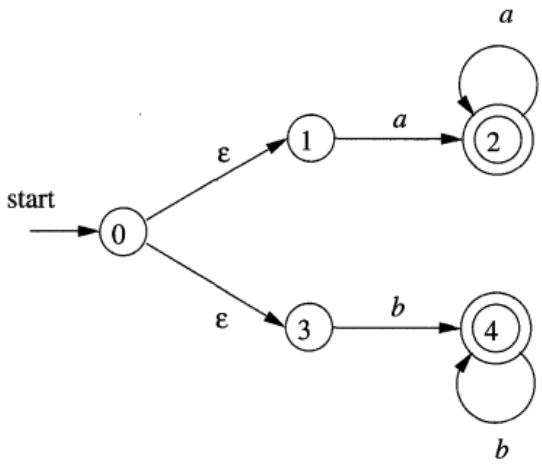
2. **Describe the languages denoted by the following regular expressions:**

    1. **((e | a) b *) ***
    2. **(a | b) * a (a | b) (a | b)**
    3. **a * ba * ba * ba ***

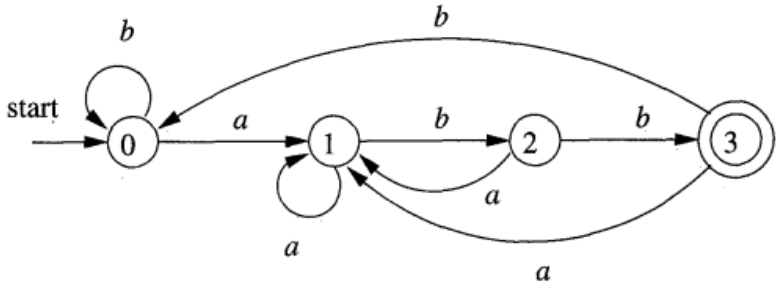
3. **Find the regular expressions of the language that have all integer numbers that start and end with zero.**
4. **Find the regular expressions for all strings over the alphabet {a,b} that contains exactly two a`s.**
5. **Find the regular expressions for a float number.**

6. **Find the transition tables for the NFA:**

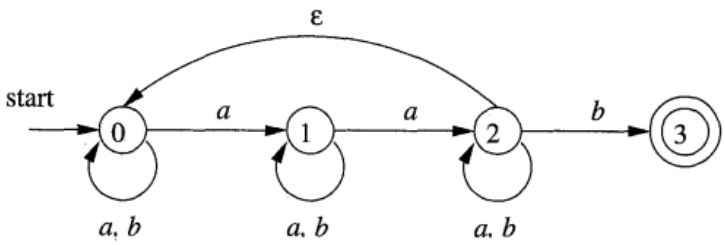
(a)



(b)



(c)



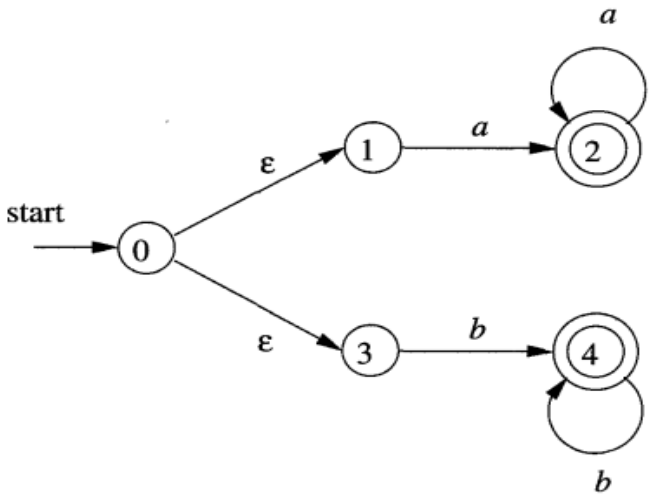
7. Convert the following regular expressions to deterministic finite automata, using algorithms:

A. **(a|b)* abb (a|b)***

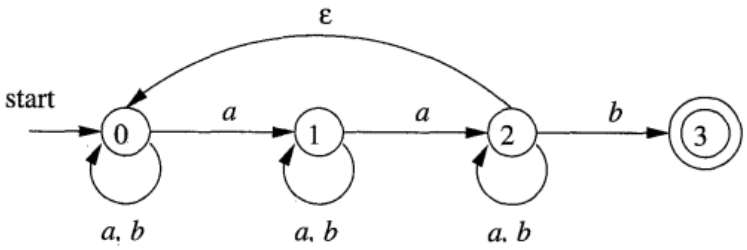
B. **a | a* b**

8. Convert from NFA to DFA:

(a)

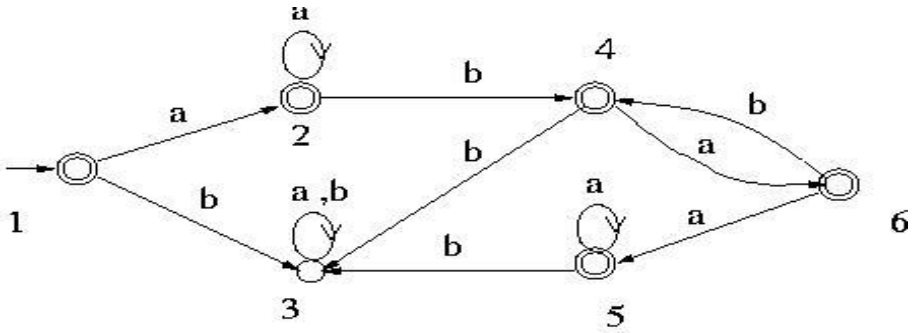


(b)

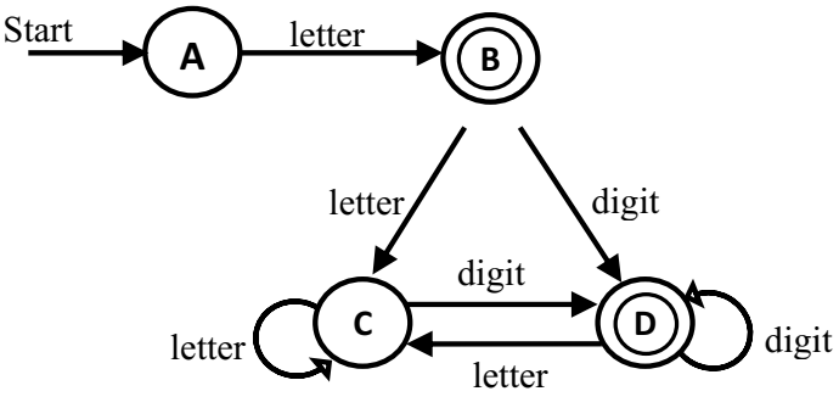


9. Minimizing the Number of States of a DFA

(a)



(b)


Start
A
letter
B
letter
digit
digit
C
D
letter
digit
letter

Chapter 3

Syntax Analysis

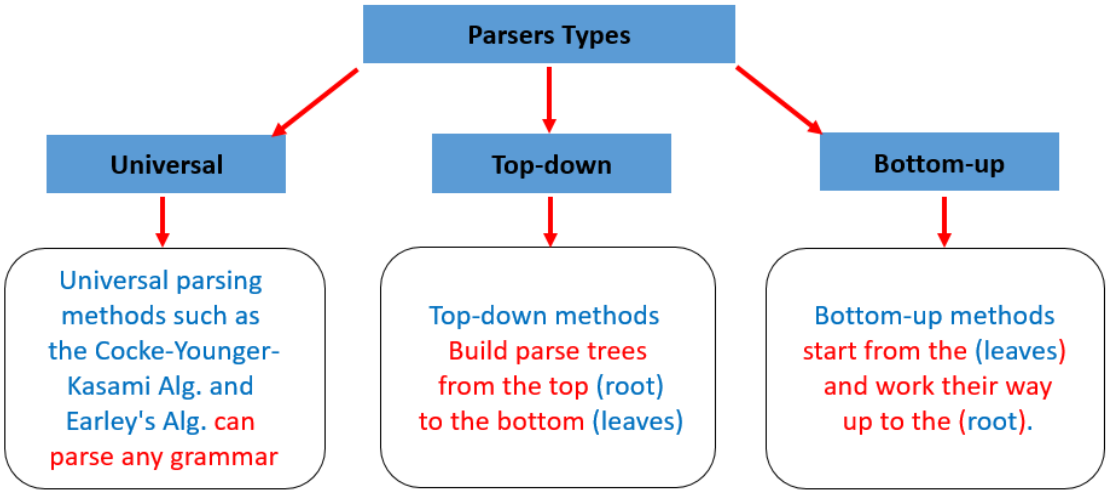
Part I

This chapter explains the parsing methods that are typically used in compilers.

The syntax of programming language constructs can be specified by context-free grammar. Generally, grammars offer significant benefits for both language designers and compiler writers. Grammar allows a language to be evolved or developed iteratively, by adding new constructs to perform new tasks.

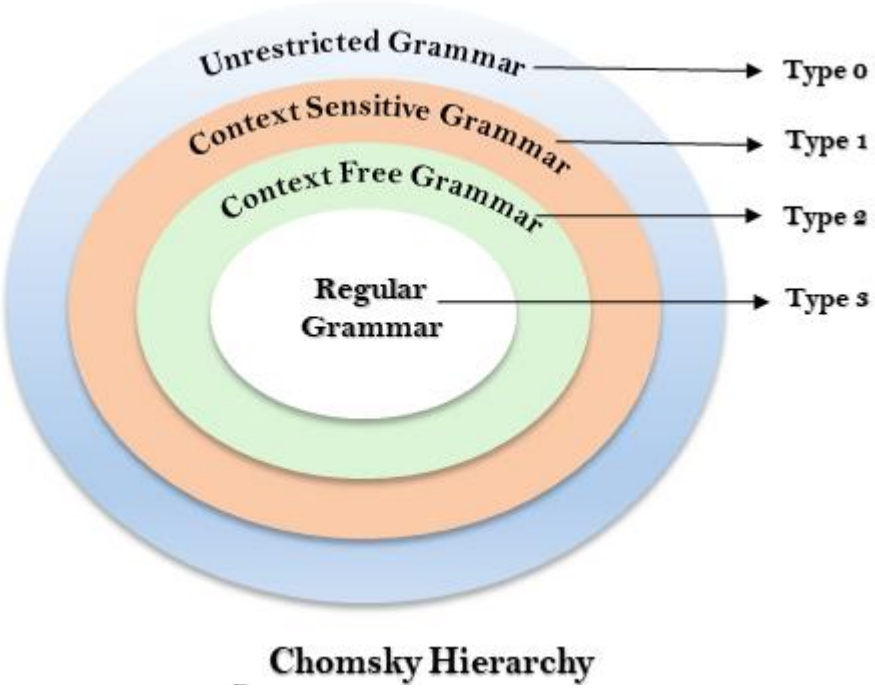
3.1 The Parsers Types

There are three general types of parsers for grammar: universal, top-down, and bottom-up.



3.2 Grammars

The classification of grammars:



3.2.1 Regular Definitions

If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form:

$$
\begin{aligned}
d_1 &\rightarrow r_1 \\
d_2 &\rightarrow r_2 \\
&\cdots \\
d_n &\rightarrow r_n
\end{aligned}
$$

Example: Find the regular definition of C- identifiers programming language?

Answer:

$$
\begin{aligned}
letter\_ &\rightarrow \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{\_} \\
digit &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
id &\rightarrow letter\_ \; (\; letter\_ \mid digit \;)^*
\end{aligned}
$$

Another Example: Find the regular definition of the Unsigned numbers (integer or floating-point)

Answer:

$$
\begin{aligned}
digit &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
digits &\rightarrow digit \; digit^* \\
optionalFraction &\rightarrow \texttt{.} \; digits \mid \epsilon \\
optionalExponent &\rightarrow (\; \texttt{E} \; (\; \texttt{+} \mid \texttt{-} \mid \epsilon \;) \; digits \;) \mid \epsilon \\
number &\rightarrow digits \; optionalFraction \; optionalExponent
\end{aligned}
$$

3.2.2 Context-Free Grammars

Context-free grammar consists of four components:

1. A set of terminal symbols called **T**.
2. A set of nonterminals called **N**.

3. A set of productions called **P**.

4. A designation of one of the nonterminals as the ***start symbol*** is usually called **S**.

Example:

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

T={ +, -, *, /, (,), id }

N = {E, T, F}

Start symbol = E

The production rule P of the Context Free Grammar is in the form :

$$A \rightarrow \alpha$$

$A \in N$ and $\alpha \in (N \cup T)^*$

P likes : E → E + T | E – T

F → (E)

Another Example:

S -> B | aBc | ε

B -> b | ε

3.3 Derivations

Derivations are used to determine a language or set of legal strings of tokens by using grammar rules.

Types of Derivations:

1) *Left Most Derivation*: Always chooses the leftmost non-terminal in the derivation process.
2) *Right Most Derivation*: Always chooses the rightmost non-terminal.

In the Derivation Process grammar derives strings by beginning with the start symbol and repeatedly replacing a nonterminal with the body of production for the nonterminal.

Example: Consider the following context-free grammar :

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

Generates the string " (id+id) " by using Left Most and Right Most Derivation

The Left Most Derivation:

$$E \Rightarrow E \Rightarrow (E) \Rightarrow (E + E) \Rightarrow (\mathrm{id} + E) \Rightarrow (\mathrm{id} + \mathrm{id})$$

The Right Most Derivation:

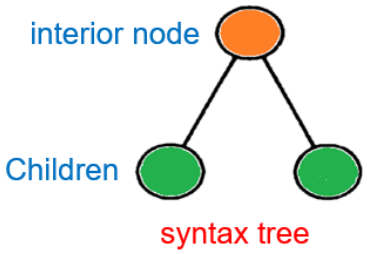
$$E \Rightarrow E \Rightarrow (E) \Rightarrow (E + E) \Rightarrow (E + \mathrm{id}) \Rightarrow (\mathrm{id} + \mathrm{id})$$

Combing Grammars:

1) **Union: L=M U N** **G: S → S_1 | S_2**
2) **Production: L=M.N** **G: S → S_1 . S_2**
3) **Closure: L= k*** **G: S → S_1 . S | ε**

3.4 Parsing

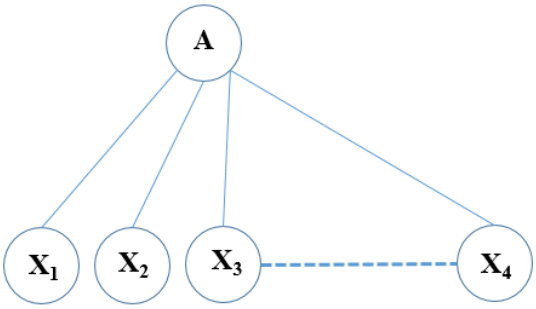
The parsing is the task of determining the syntax or the structure of a program. It is the process of determining how a string of terminals can be generated by grammar.



Parse Tree: is a tree with internal nodes represented by a non-terminal and leave-node represented by a terminal.

The parse tree is a tree with the following properties:

1) **Root label by start symbol.**
2) **Each leaf is labeled by terminal or ε.**
3) **Each internal node is labeled by non-terminal**
4) **If the production rule has the form: A → X_1 X_2 ..... X_n can be represented by:**



Example: Find the tree of the string: " – (id+id) "

Answer:

Derivation is :

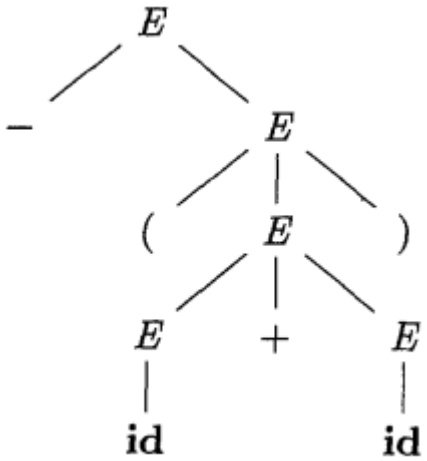
Rightmost:

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

Leftmost:

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

Parse tree:



3.5 Ambiguity

For some strings there exist more than one parse tree or more than leftmost derivation and more than rightmost derivation.

The **Grammar** is called **Ambiguous Grammar** if and only if it produces **more** than **one parse tree** for the **same string**.

Important Notes:

- There is no general algorithm that tells whether context-free grammar is ambiguous or not.
- There is no standard procedure for eliminating ambiguity, So In general, we try to eliminate ambiguity by rewriting the grammar.

- Every ambiguous grammar fails to be LR Grammar.

Example:

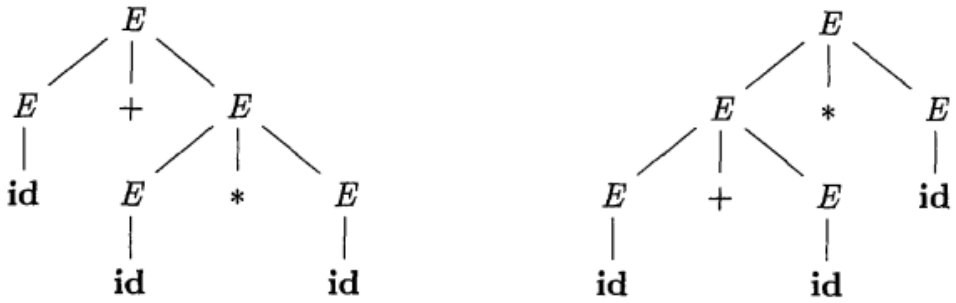
$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

Answer:

This grammar permits two distinct leftmost derivations for the sentence id +id*id:

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

The corresponding parse trees for this string are:



So, this **grammar** is ***ambiguous***.

3.6 Eliminating Ambiguity

The are many problems that cause the ambiguity of grammar like :

1. left recursion problem and

2. left factoring problems.

3.6.1 Eliminating Recursion Problem

A **grammar** is ***left recursive*** if it has a nonterminal **A** that appears in the first of the right-hand side of the production rule. This grammar in this form may cause endless recursion.

Top-down parsing methods cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion.

- **Left Recursion has 3 Types:**
    1. **Direct ($A \rightarrow A\alpha$).**
    2. **Indirect ($A \rightarrow Bc$, $B \rightarrow A$).**
    3. **Hidden ($A \rightarrow BA$, $B \rightarrow \epsilon$).**

1. **<u>Elimination of Direct Left Recursion:</u>**

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

$$\Downarrow$$

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \varepsilon$$

The General Form of eliminating the direct left recursion:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

$$\Downarrow$$

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$$

Example: **Eliminate the left recursion in the following grammar:**

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

Answer:

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

2. Elimination of Indirect Left Recursion:

The Following Algorithm to eliminate left recursion from a grammar:

- **Input:**

  An Ambigous Grammar.

- **Output:**

  A new Grammar after Eliminating Left Recursion.

  An equivalent grammar with no left recursion

Method:

1) arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.

2) ***for (each i from 1 to n) {***

3***)*** ***for (each j from 1 to i - 1) {***

4) replace each production of the form $A_i \rightarrow A_j \gamma$ by the productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \delta_3 \gamma \mid \ldots\ldots\ldots \mid \delta_k \gamma$,where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \delta_3 \mid \ldots\ldots\ldots \mid \delta_k$ are all current A_j-productions

5***) }***

6) eliminate the immediate left recursion among the A_i-productions

7) **}**

Example: Eliminate left recursion:

S → E

E → E+T

E → T

T → E-T

T → F

F → E*F

F → id

Answer:

S → E

E → TE′

E′ → +TE′ | e

T → FT′

T′ → E′ -T T′ | e

F → id F′

F′ → T′ E′ *F F′ | e

3.6.2 Eliminating Left Factoring Problem

We need to eliminate left factoring from the grammar because we need to know exactly the production rule in each iteration in top-down parsing.

The general form for eliminating the left factoring:

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

$$\Downarrow$$

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2$$

Example: **Solve The left factoring problem in the following grammar:**

$$S \rightarrow i\ E\ t\ S \mid i\ E\ t\ S\ e\ S \mid a$$
$$E \rightarrow b$$

Answer:

$$S \rightarrow i\ E\ t\ S\ S' \mid a$$
$$S' \rightarrow e\ S \mid \epsilon$$
$$E \rightarrow b$$

3.7 Top-Down Parsing (T.D.P)

T.D.P is the process of constructing a parse tree for the input string by starting from the root and creating the nodes of the parse tree preorder.

T.D.P can be viewed as finding a leftmost derivation for an input string.

Top-down parsing comes in two forms:

1. ***Recursive Descent Parsing*** (Back Tracking Parser).

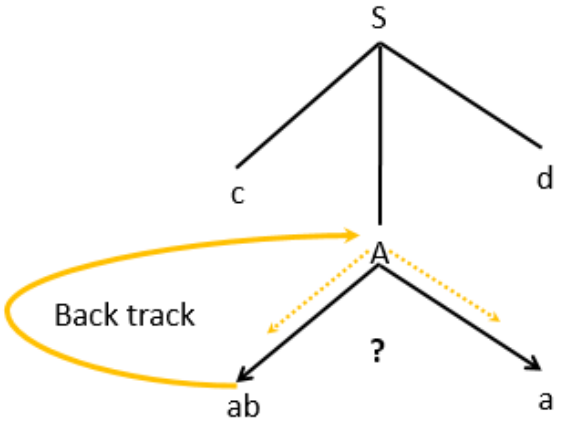
   In Recursive Descent Parsing, we may need to backtrack To find the correct production rule that applied.

Example:

S → cAd

A → ab | a

With the word: "cad" find T.D.P.



The idea of recursive descent parsing is a set procedure of each non-terminal i.e terminal -> match & Non-terminal -> method.

Recursive-Descent Parsing Algorithm:

```
    void A() {
1)      Choose an A-production, A → X1X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi is a nonterminal )
4)              call procedure Xi();
5)          else if ( Xi equals the current input symbol a )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```

Example:

S → aSa | b

S( )

{

if (next_token ==`a`)

match(a);

S( )

match(a);

else if (next_token ==`b`)

match(b);

else

error ( );

}

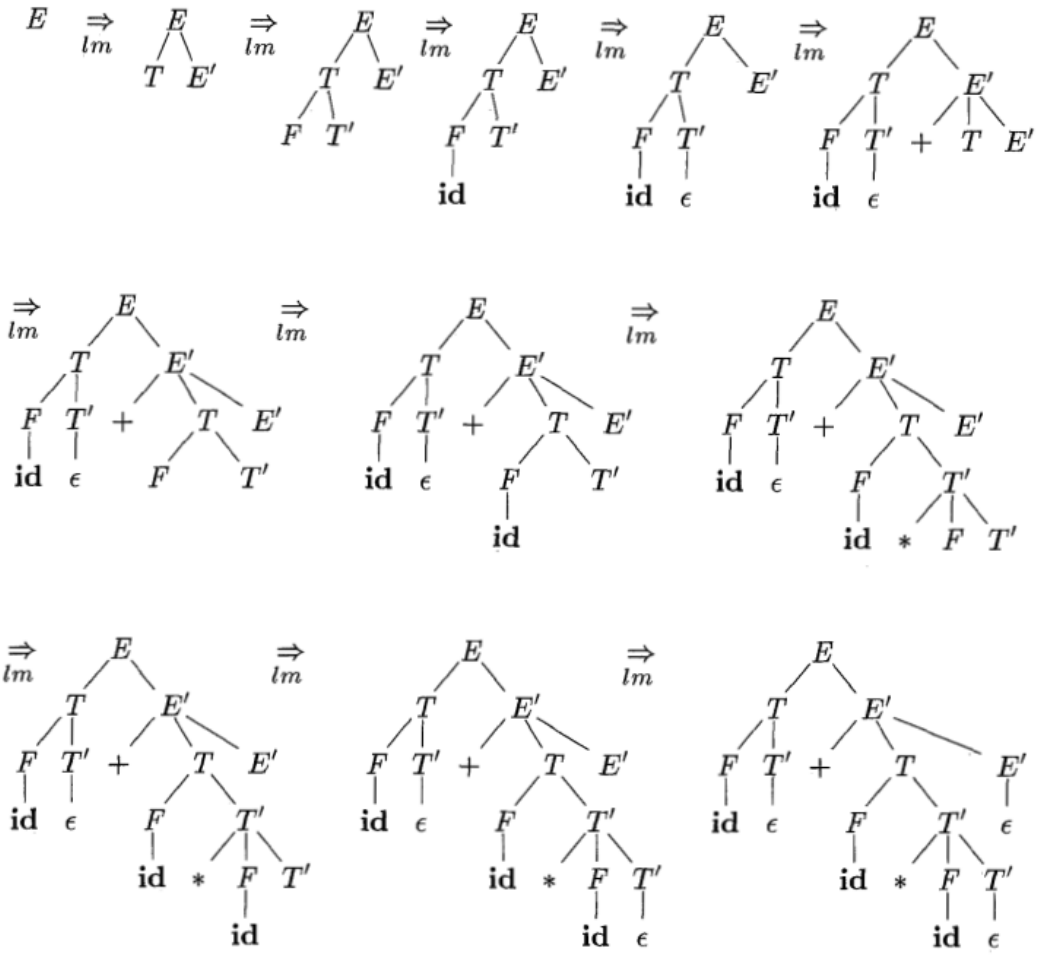
2. Predictive Parsing

In predictive parsing, there is no backtracking, So, this method required choose the correct production by looking ahead at the input fixed number of symbols.

Example: **Find parse tree for (id+id*id) from the following grammar:**

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

Answer:



Note:

- LL(1) is a special case of recursive descent parsing.

3.8 First and Follow

The construction of the top-down parsing depends on two functions "FIRST" and "FOLLOW".

1. Finding FIRST:

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

1. If X is a terminal, then $\text{FIRST}(X) = \{X\}$.
2. If X is a nonterminal and $X \to Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in $\text{FIRST}(X)$ if for some i, a is in $\text{FIRST}(Y_i)$, and ϵ is in all of $\text{FIRST}(Y_1), \ldots, \text{FIRST}(Y_{i-1})$; that is, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in $\text{FIRST}(Y_j)$ for all $j = 1, 2, \ldots, k$, then add ϵ to $\text{FIRST}(X)$. For example, everything in $\text{FIRST}(Y_1)$ is surely in $\text{FIRST}(X)$. If Y_1 does not derive ϵ, then we add nothing more to $\text{FIRST}(X)$, but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add $\text{FIRST}(Y_2)$, and so on.
3. If $X \to \epsilon$ is a production, then add ϵ to $\text{FIRST}(X)$.

2. Finding FOLLOW:

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 →α B β, then everything in FIRST(β) except ε is in FOLLOW(B).
3. If there is a production A →α B , or a production A →α B β, where FIRST(β) contains ε, then everything in FOLLOW (A) is in FOLLOW(B).

Example: Find the FIRST and the FOLLOW:

S → cAd
A→ ab | a

Answer:

Symbol x	First(X)
S	c
cAd	c
A	a
a	a
c	c
b	b

Non-Terminal x	Follow(X)
S	{$}
A	{d}

Example: Find the FIRST and FOLLOW for the following grammar:

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

Answer:

Symbol x	First(X)	Symbol x	First(X)
E	{(, id}	+TE'	{+}
E'	{+, ε }	(E)	{(}
T	{(, id}	FT'	{(, id}
T'	{*, ε }	*FT'	{*}
F	{(, id}	TE'	{(, id}

Non-Terminal x	Follow(X)
E	{$,)}
E'	{$,)}
T	{$,), +}
T'	{$,), +}
F	{$,), +,*}

3.9 Chapter Problems

- **Write regular definitions for the following languages:**
  i. **All strings of lowercase letters that contain the five vowels are in order.**
  ii. **Comments, consisting of a surrounded by / * and * / .**

- **Find Grammar for the following languages:**

1) **L={ ϵ , a , aa , aaa , ……}**
2) **L={bb, bbbb, bbbbbb,…..}**
3) **L={awa | w ϵ {a,b}*}**
4) **L={ϵ, a, aa , aaa,…}U {ϵ , b , bb , bbb,..}**
5) **L={ a , ϵ , b , bb, bbb, ……}**
6) **L={$a^n b^m$: n ≥ 1,m ≥1}**

- **Consider the context-free grammar: S → SS+ | SS* | a. and the string aa +a*.**

  a) Give the leftmost derivation for the string.

  b) Give the rightmost derivation for the string.

  c) Give a parse tree for the string.

  d) Is the grammar ambiguous or unambiguous? Justify your answer.

  e) Describe the language generated by this grammar.

- **Consider the context-free grammar: S → 0S1 | 01 the string 000111.**

  a) Give a leftmost derivation for the string.

  b) Give the rightmost derivation for the string.

  c) Give a parse tree for the string.

  d) Is the grammar ambiguous or unambiguous? Justify your answer.

  e) Describe the language generated by this grammar.

- **Design grammar for the following languages:**

  a) The set of all strings of 0s and 1s such that every 0 is immediately followed

  by at least one 1.

  b) The set of all strings of 0s and 1s that are palindromes; that is, the string

  reads the same backward as forward.

- **Eliminate left recursion:**

  1. $S \rightarrow S+a \mid b$.

  2. $S \rightarrow S(S)S \mid \epsilon$.

- **Eliminate left recursion:**

  S → Aa | b

  A → Ac

  A → Sd | e

- **Eliminate Left Factoring for the grammar:** $S \rightarrow aa \mid ab \mid a$.

- **Compute First and Follow for the following grammar:**

  S → ASb | C

  A → a

  S → c C | ε

Chapter 4

Syntax Analysis

Part II

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

4.1 LL(1) Grammars

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

A grammar G is LL(1) if and only if whenever A → α | β are two distinct productions of G, the following conditions hold:

1. For no terminal a do both α and β derive strings beginning with a.
2. At most one of α and β can derive the empty string.

3. If β →* ε, then α does not derive any string beginning with a terminal in FOLLOW(A). Likewise, if α →* ε, then β does not derive any string beginning with a terminal in FOLLOW(A).

Properties of LL(1) Grammars:

1. **No left-recursive** grammar is **LL(1)**.
2. **No ambiguous** grammar is **LL(1)**.
3. Some languages have no **LL(1)** grammar.
4. For every LL(1) grammar, each parsing table entry uniquely identifies a production or signals an error.

LL(1) Parsing Table

Construction of a Predictive Parsing Table Algorithm:

- **Input:**

  A Grammar G.

- **Output:**

  A Parse Table M.

- **Method: for each production A → α of the grammar do the following:**

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

FOLLOW(A), add A → α to M[A,$] as well. If after performing the above, there is no production at all in M[A, a], then set [M, a] to **error**.

Example: Produce the Parse Table for the Following Grammar:

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

Answer:

X	First(X)	X	First(X)
id	{id}	T	{(, id}
(E)	{(}	+TE'	{+}
F	{(, id}	E'	{+, ε}
*FT'	{*}	TE'	{(, id}
T'	{*, ε}	E	{(, id}
FT'	{(, id}		

X	Follow(X)
E	{$,)}
E'	{$,)}
T	{$,), +}
T'	{$,), +}
F	{$,), +,*}

NON - INPUT SYMBOL	TERMINAL					
	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

Example: **Devise predicative parsers and show the parsing tables for each of the following grammars:**

a) $S \rightarrow 0S1 \mid 01$.

Answer:

S → 0A

A → S1 | 1

X	First(X)
0	{0}
1	{1}
S	{0}
A	{0,1}

X	Follow(X)
S	{$,1}
A	{$,1}

Input Symbols			
Non-Terminal	**0**	**1**	**$**
S	**S → 0A**	**error**	**error**
A	**A → S1**	**A→ 1**	**error**

4.2 Predictive parsing Algorithm:

Table-driven predictive parsing Algorithm:

- **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.

- **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 *$*. The uses the predictive parsing table M to produce a predictive parse for the input

```
set ip to point to the first symbol of w;
set X to the top stack symbol;
while ( X ≠ $ ) { /* stack is not empty */
        if ( X is a ) pop the stack and advance ip;
        else if ( X is a terminal ) error();
        else if ( M[X, a] is an error entry ) error();
        else if ( M[X, a] = X → Y1Y2 ··· Yk ) {
                output the production X → Y1Y2 ··· Yk;
                pop the stack;
                push Yk, Yk−1, ... , Y1 onto the stack, with Y1 on top;
        }
        set X to the top stack symbol;
}
```

Example: Find the Table driven predictive parsing for the following string: id+id*id.

Answer:

NON - INPUT SYMBOL	TERMINAL					
	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

Matched	Stack	Input	Action
	E$	id+id*id$	
	TE'$	id+id*id$	Output: *E* → TE'
	FT'E'$	id+id*id$	Output: *T* → FT'
	idT'E'$	id+id*id$	Output: *F* → id
id	T'E'$	+id*id$	match id
id	E'$	+id*id$	Output: T' → ϵ
id	+TE'$	+id*id$	Output: *E'* → +TE'
id+	TE'$	id*id$	match +
id+	FT'E'$	id*id$	Output: *T* → FT'
id+	idT'E'$	id*id$	Output: *F* → id
id+id	T'E'$	*id$	match id
id+id	*FT'E'$	*id$	Output: T' → *FT'
id+id*	FT'E'$	id$	match *
id+id*	idT'E'$	id$	Output: *F* → id
id+id*id	T'E'$	$	match id
id+id*id	E'$	$	Output: T' → ϵ
id+id*id	$	$	Output: E' → ϵ

So, **id+id*id** is an **Accepted string.**

Example: **Generates the Table driven predictive parsing for the string: id*+id.**

Answer:

NON - INPUT SYMBOL	TERMINAL					
	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

Matched	Stack	Input	Action
	E$	)id*+id$	error , skip ')'
	E$	id*+id$	Output: E → TE'
	TE'$	id*+id$	Output: T → FT'
	FT'E'$	id*+id$	Output: F → id
	idT'E'$	id*+id$	match id
id	T'E'$	*+id$	Output: T' → *FT'
id	*FT'E'$	*+id$	match *
id*	FT'E'$	+id$	error , remove Nonterminal from Stack
id*	T'E'$	+id$	Output: T' → ε
id*	E'$	+id$	Output: E'→ +TE'
id*	+TE'$	+id$	match +
id*+	TE'$	id$	Output: T → FT'
id*+	FT'E'$	id$	Output: F → id
id*+	idT'E'$	id$	match id
id*+id	T'E'$	$	Output: T' → ε
id*+id	E'$	$	Output: E' → ε
id*+id	T'$	$	
id*+id	$	$	

4.3 Error Recovery in Predictive Parsing

An error is detected during the predictive parsing when the terminal on top of the stack does not match the next input symbol, or when nonterminal A on top of the stack, a is the next input symbol, and parsing table entry M[A, a] is empty.

The error recovery in predictive parsing using the following two methods:

1. Panic Mode Recovery
2. Phrase Level Recovery

4.3.1 Panic Mode

Panic Mode is an error recovery that is based on the idea of skipping symbols from input until a token in a selected set of synchronizing tokens appears.

Some points to follow are:

1. Remove the terminal from the string.
2. Introduce the nonterminal by that terminal can be produced.
3. Pop non-terminal from the stack.

Error Recovery Steps:

- As a starting point, place all symbols in FOLLOW(A) into the synchronizing set for nonterminal A. If we ***skip tokens until*** an element of ***FOLLOW(A)*** is seen and ***pop A*** from the stack, parsing can likely continue.
- It is *not enough* to use FOLLOW(A) as the synchronizing set for A.

Note:

- **Synchronizing Tokens: Synch added to the parsing table at the FOLLOW of Non-terminal.**

Example:

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ id			$F \to (E)$		

Answer:

X	Follow(X)
E	{$,)}
E'	{$,)}
T	{$,), +}
T'	{$,), +}
F	{$,), +, *}

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E \to +TE'$			$E \to \epsilon$	$E \to \epsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ id	synch	synch	$F \to (E)$	synch	synch

STACK	INPUT	REMARK
E $	) id * + id $	error, skip)
E $	id * + id $	id is in FIRST(E)
TE' $	id * + id $	
$FT'E'$ $	id * + id $	
id $T'E'$ $	id * + id $	
$T'E'$ $	* + id $	
* $FT'E'$ $	* + id $	
$FT'E'$ $	+ id $	error, $M[F, +] =$ synch
$T'E'$ $	+ id $	F has been popped
E' $	+ id $	
+ TE' $	+ id $	
TE' $	id $	
$FT'E'$ $	id $	
id $T'E'$ $	id $	
$T'E'$ $	$	
E' $	$	
$	$	

4.4 Bottom-Up Parsing

A bottom-up parse 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).

The goal of bottom-up parsing is constructing the derivation in reverse which is called Reduction. It is the process of reducing the string w to the start symbol of the grammar S.

At each reduction step, a specific substring matching the body of production is replaced by the nonterminal at the head of that production.

4.4.1. Handle Pruning

- a "handle" is a substring that matches the body of a production, and whose reduction represents one step along with the reverse of a rightmost derivation.

Example: String word is: id+id*id

E → E+E
E → E*E
E → id

Answer:

LM:

E -> E*E -> E+E*E -> id+ E*E -> id+id* E -> id+id*id

RM:

E -> E*E -> E*id -> E+E*id -> E+id*id -> id+id*id

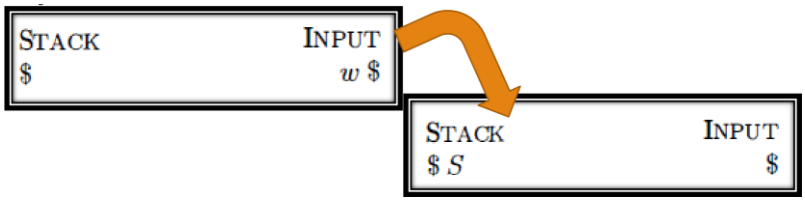
Right Sentential Form (RSF)	Handle	Reducing Production Rule
id+id*id	id	E → id
E+id*id	id	E → id
E+E*id	E+E	E → E+E
E*id	id	E → id
E*E	E*E	E → E*E
E	-	-

4.4.2. Shift-Reduce Parsing:

It is a form of bottom-up parsing in which a stack holds grammar symbols & the input buffer holds the rest of the string to be parsed.

Stack	Input	Action
$	w $	

The process is to Shift zero or more input symbols onto the stack until it is ready to reduce a string of grammar symbols on top of the stack.



- ***The Primary Operations:***

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.

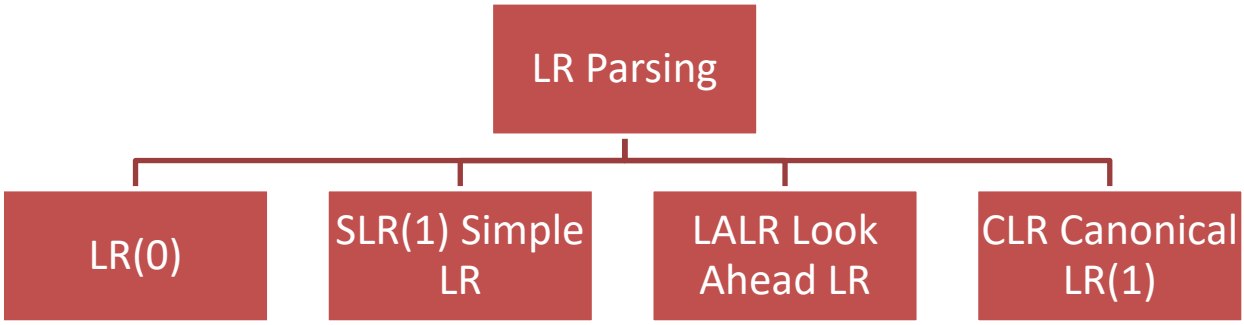
Example:

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathrm{id}
\end{aligned}
$$

Answer:

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

4.5 LR Parsing



LR Parsing

Means **L**: *Left-to-right Scanning*. & **R** *Rightmost derivation reverse*.

Bottom-up parser based on a concept called LR(k).

4.5.1. LR(0) Automation

To construct the LR(0) for grammar we need to:

1. **Augmented grammar, S` → S**
2. **Closure Function,**
3. **Go To Function,**

The acceptance occurs if and only if the parser is about to reduce by S` → S.

LR(0) items:

- For **A->XYZ** we have following items:
  1. **A->.XYZ**
  2. **A->X.YZ**
  3. **A->XY.Z**
  4. **A->XYZ.**
- Rule starts with **A->.XYZ**
- The production **A -> ϵ** generates only one item, **A -> .**

Closure of Item Sets

1. Initially, add every item in I to CLOSURE(I).

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

Example: **Is this grammar is LR or not?**

$$S \to AA$$

$$A \to aA \mid b$$

Answer:

1. **Generates Augmented Grammar**

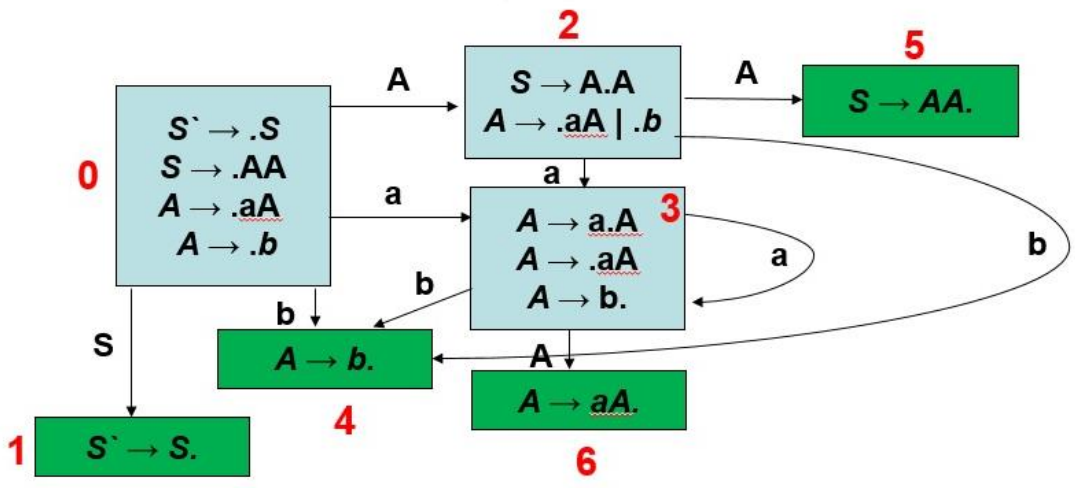
$$S` \to S$$

$$S \to AA$$

$$A \to aA$$

$$A \to b$$

2. **Create Machine**



3. Generate Action & Go To Table

		Action			Go To	
		a	b	$	S	A
	0	S3	S4	-	1	2
F	1	-	-	accept	-	-
	2	S3	S4	-	-	5
	3	S3	S4	-	-	6
F	4	r3	r3	r3	-	-
F	5	r1	r1	r1	-	-
F	6	r2	r2	r2	-	-

Check the string "aabb".

Stack	Input	Action
$ 0	aabb $	S3
$0a3	abb$	S3
$0a3a3	bb$	S4
$0a3a3b4	b$	r3: $A \rightarrow b$ Go To 6
$0a3a3A6	b$	r2: $A \rightarrow aA$ Go To 6
$0a3A6	b$	r2: $A \rightarrow aA$ Go To 2
$0A2	b$	S4
$0A2b4	$	r3: $A \rightarrow b$ Go To 5
$0A2A5	$	r1: $S \rightarrow AA$ Go To 1
$0S1	$	accept

4.5.2. LR(1)

- Any grammar LR(0) is SLR(0) grammar.
- LR(0) is not in conflict with shift or reduce.

- If exist **s/r** (shift-reduce) or **r/r** (reduce-reduce), that means it is not an LR(0).
- At LR(1) or SLR, we set with the following of non-terminal only reduce.
- **s/r** (shift-reduce): in follow of A only.
- **r/r** (reduce-reduce): in following of A & follow of B also.

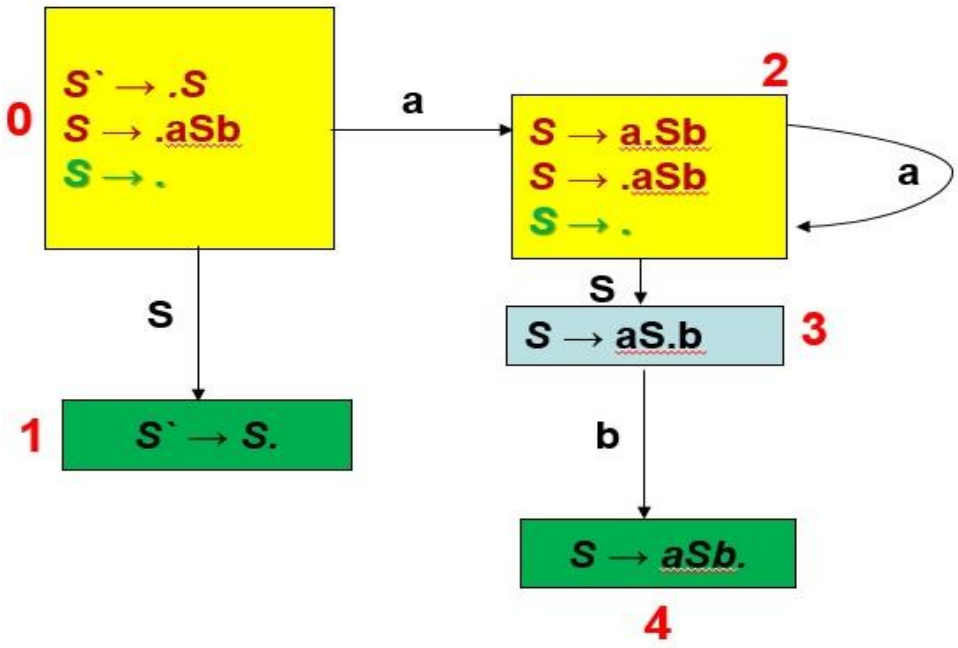
Example: Is this grammar is LR(0) or not?

$$S \rightarrow aSb \mid \epsilon$$

Answer:

Generates Augmented Grammar

$$S` \rightarrow S$$
$$S \rightarrow aSb$$
$$S \rightarrow \epsilon$$



	Action			Go To
	a	b	$	S
F 0	S2	r2	r2	1
1	-	-	accept	-
F 2	S2	r2	r2	3
3	-	S4	-	-
F 4	-	r1	r1	-

Check the string aabb.

Stack	Input	Action
$ 0	aabb $	S2
$0a2	abb$	S2
$0a2a2	bb$	r2: $S \rightarrow \epsilon$ Go To 3
$0a2a2S3	bb$	S4
$0a2a2S3b4	b$	r1: $S \rightarrow aSb$ Go To 3
$0a2S3	b$	S4
$0a2S3b4	$	r1: $S \rightarrow aSb$ Go To 1
$0S1	$	accept

4.6 Chapter Exercises:

- **Devise predicative parsers and show the parsing tables for each of the following grammars:**

$$S \rightarrow +SS \mid *SS \mid a.$$

- **Devise predicative parsers and show the parsing tables for each of the following grammars:**

$$S \rightarrow S(S)S \mid \epsilon$$

- **Check this grammar is LR(0) or not?**

$$S \rightarrow aAc$$
$$A \rightarrow Abb \mid b$$

- **Is this grammar is LR(0) or not?**

$$S \rightarrow A \mid a$$
$$A \rightarrow a$$

References

1. Compilers, Principals, techniques & tools, second edition, *Alferd V . Aho.*
2. Basics of Compiler Design , *Torben Egidius Mogensen.*
3. Compiler Design: Theory, Tools, and Examples *Seth D. Bergmann*