

Compiler Design and Construction (CSC 352)

By
Bhupendra Saud
for

*B. Sc. Computer Science & Information
Technology (Prime College)*

Course Title: Compiler Design and Construction

Course no: CSC-352

Full Marks: 70+10+20

Credit hours: 3

Pass Marks: 28+4+8

Nature of course: Theory (3 Hrs.) + Lab (3 Hrs.)

Course Synopsis: Analysis of source program. The phases of compiler.

Goal: This course introduces fundamental concept of compiler and its different phases.

Course Contents:

Unit. 1:

1.1 Introduction to compiling: Compilers, Analysis of source program, the phases of compiler, compiler-construction tools. **4 hrs.**

1.2 A Simple One-Pass Compiler: Syntax Definition, Syntax directed translation, Parsing, Translator for simple expression, Symbol Table, Abstract Stack Machines. **5 hrs**

Unit 2:

2.1 Lexical Analysis: The role of the lexical analyzer, Input buffering, Specification of tokens, Recognition of tokens, Finite Automata, Conversion Regular Expression to an NFA and then to DFA, NFA to DFA, State minimization in DFA, Flex/lex introduction. **8 Hrs.**

2.2 Syntax Analysis: The role of parser, Context free grammars, Writing a grammar, Top-down parsing, Bottom-up parsing, error recovery mechanism, LL grammar, Bottom up parsing-Handles, shift reduced parsing, LR parsers-SLR,LALR,LR,LR/LALR Grammars, parser generators. **10 Hrs.**

Unit 3:

3.1 Syntax Directed Translation: Syntax-directed definition, Syntax tree and its construction, Evaluation of S-attributed definitions, L-attributed, Top-down translation, Recursive evaluators.

5 Hrs.

3.2 Type Checking: Type systems, Specification of a simple type checker, Type conversions equivalence of type expression, Type checking Yacc/Bison.

3 Hrs.

Unit 4:

4.1 Intermediate Code Generation: Intermediate languages, three address code, Declarations, Assignments Statements, Boolean Expressions, addressing array elements, case statements, Back patching, procedure calls.

4 Hrs.

4.2 Code Generation and optimization: Issues in design of a code generator, the target machine, Run –time storage management, Basic blocks and flow graphs, next use information's, a simple code generator, Peephole organization, generating code from dags.

6 Hrs.

Third Year/Six Semester

Subject: Compiler Design and Construction

FM: 60

Time: 3 hours

PM: 24

Candidates are required to give their answer in their own words as for as practicable.

Attempt all the questions.

Every question contains equal marks.

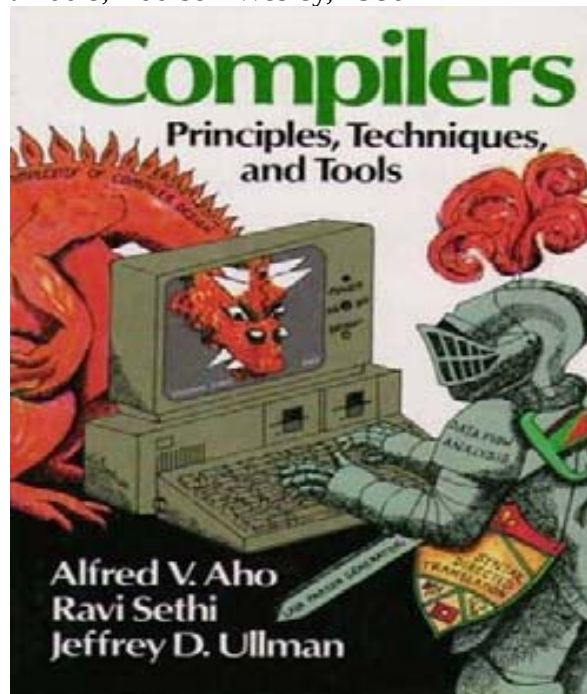
1. What do mean by compiler? How source program analyzed? Explain in brief.
2. Discuss the role of symbol table in compiler design.
3. Convert the regular expression '0+ (1+0)*00' first into NFA and then into DFA using Thomson's and Subset Construction methods.
4. Consider the grammar:
 $S \rightarrow (L) | a$
 $L \rightarrow L, S | S$
5. Consider the grammar
 $C \rightarrow AB$
 $A \rightarrow a$
 $B \rightarrow a$
Calculate the canonical LR (0) items.
6. Describe the inherited and synthesized attributes of grammar using an example.
7. Write the type expressions for the following types.
 - a. An array of pointers to reals, where the array index range from 1 to 100.
 - b. Function whose domains are function from characters and whose range is a pointer of integer.
8. What do you mean by intermediate code? Explain the role of intermediate code in compiler design.
9. What is operation of simple code generator? Explain.
10. Why optimization is often required in the code generated by simple code generator?
 Explain the unreachable code optimization.

Prerequisites

- Introduction to Automata and Formal Languages
- Introduction to Analysis of Algorithms and Data Structures
- Working knowledge of C/C++
- Introduction to the Principles of Programming Languages, Operating System & Computer Architecture is plus

Resources

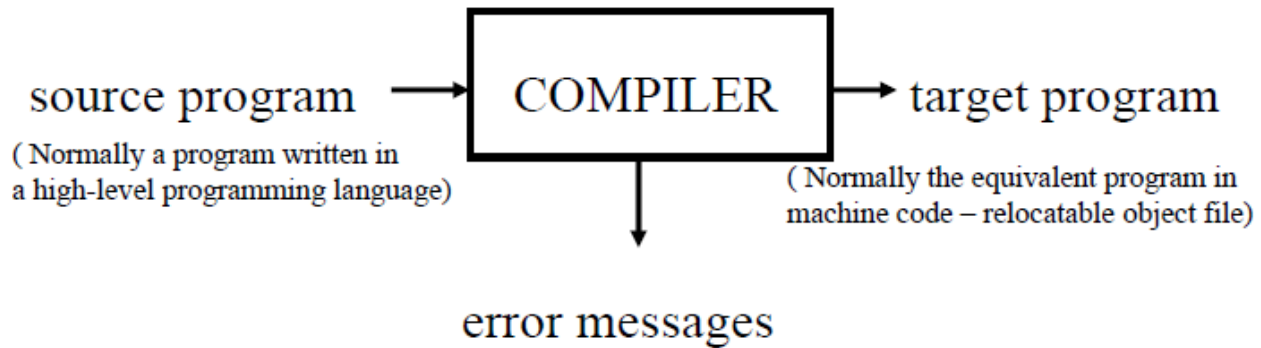
Text Book: Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986



References: Research and technical papers.

What is Compiler?

A **compiler** is a program that takes a program written in a *source language* and translates it into an equivalent low level program in a *target language*.



Phases of a Compiler

There are two major parts of a compiler: **Analysis** and **Synthesis**

– In analysis phase, an intermediate representation is created from the given source program.

This phase (Source code analysis phase) is mainly divided into following three parts:

- Lexical Analyzer
- Syntax Analyzer and
- Semantic Analyzer

– In synthesis phase, the equivalent target program is created from this intermediate representation. This phase is divided into following three parts:

- Intermediate Code Generator
- Code Optimizer and
- Final Code Generator

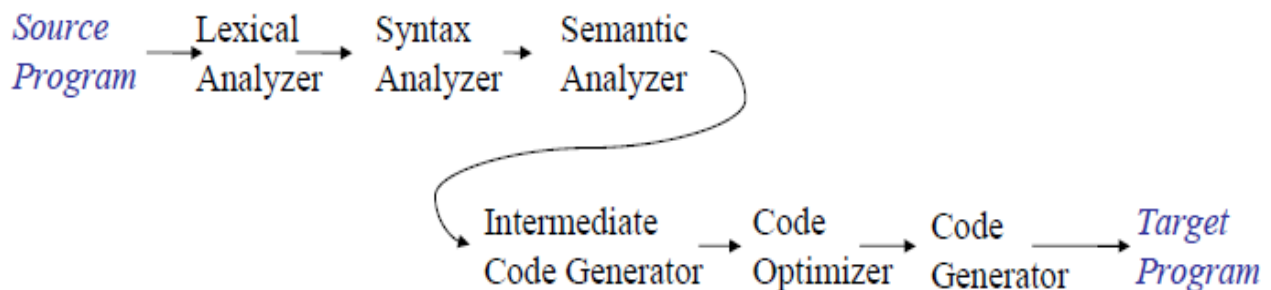


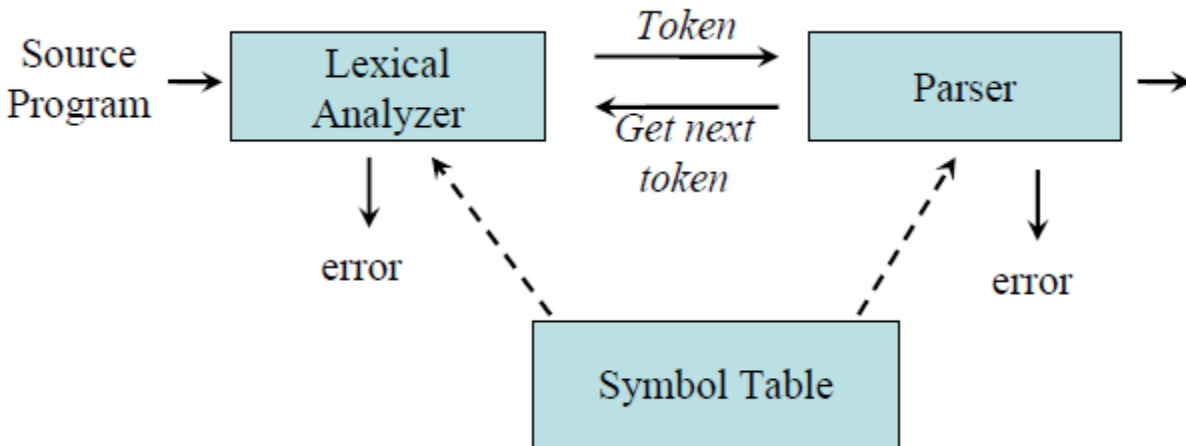
Figure: Phases of a compiler

Lexical Analyzer

Lexical Analyzer reads the source program in character by character ways and returns the *tokens* of the source program.

Normally a lexical analyzer doesn't return a list of tokens, it returns a token only when the parser asks a token from it.

Lexical analyzer may also perform other auxiliary operation like removing redundant white space, removing token separator (like semicolon) etc.

**Example:**

newval := oldval + 12

tokens: newval	identifier
:=	assignment operator
oldval	identifier
+	add operator
12	a number

Put information about identifiers into the symbol table.

Regular expressions are used to describe tokens (lexical constructs).

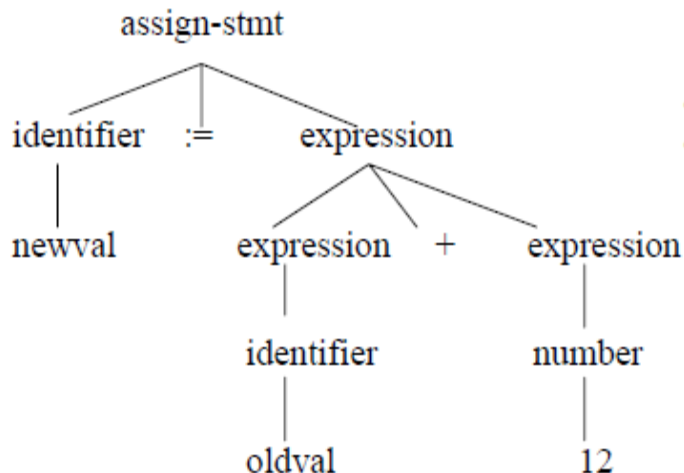
A (Deterministic) Finite State Automaton (DFA) can be used in the implementation of a lexical analyzer.

Syntax Analyzer

A **Syntax Analyzer** creates the syntactic structure (generally a parse tree) of the given source program.

Syntax analyzer is also called the **parser**. Its job is to analyze the source program based on the definition of its syntax. It works in lock-step with the lexical analyzer and is responsible for creating a parse-tree of the source code.

Ex: *newval := oldval + 12*



- In a parse tree, all terminals are at leaves.
- All inner nodes are non-terminals in a context free grammar.

The syntax of a language is specified by a **context free grammar** (CFG).

The rules in a CFG are mostly recursive.

A syntax analyzer checks whether a given program satisfies the rules implied by a CFG or not.

– If it satisfies, the syntax analyzer creates a parse tree for the given program.

Semantic Analyzer

A semantic analyzer checks the source program for semantic errors and collects the type information for the code generation.

– Type-checking is an important part of semantic analyzer.

Ex: *newval* := *oldval* + 12

- The type of the identifier *newval* must match with type of the expression (*oldval*+12)

Synthesis phase

Intermediate Code Generation

An intermediate language is often used by many compiler for analyzing and optimizing the source program. The intermediate language should have two important properties:

– It should be simple and easy to produce.

– It should be easy to translate to the target program

A compiler may produce an explicit intermediate codes representing the source program.

These intermediate codes are generally machine (architecture) independent. But the level of intermediate codes is close to the level of machine codes.

Ex:

newval := *oldval* * *fact* + 12



id1 := *id2* * *id3* + 12

temp1 = intTofloat(12)

temp2 = *id2* * *id3*

temp3 = *temp1* + *temp2*

id1 = *temp3*

Code Optimization

The process of removing unnecessary part of a code is known as code optimization. Due to code optimization process it decreases the time and space complexity of the program. i.e


- Detection of redundant function calls
- Detection of loop invariants
- Common sub-expression elimination
- Dead code detection and elimination

Ex:

```
temp1 = intTofloat (12)
temp2 = id2 * id3
temp3 = temp1 + temp2
id1 = temp3
```


```
temp1 = id2 * id3
id1 = temp1 + 12
```

```
b := 0
t1 := a + b
t2 := c * t1
a := t2
```



```
a := c*a
b := 0
```

```
a := c * a
b := 0
```



```
a := c*a
b := 0
```

Code Generation

This involves the translation of optimized intermediate code into the target language

The target code is normally is a relocatable object file containing the machine or assembly codes.

Ex:

(Assume that we have an architecture with instructions whose at least one of its operands is a machine register)

```
MOVE  id2, R1
MULT  id3, R1
ADD   #1, R1
MOVE  R1, id1
```

Cousins of Compiler

There are several major kinds of compilers:

Native Code Compiler

Translates source code into hardware (assembly or machine code) instructions.

Example: gcc.

Virtual Machine Compiler

Translates source code into an abstract machine code, for execution by a virtual machine interpreter. Example: javac.

JIT Compiler

Translates virtual machine code to native code Example: Sun's Hotspot java machine.

Preprocessor

Translates source code into simpler or slightly lower level source code, for compilation by another compiler. Examples: cpp, m4.

Pure interpreter

Executes source code on the fly, without generating machine code. Example: Lisp.

Compiler Construction Tools

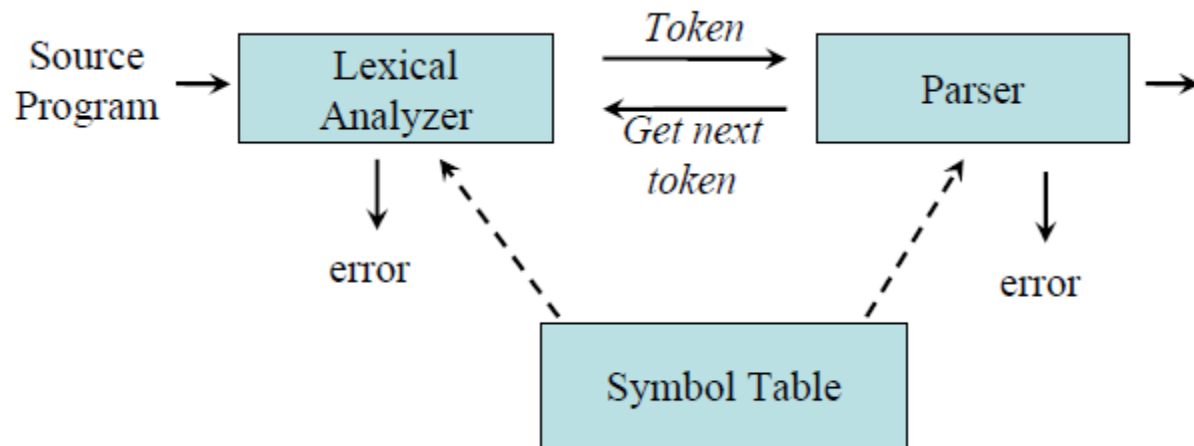
- **Scanner Generators:**
Generates lexers automatically i.e. tokenize the source program automatically.
Example: flex, lex, etc
- **Parser Generators:**
Produces syntax analyzers automatically. Example: bison, yacc etc.
- **Syntax Directed Translation Engines:**
Walks through parse tree generated by parser generator and generates intermediate code automatically. Example: bison, yacc, JavaCC etc.
- **Automatic Code Generators:**
Take optimized intermediate code and generate target code automatically.
- **Data Flow Engines:**
Optimizers using data flow analysis.

Lexical Analysis

Lexical Analyzer reads the source program in character by character ways and returns the *tokens* of the source program.

Normally a lexical analyzer doesn't return a list of tokens, it returns a token only when the parser asks a token from it.

Lexical analyzer may also perform other auxiliary operation like removing redundant white space, removing token separator (like semicolon) etc.

**Example:**

newval := *oldval* + 12

tokens: newval	identifier
:=	assignment operator
oldval	identifier
+	add operator
12	a number

Put information about identifiers into the symbol table.

Regular expressions are used to describe tokens (lexical constructs).

A (Deterministic) Finite State Automaton (DFA) can be used in the implementation of a lexical analyzer.

Tokens, Patterns, Lexemes

-A *token* is a logical building block of language. They are the sequence of characters having a collective meaning.

Eg: identifier, keywords etc

-A sequence of input characters that make up a single token is called a *lexeme*.

A token can represent more than one lexeme.

Eg: abc, 12 etc

In the statement

Float pi=3.1415

The variable pi is called a lexeme for the token 'identifier'

-Patterns are the rules for describing whether a given lexeme belonging to a token or not.

Regular expressions are widely used to specify patterns.

Attributes of Tokens

When a token represents more than one lexeme, lexical analyzer must provide additional information about the particular lexeme. This additional information is called as the *attribute* of the token.

For simplicity, a token may have a single attribute which holds the required information for that token.

Example: the tokens and the associated attribute for the following statement.

$A=B*C+2$
 <id, pointer to symbol table entry for A>
 <assig operator>
 <id, pointer to symbol table entry for B>
 <mult_op>
 <id, pointer to symbol table entry for C>
 <add_op>
 <num, integer value 2>

Input Buffering

Many times, a scanner has to look ahead several characters from the current character in order to recognize the token.

For example *int* is keyword in C, while the term *inp* may be a variable name. When the character 'i' is encountered, the scanner cannot decide whether it is a keyword or a variable name until it reads two more characters.

In order to efficiently move back and forth in the input stream, input buffering is used.

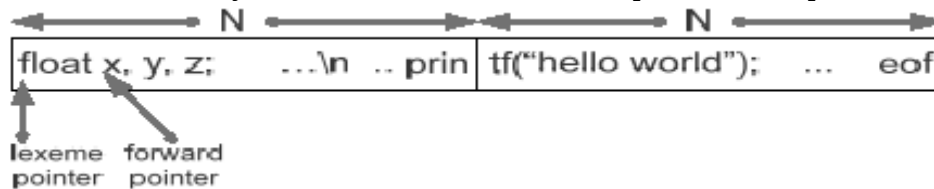


Fig: - An input buffer in two halves

Here, we divide the buffer into two halves with N-characters each.

Rather than reading character by character from file we read N input character at once. If there are fewer than N characters in input eof marker is placed.

There are two pointers (see in above fig.) the portion between lexeme pointer and forward pointer is current lexeme. Once the match for pattern is found, both the pointers points at the same place and forward pointer is moved.

The forward pointer performs tasks like below:

```

    If forward at end of first half then,
        Reload second half
        Forward++
    end if
    else if forward at end of second half then,
        Reload first half
        Forward=start of first half
    end else if
    else
        forward++
  
```

Specification of Tokens

Strings & Languages

A *string* s is a finite sequence of symbols from Σ

- $|s|$ denotes the length of string s
- ϵ denotes the empty string, thus $|\epsilon| = 0$

Operators on Strings:

- Concatenation: xy represents the concatenation of strings x and y .

$$s\epsilon = s$$

$$\epsilon s = s$$

- $S^n = s s s \dots s$ (n times) $s^0 = \epsilon$

A *language* is a specific set of strings over some fixed alphabet Σ .

Operation on Languages

Concatenation: $L_1 L_2 = \{ s_1 s_2 \mid s_1 \in L_1 \text{ and } s_2 \in L_2 \}$

Union: $L_1 \cup L_2 = \{ s \mid s \in L_1 \text{ or } s \in L_2 \}$

Exponentiation: $L^0 = \{\epsilon\}$ $L^1 = L$ $L^2 = LL$

Kleene Closure: $L^* = \cup_{i=0, \dots, \infty} L^i$

Positive Closure: $L^+ = \cup_{i=1, \dots, \infty} L^i$

Operation on Languages: Example

$L_1 = \{a, b, c, d\}$ $L_2 = \{1, 2\}$

$L_1 L_2 = \{a1, a2, b1, b2, c1, c2, d1, d2\}$

$L_1 \cup L_2 = \{a, b, c, d, 1, 2\}$

$L_1^3 =$ all strings with length three (using a, b, c, d)

$L_1^* =$ all strings using letters a, b, c, d and empty string

$L_1^+ =$ doesn't include the empty string

Regular Expressions

Regular expressions are the algebraic expressions that are used to describe tokens of a programming language.

If r and s are regular expressions denoting languages $L_1(r)$ and $L_2(s)$ respectively, then

- $r+s$ is a regular expression denoting $L_1(r) \cup L_2(s)$
- rs is a regular expression denoting $L_1(r) L_2(s)$
- r^* is a regular expression denoting $(L_1(r))^*$
- (r) is a regular expression denoting $L_1(r)$

Properties of Regular Expressions

For regular expression r, s & t

$r + s = s + r$ union is commutative)

$r + (s + t) = (r + s) + t$ (union is associative)
 $(rs)t = r(st)$ (concatenation is associative)
 $r(s + t) = rs + rt$ (concatenation distributes over union)
 $\epsilon r = r\epsilon = r$ (ϵ is the identity element for concatenation)
 $r^{**} = r^*$ (closure is idempotent)

Regular Expressions: Examples

Given the alphabet $A = \{0, 1\}$

1. $1(1+0)^*0$ denotes the language of all string that begins with a '1' and ends with a '0'.
2. $(1+0)^*00$ denotes the language of all strings that ends with 00 (binary number multiple of 4)
3. $(01)^* + (10)^*$ denotes the set of all stings that describe alternating 1s and 0s
4. $(0^*10^*10^*10^*)$ denotes the string having exactly three 1's.
5. $(11^*(0+\epsilon)11^*(0+\epsilon)11^*)$ denotes the string having at most two 0's and at least three 1's.
6. $(A|B|C|.....|Z|a|b|c|.....|z|_|) . ((A|B|C|.....|Z|a|b|c|.....|z|_|)$
 $(1|2|.....|9))^*$ denotes the regular expression to specify the identifier like in C. [TU]

Exercise:

Given the alphabet $A = \{0, 1\}$ write the regular expression for the following

1. String that either have substring 001 or 100
2. Strings where no two 1s occurs consecutively
3. String which have an odd numbers of 0s
4. String which have an odd numbers of 0s and an even numbers 1s
5. String that have at most 2 0s
6. String that at least 3 1s
7. Strings that have at most two 0s and at least three 1s

Regular Definitions

To write regular expression for some languages can be difficult, because their regular expressions can be quite complex. In those cases, we may use *regular definitions*.

The regular definition is a sequence of definitions of the form,

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

.....

$d_n \rightarrow r_n$

Where d_i is a distinct name and r_i is a regular expression over symbols in $\Sigma \cup \{d_1, d_2 \dots d_{i-1}\}$

Where, Σ = Basic symbol and

$\{d_1, d_2 \dots d_{i-1}\}$ = previously defined names.

Regular Definitions: Examples

Regular definition for specifying identifiers in a programming language like C

letter $\rightarrow A|B|C|.....|Z|a|b|c|.....|z$

underscore $\rightarrow \text{'_}'$

digit $\rightarrow 0|1|2|.....|9$

id $\rightarrow (\text{letter} | \text{underscore}).(\text{letter} | \text{underscore} | \text{digit})^*$

If we are trying to write the regular expression representing identifiers without using regular definition, it will be complex.

$(A | B | C | \dots | Z | a | b | c | \dots | z | _ |) . ((A | B | C | \dots | Z | a | b | c | \dots | z | _ |) (1 | 2 | \dots | 9))^*$

Exercise:

1 Write regular definition for specifying floating point number in a programming language like C

Solⁿ: digit $\rightarrow 0 | 1 | 2 | \dots | 9$
 num $\rightarrow \text{digit}^* (. \text{digit}^+)$

2. Write regular definitions for specifying an integer array declaration in language like C

Solⁿ: letter $\rightarrow A | B | C | \dots | Z | a | b | c | \dots | z$
 underscore $\rightarrow _$
 digit $\rightarrow 1 | 2 | \dots | 9$
 array $\rightarrow (\text{letter} | \text{underscore}) . (\text{letter} | \text{underscore} | \text{digit})^* ([\text{digit}^+ . 0^*])^+$

Recognition of Tokens

A recognizer for a language is a program that takes a string x, and answers “yes” if x is a sentence of that language, and “no” otherwise.

Recognition of token implies implementing a regular expression recognizer. That entails the implementation of Finite Automation

A finite automaton can be: deterministic (DFA) or non-deterministic (NFA)

This means that we may use a deterministic or non-deterministic automaton as a lexical analyzer.

- Deterministic – faster recognizer, but it may take more space
- Non-deterministic – slower, but it may take less space
- Deterministic automata are widely used lexical analyzers.

Design of a Lexical Analyzer

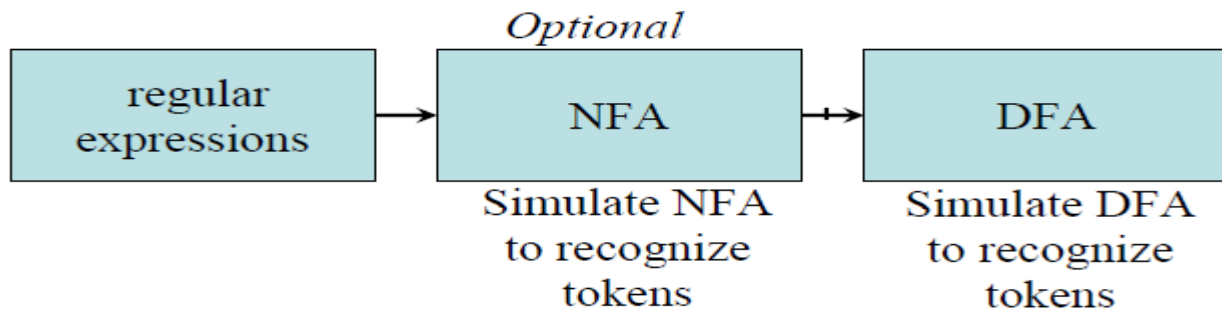
First, we define regular expressions for tokens; then we convert them into a DFA to get a lexical analyzer for our tokens.

Algorithm1:

Regular Expression \rightarrow NFA \rightarrow DFA (two steps: first to NFA, then to DFA)

Algorithm2:

Regular Expression \rightarrow DFA (directly convert a regular expression into a DFA)



Non-Deterministic Finite Automaton (NFA)

An NFA is a 5-tuple $(S, \Sigma, \delta, s_0, F)$ where

S is a finite set of *states*

Σ is a finite set of *symbols*

δ is a *transition function*

$s_0 \in S$ is the *start state*

$F \subseteq S$ is the set of *accepting (or final) states*

A NFA accepts a string x , if and only if there is a path from the starting state to one of accepting states.

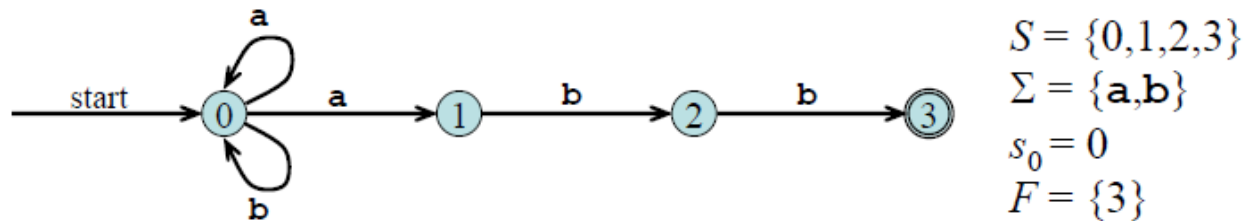


Fig: - NFA for regular expression $(a + b)^* a b b$

ϵ - NFA

In NFA if a transition made without any input symbol is called ϵ -NFA.

Here we need ϵ -NFA because the regular expressions are easily convertible to ϵ -NFA.

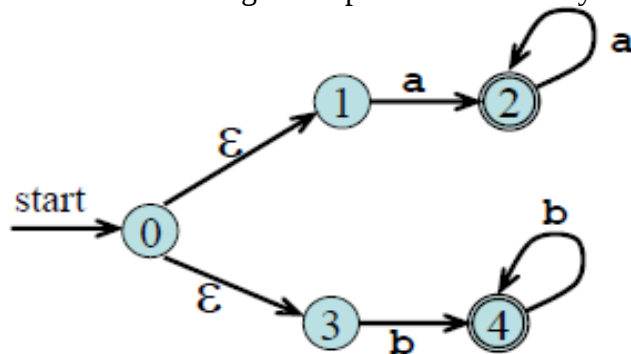
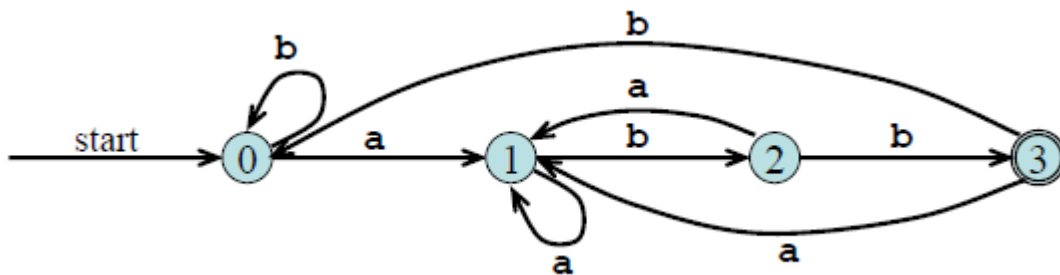


Fig: - ϵ -NFA for regular expression $aa^* + bb^*$

Deterministic Finite Automaton (DFA)

DFA is a special case of NFA. There is only difference between NFA and DFA is in the transition function.

In NFA transition from one state to multiple states take place while in DFA transition from one state to only one possible next state take place.

Fig:-DFA for regular expression $(a+b)^*abb$

Conversion: Regular Expression to NFA

Thomson's Construction

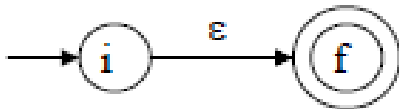
Thomson's Construction is simple and systematic method.

It guarantees that the resulting NFA will have exactly one final state, and one start state.

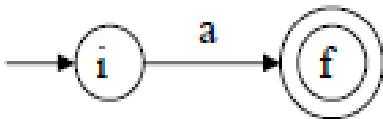
Method:

- First parse the regular expression into sub-expressions
- construct NFA's for each of the basic symbols in regular expression (r)
- Finally combine all NFA's of sub-expressions and we get required NFA of given regular expression.

1. To recognize an empty string ϵ

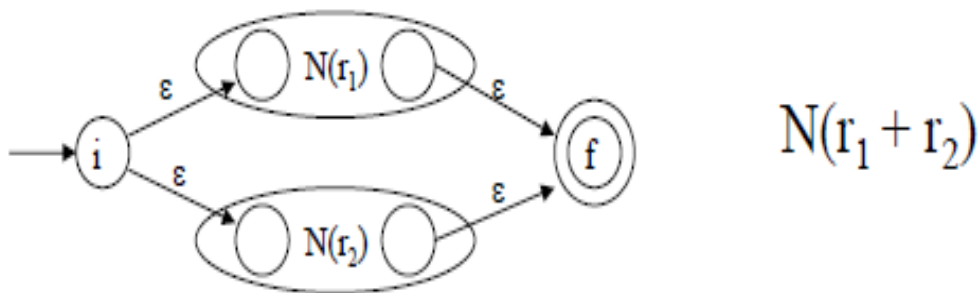


2. To recognize a symbol a in the alphabet Σ

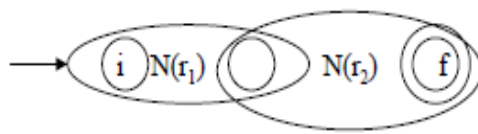


3. If $N(r_1)$ and $N(r_2)$ are NFAs for regular expressions r_1 and r_2

- a. For regular expression $r_1 + r_2$



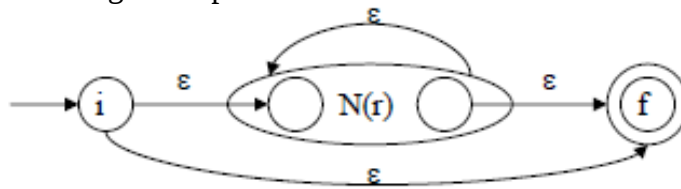
- b. For regular expression $r_1 r_2$



The start state of $N(r_1)$ becomes the start state of $N(r_1 r_2)$ and final state of $N(r_2)$ become final state of $N(r_1 r_2)$

$N(r_1 r_2)$

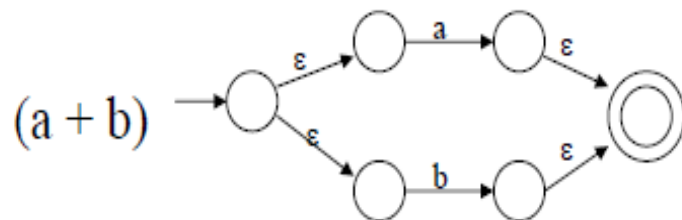
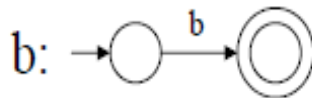
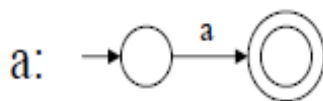
c. For regular expression r^*

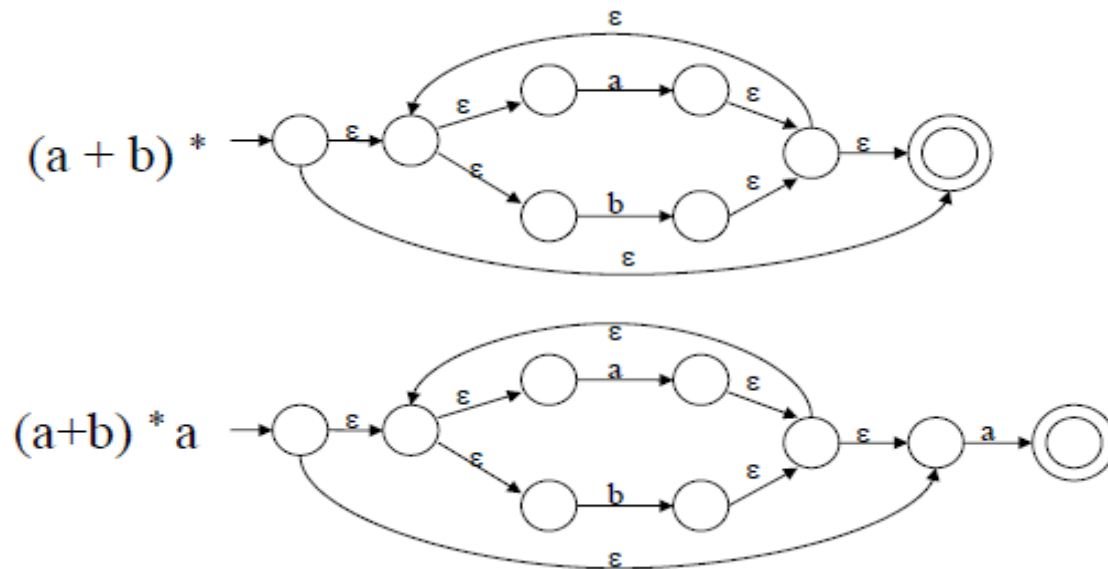


$N(r^*)$

Using rule 1 and 2 we construct NFA's for each basic symbol in the expression, we combine these basic NFA using rule 3 to obtain the NFA for entire expression.

Example: - NFA construction of RE $(a + b)^* a$





Conversion from NFA to DFA

Subset Construction Algorithm

put ϵ -closure(s_0) as an unmarked state in to $Dstates$

while there is an unmarked state T in $Dstates$ **do**

mark T

for each input symbol $a \in \Sigma$ **do**

$U = \epsilon$ -closure(move(T, a))

if U is not in $Dstates$ **then**

add U as an unmarked state to $Dstates$

end if

$Dtran[T, a] = U$

end do

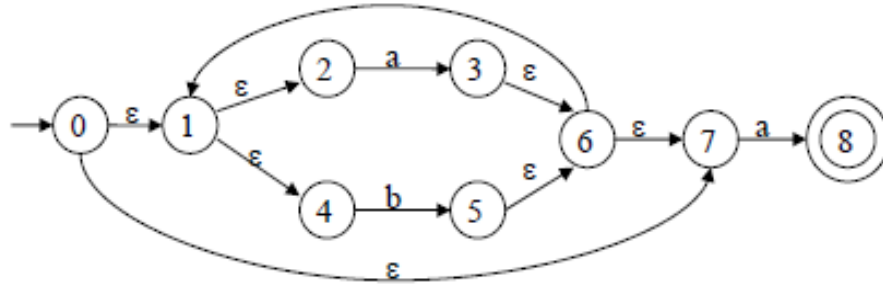
end do

The algorithm produces:

Dstates: $Dstates$ is the set of states of the new DFA consisting of sets of states of the NFA

Dtran: $Dtran$ is the transition table of the new DFA

Subset Construction Example (NFA to DFA)



$$S_0 = \varepsilon\text{-closure}(\{0\}) = \{0,1,2,4,7\}$$

S_0 into $Dstates$ as an unmarked state

\Downarrow mark S_0

$$\varepsilon\text{-closure}(\text{move}(S_0, a)) = \varepsilon\text{-closure}(\{3,8\}) = \{1,2,3,4,6,7,8\} = S_1 \quad S_1 \text{ into } Dstates$$

$$\varepsilon\text{-closure}(\text{move}(S_0, b)) = \varepsilon\text{-closure}(\{5\}) = \{1,2,4,5,6,7\} = S_2 \quad S_2 \text{ into } Dstates$$

$$Dtran[S_0, a] \leftarrow S_1 \quad Dtran[S_0, b] \leftarrow S_2$$

\Downarrow mark S_1

$$\varepsilon\text{-closure}(\text{move}(S_1, a)) = \varepsilon\text{-closure}(\{3,8\}) = \{1,2,3,4,6,7,8\} = S_1$$

$$\varepsilon\text{-closure}(\text{move}(S_1, b)) = \varepsilon\text{-closure}(\{5\}) = \{1,2,4,5,6,7\} = S_2$$

$$Dtran[S_1, a] \leftarrow S_1 \quad Dtran[S_1, b] \leftarrow S_2$$

\Downarrow mark S_2

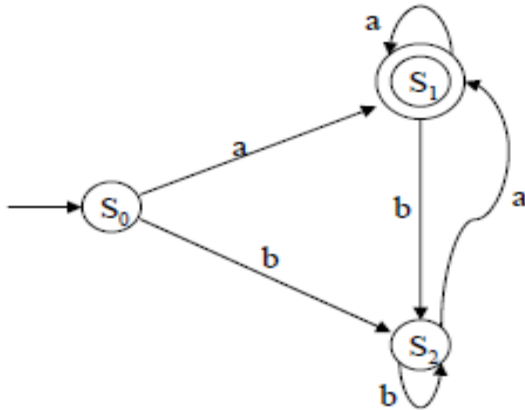
$$\varepsilon\text{-closure}(\text{move}(S_2, a)) = \varepsilon\text{-closure}(\{3,8\}) = \{1,2,3,4,6,7,8\} = S_1$$

$$\varepsilon\text{-closure}(\text{move}(S_2, b)) = \varepsilon\text{-closure}(\{5\}) = \{1,2,4,5,6,7\} = S_2$$

$$Dtran[S_2, a] \leftarrow S_1 \quad Dtran[S_2, b] \leftarrow S_2$$

S_0 is the start state of DFA since 0 is a member of $S_0 = \{0, 1, 2, 4, 7\}$

S_1 is an accepting state of DFA since 8 is a member of $S_1 = \{1, 2, 3, 4, 6, 7, 8\}$



this is final DFA

Exercise:

Convert the following regular expression first into NFA and then into DFA

1. $0^+(1+0)^*00$
2. zero $\rightarrow 0$; one $\rightarrow 1$; bit \rightarrow zero + one; bits \rightarrow bit*

Conversion from RE to DFA Directly

Important States:

A state S of an NFA without ϵ - transition is called the important state if,

$$\text{move}(\{s\}, a) \neq \emptyset$$

In an optimal state machine all states are important states

Augmented Regular Expression:

When we construct an NFA from the regular expression then the final state of resulting NFA is not an important state because it has no transition. Thus to make important state of the accepting state of NFA we introduce an 'augmented' character ($\#$) to a regular expression r .

This resulting regular expression is called the augmented regular expression of original expression r .

Conversion steps:

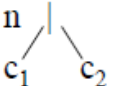
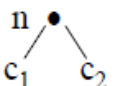

1. Augment the given regular expression by concatenating it with special symbol $\#$ i.e. $r \rightarrow (r) \#$
2. Create the syntax tree for this augmented regular expression

In this syntax tree, all alphabet symbols (plus $\#$ and the empty string) in the augmented regular expression will be on the leaves, and all inner nodes will be the operators in that augmented regular expression.

3. Then each alphabet symbol (plus $\#$) will be numbered (position numbers)
4. Traverse the tree to construct functions *nullable*, *firstpos*, *lastpos*, and *followpos*
5. Finally construct the DFA from the *followpos*

To evaluate *followpos*, we need three functions to defined the nodes (not just for leaves) of the syntax tree.

Rules for calculating nullable, firstpos & lastpos

node <u>n</u>	<u>nullable(n)</u>	<u>firstpos(n)</u>	<u>lastpos(n)</u>
is leaf labeled ϵ	true	Φ	Φ
is leaf labeled with position i	false	$\{i\}$ (position of leaf node)	$\{i\}$
	nullable(c_1) or nullable(c_2)	$\text{firstpos}(c_1) \cup \text{firstpos}(c_2)$	$\text{lastpos}(c_1) \cup \text{lastpos}(c_2)$
	nullable(c_1) and nullable(c_2)	if (nullable(c_1)) then $\text{firstpos}(c_1) \cup \text{firstpos}(c_2)$ else $\text{firstpos}(c_1)$	if (nullable(c_2)) then $\text{lastpos}(c_1) \cup \text{lastpos}(c_2)$ else $\text{lastpos}(c_2)$
	true	$\text{firstpos}(c_1)$	$\text{lastpos}(c_1)$

How to evaluate followpos

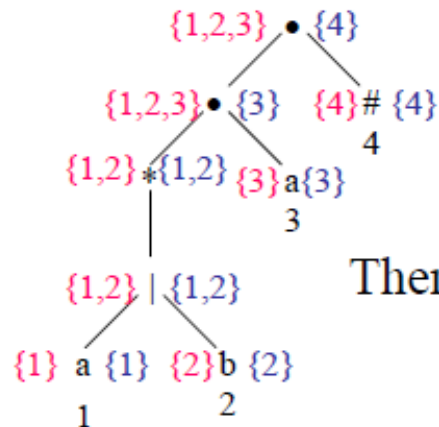
```

for each node  $n$  in the tree do
    if  $n$  is a cat-node with left child  $c_1$  and right child  $c_2$  then
        for each  $i$  in  $\text{lastpos}(c_1)$  do
             $\text{followpos}(i) := \text{followpos}(i) \cup \text{firstpos}(c_2)$ 
        end do
    else if  $n$  is a star-node
        for each  $i$  in  $\text{lastpos}(n)$  do
             $\text{followpos}(i) := \text{followpos}(i) \cup \text{firstpos}(n)$ 
        end do
    end if
end do

```

How to evaluate followpos: Example

For regular expression: $(a \mid b)^* a \#$



Then we can calculate followpos

$\text{followpos}(1) = \{1,2,3\}$
 $\text{followpos}(2) = \{1,2,3\}$
 $\text{followpos}(3) = \{4\}$
 $\text{followpos}(4) = \{\}$

After we calculate follow positions, we are ready to create DFA for the regular expression.

Conversion from RE to DFA Example1

Note: - the start state of DFA is $\text{firstpos}(\text{root})$

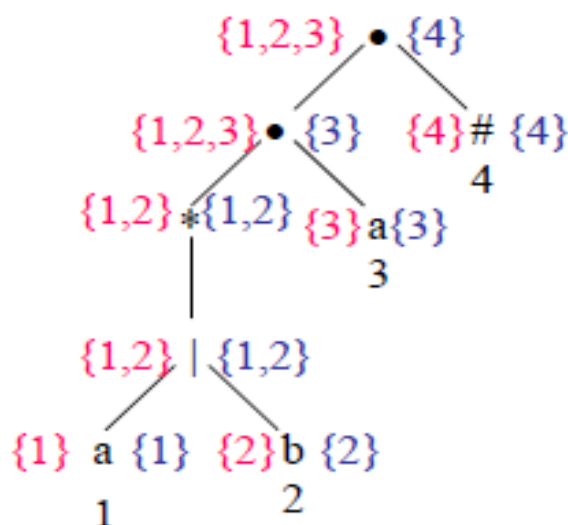
the accepting states of DFA are all states containing the position of #

For the RE --- $(a \mid b)^* a$

Its augmented regular expression is;

$(a \mid b)^* a \#$
 $\begin{matrix} 1 & 2 & 3 & 4 \end{matrix}$

The syntax tree is:



Now we calculate followpos,

$\text{followpos}(1)=\{1,2,3\}$
 $\text{followpos}(2)=\{1,2,3\}$
 $\text{followpos}(3)=\{4\}$
 $\text{followpos}(4)=\{\}$

$S_1 = \text{firstpos}(\text{root}) = \{1,2,3\}$

mark S_1

for a: $\text{followpos}(1) \cup \text{followpos}(3) = \{1,2,3,4\} = S_2$ $\text{move}(S_1, a) = S_2$

for b: $\text{followpos}(2) = \{1,2,3\} = S_1$ $\text{move}(S_1, b) = S_1$

mark S_2

for a: $\text{followpos}(1) \cup \text{followpos}(3) = \{1,2,3,4\} = S_2$ $\text{move}(S_2, a) = S_2$

for b: $\text{followpos}(2) = \{1,2,3\} = S_1$ $\text{move}(S_2, b) = S_1$

Now

start state: S_1

accepting states: $\{S_2\}$

Note:- Accepting states=states containing position of # ie 4.

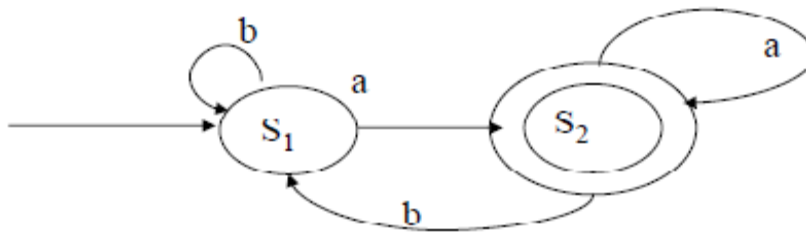


Fig: Resulting DFA of given regular expression

Conversion from RE to DFA

Example2

For RE---- (a | ϵ) b c* #

$\text{followpos}(1)=\{2\}$
 $\text{followpos}(2)=\{3,4\}$
 $\text{followpos}(3)=\{3,4\}$
 $\text{followpos}(4)=\{\}$

$S_1 = \text{firstpos}(\text{root}) = \{1,2\}$

mark S_1

for a: $\text{followpos}(1)=\{2\}=S_2$ $\text{move}(S_1, a)=S_2$

for b: $\text{followpos}(2)=\{3,4\}=S3$ $\text{move}(S1,b)=S3$
 mark S2

for b: $\text{followpos}(2)=\{3,4\}=S3$ $\text{move}(S2,b)=S3$
 mark S3

for c: $\text{followpos}(3)=\{3,4\}=S3$ $\text{move}(S3,c)=S3$

start state: S1

accepting states: {S3}

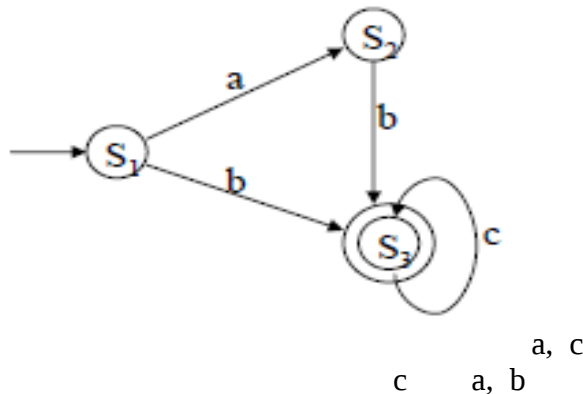


fig:- DFA for above RE

D

State minimization in DFA:

Partition the set of states into two groups:

- G1: set of accepting states
- G2: set of non-accepting states

For each new group G:

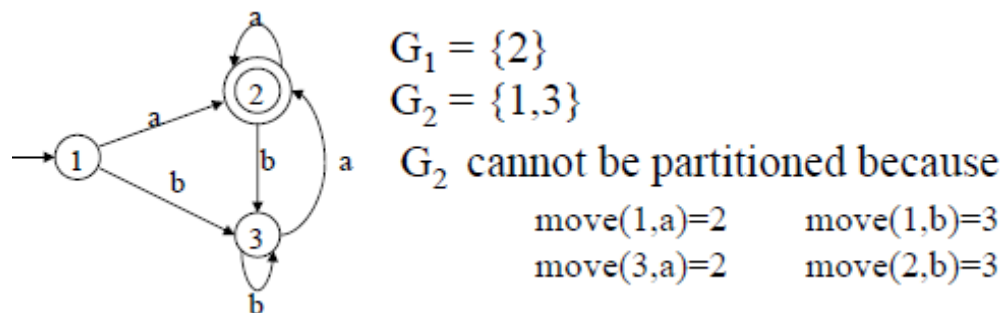
– partition G into subgroups such that states s1 and s2 are in the same group if for all input symbols a, states s1 and s2 have transitions to states in the same group.

Start state of the minimized DFA is the group containing the start state of the original DFA.

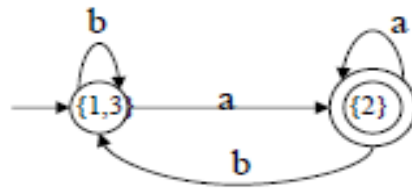
Accepting states of the minimized DFA are the groups containing the accepting states of the original DFA.

State Minimization in DFA

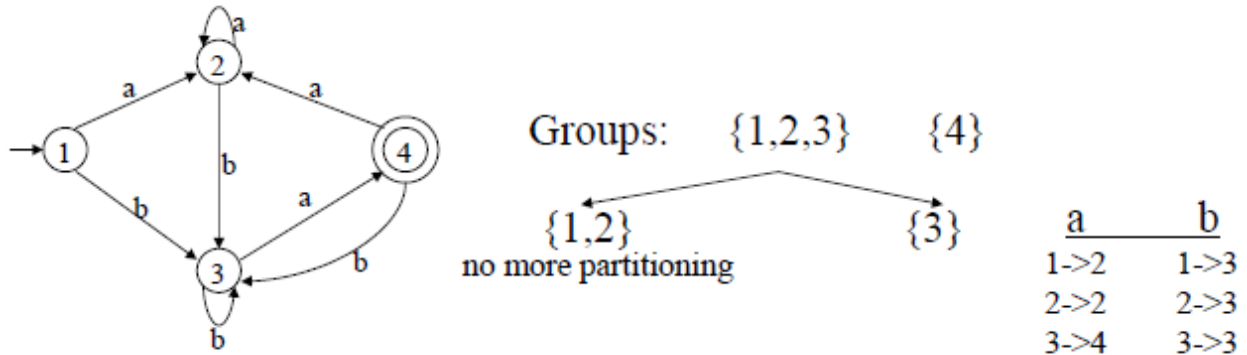
Example1:



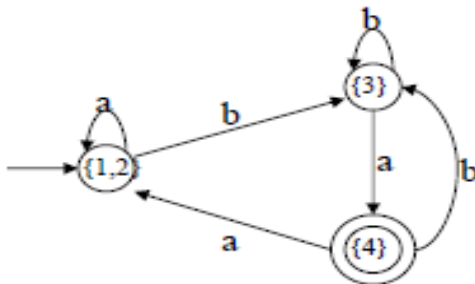
So, the minimized DFA (with minimum states)



Example 2:

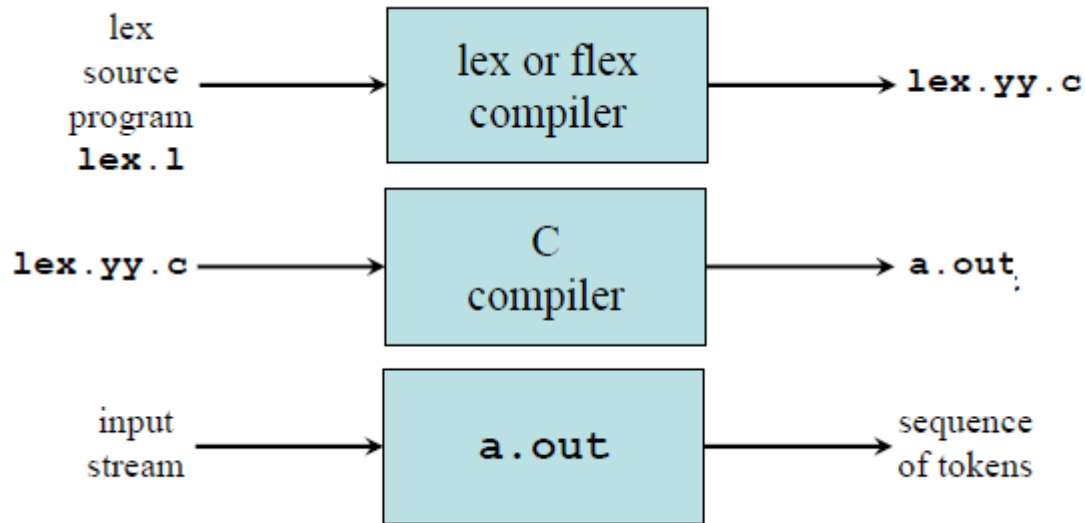


So minimized DFA is:



Flex: Language for Lexical Analyzer

Systematically translate regular definitions into C source code for efficient scanning.
Generated code is easy to integrate in C applications



Flex: An introduction

Flex is a tool for generating scanners. A scanner is a program which recognizes lexical patterns in text. The flex program reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. flex generates as output a C source file, 'lex.yy.c' by default, which defines a routine yylex(). This file can be compiled and linked with the flex runtime library to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.

Flex specification:

A *flex specification* consists of three parts:

Regular definitions, C declarations in %{ %}

%%

Translation rules

%%

User-defined auxiliary procedures

The *translation rules* are of the form:

```

p1      {action1}
p2      {action2}
.....
pn     { actionn }
```

In all parts of the specification comments of the form */* comment text */* are permitted.

Regular definitions:

It consist two things:

- Any C code that is external to any function should be in *%{ %}*
- Declaration of simple name definitions i.e specifying regular expression e.g
 DIGIT [0-9]

ID [a-z][a-z0-9]*

The subsequent reference is as {DIGIT}, {DIGIT}+ or {DIGIT}*

Translation rules:

Contains a set of regular expressions and actions (C code) that are executed when the scanner matches the associated regular expression e.g

```
{ID}                printf("%s", getlogin());
```

Any code that follows a regular expression will be inserted at the appropriate place in the recognition procedure *yylex()*

Finally the user code section is simply copied to *lex.yy.c*

Practice

- Get familiar with FLEX
 1. Try sample*.lex
 2. Command Sequence:


```
flex sample*.lex
gcc lex.yy.c -lfl
./a.out
```

Flex operators and Meaning

x	match the character x
\.	match the character .
"string"	match contents of string of characters
.	match any character except newline
^	match beginning of a line
\$	match the end of a line
[xyz]	match one character x , y , or z (use \ to escape -)
[^xyz]	match any character except x , y , and z
[a-z]	match one of a to z
r*	closure (match zero or more occurrences)
r+	positive closure (match one or more occurrences)
r?	optional (match zero or one occurrence)
r1r2	match r1 then r2 (concatenation)
r1 r2	match r1 or r2 (union)
(r)	grouping
r1r2	match r1 when followed by r2
{d}	match the regular expression defined by d
'r{2,5}'	anywhere from two to five r 's
'r{2,}'	two or more r 's
'r{4}'	exactly 4 r 's

Flex Global Function, Variables & Directives

yylex() is the scanner function that can be invoked by the parser

yytext extern char *yytext; is a global char pointer holding the currently matched lexeme.

yylen extern int yylen; is a global int that contains the length of the currently matched lexeme.

ECHO copies yytext to the scanner's output

REJECT directs the scanner to proceed on to the "second best" rule which matched the input
yyomore() tells the scanner that the next time it matches a rule, the corresponding token should be appended onto the current value of *yytext* rather than replacing it.

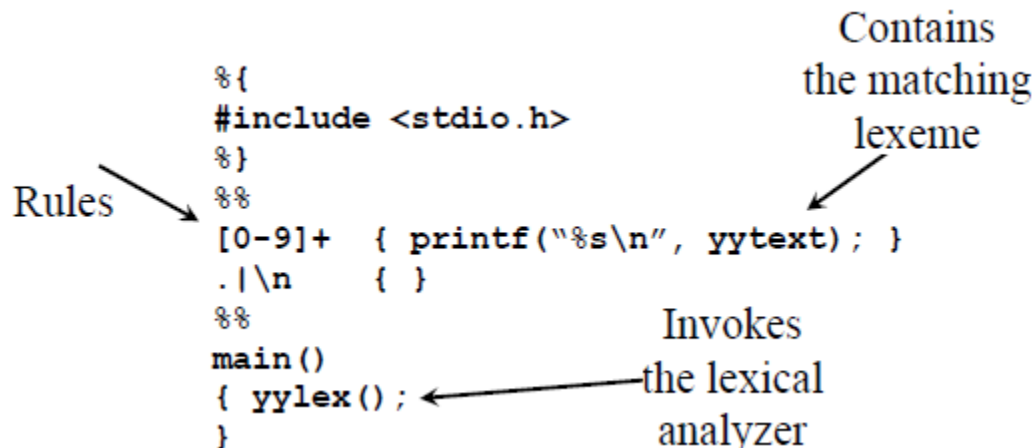
yyless(*n*) returns all but the first *n* characters of the current token back to the input stream, where they will be rescanned when the scanner looks for the next match

unput(*c*) puts the character *c* back onto the input stream. It will be the next character scanned

input() reads the next character from the input stream

YY_FLUSH_BUFFER flushes the scanner's internal buffer so that the next time the scanner attempts to match a token; it will first refill the buffer.

Flex Example1



```

%{
#include <stdio.h>
%}
%%
[0-9]+ { printf("%s\\n", yytext); }
.|\\n { }
%%
main()
{ yylex(); }

```

Example2

```

/*
* Description: Count the number of characters and the number of lines
* from standard input
* Usage:
  (1) $ flex sample2.lex
  * (2) $ gcc lex.yy.c -lfl
  * (3) $ ./a.out
  * stdin> whatever you like
  * stdin> Ctrl-D
* Questions: Is it ok if we do not indent the first line?
* What will happen if we remove the second rule?
*/

```

```

int num_lines = 0, num_chars = 0;
%%
\\n    ++num_lines; ++num_chars;

```

```

        .      ++num_chars;
%%
main()
{
    yylex();
    printf("# of lines = %d, # of chars = %d\n", num_lines, num_chars);
}

```

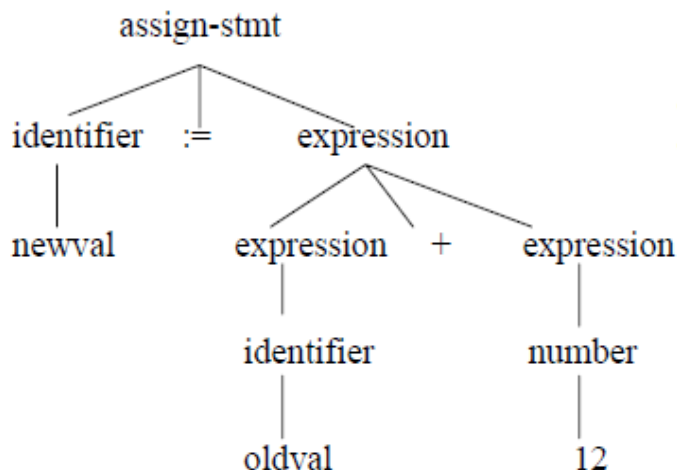
unit: 2

Syntax Analysis

A **Syntax Analyzer** creates the syntactic structure (generally a parse tree) of the given source program.

Syntax analyzer is also called the **parser**. Its job is to analyze the source program based on the definition of its syntax. It works in lock-step with the lexical analyzer and is responsible for creating a parse-tree of the source code.

Ex: newval: = oldval + 12



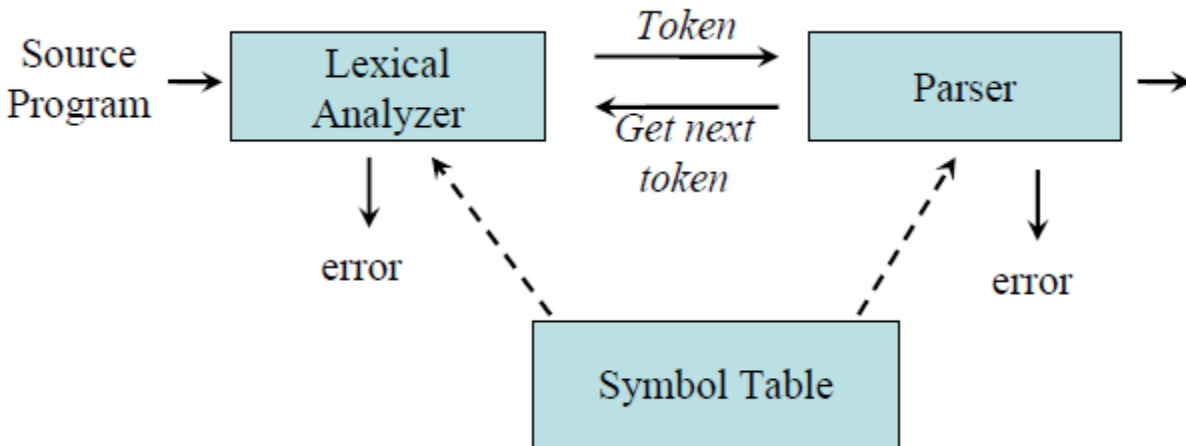
- In a parse tree, all terminals are at leaves.
- All inner nodes are non-terminals in a context free grammar.

The syntax of a language is specified by a **context free grammar** (CFG).

The rules in a CFG are mostly recursive.

A syntax analyzer checks whether a given program satisfies the rules implied by a CFG or not.

– If it satisfies, the syntax analyzer creates a parse tree for the given program.



Context-Free Grammars

Context-free grammar is a 4-tuple $G = (N, T, P, S)$ where

- T is a finite set of tokens (*terminal* symbols)
- N is a finite set of *non-terminals*
- P is a finite set of *productions* of the form

$$\alpha \rightarrow \beta$$

Where,

$$\alpha \in (N \cup T)^* N (N \cup T)^* \text{ and } \beta \in (N \cup T)^*$$

- $S \in N$ is a designated *start symbol*

Programming languages usually have recursive structures that can be defined by a context-free grammar (CFG).

CFG: Notational Conventions

Terminals are denoted by lower-case letters and symbols (single atoms) and **bold** strings (tokens)

$$a, b, c, \dots \in T$$

specific terminals:

$$\mathbf{0}, \mathbf{1}, \mathbf{id}, +$$

Non-terminals are denoted by *lower-case italicized* letters or upper-case letters symbols

$$A, B, C, \dots \in N$$

specific non-terminals:

$$expr, term, stmt$$

Production rules are of the form

$$A \rightarrow \alpha,$$

, that is read as “A can produce α ”

Strings comprising of both terminals and non-terminals are denoted by greek letters

$$\alpha, \beta, \text{ etc}$$

Left-most derivation:

If we always choose the left-most non-terminal in each derivation step, this derivation is called as **left-most derivation**.

Eg:-

$$E \xRightarrow{\text{lm}} -E \xRightarrow{\text{lm}} -(E) \xRightarrow{\text{lm}} -(E+E) \xRightarrow{\text{lm}} -(id+E) \xRightarrow{\text{lm}} -(id+id)$$

Right-most derivation:

If we always choose the right-most non-terminal in each derivation step, this derivation is called as **right-most derivation**.

Eg:

$$E \xRightarrow{\text{rm}} -E \xRightarrow{\text{rm}} -(E) \xRightarrow{\text{rm}} -(E+E) \xRightarrow{\text{rm}} -(E+id) \xRightarrow{\text{rm}} -(id+id)$$

Parse Tree

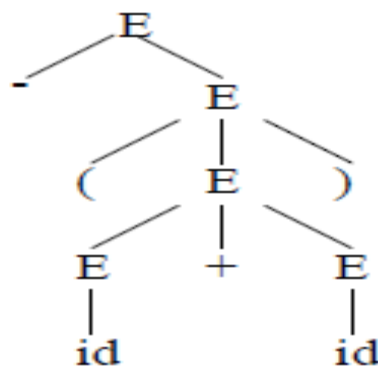
A parse tree is a graphic representation of a CFG with the following properties:

- The root node is labeled by start symbol.
- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.

Eg: let us consider a CFG:

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

Then the parse tree for $-(id+id)$ is:



Ambiguity of a grammar:

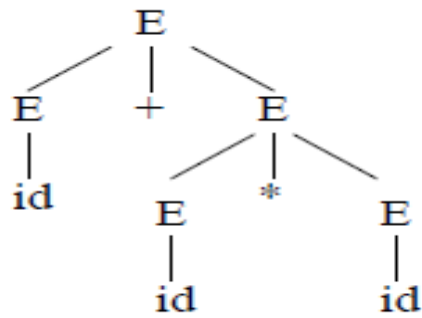
A grammar G is said to be ambiguous if there is a string $w \in L(G)$ for which we can construct more than one parse tree rooted at start symbol of the production.

Eg: let us consider a CFG:

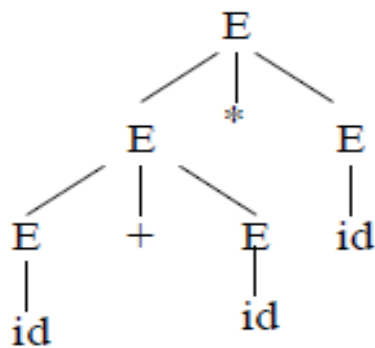
$$E \rightarrow E + E \mid E * E \mid (E) \mid -E \mid id$$

The parse trees for the string $id+id*id$ are as follows:

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



Another possible parse tree is:



Thus the above grammar is ambiguous.

Parsing

Given a stream of input tokens, *parsing* involves the process of reducing them to a non-terminal. Parsing can be either *top-down* or *bottom-up*.

Top-down parsing involves generating the string starting from the first non-terminal and repeatedly applying production rules.

Bottom-up parsing involves repeatedly rewriting the input string until it ends up in the first non-terminal of the grammar.

Top-Down Parsing

The parse tree is created top to bottom.

Top-down parser

- Recursive-Descent Parsing
- Predictive Parsing

Recursive-Descent Parsing

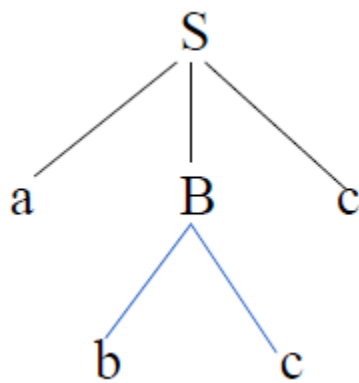
- Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
- It is a general parsing technique, but not widely used.
- Not efficient

It tries to find the left-most derivation

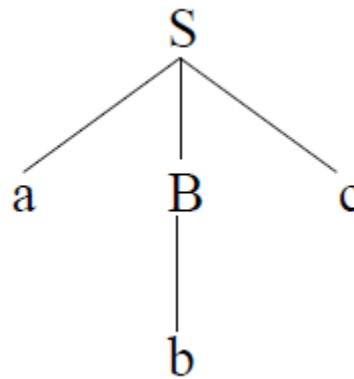
Eg: $S \rightarrow aBc$

$$B \rightarrow bc \vee b$$

Input: abc



Fails



backtrack

Method: let input $w = abc$, initially create the tree of single node S . The left most node a match the first symbol of w , so advance the pointer to b and consider the next leaf B . Then expand B using first choice bc . There is match for b and c , and advanced to the leaf symbol c of S , but there is no match in input, report failure and go back to B to find another alternative b that produce match.

Left Recursion

A grammar is **left recursive** if it has a non-terminal A such that there is a derivation.

$$A \rightarrow A\alpha \quad \text{For some string } \alpha$$

Top-down parsing techniques **cannot** handle left-recursive grammars.

So, we have to convert our left-recursive grammar into an equivalent grammar which is not left-recursive.

Eg:

Immediate Left-Recursion:

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

Eliminate immediate left recursion

$$A \rightarrow \beta A'$$

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

In general,

$$A \rightarrow A \alpha_1 \mid \dots \mid A \alpha_m \mid \beta_1 \mid \dots \mid \beta_n \quad \text{where } \beta_1 \dots \beta_n \text{ do not start with } A$$



eliminate immediate left recursion

$$A \rightarrow \beta_1 A' \mid \dots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \dots \mid \alpha_m A' \mid \varepsilon \quad \text{an equivalent grammar}$$

Immediate Left-Recursion - Example

$$E \rightarrow E+T \mid T$$

$$T \rightarrow T*F \mid F$$

$$F \rightarrow \text{id} \mid (E)$$



eliminate immediate left recursion

$$E \rightarrow T E'$$

$$E' \rightarrow +T E' \mid \varepsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow *F T' \mid \varepsilon$$

$$F \rightarrow \text{id} \mid (E)$$

Non-Immediate Left-Recursion

By just eliminating the immediate left-recursion, we may not get a grammar which is not left-recursive.

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Sc \mid d \quad \text{This grammar is not immediately left-recursive, but it is still left-recursive.}$$

$$\underline{S} \Rightarrow Aa \Rightarrow \underline{S}ca \quad \text{or}$$

$$\underline{A} \Rightarrow Sc \Rightarrow \underline{A}ac \quad \text{causes to a left-recursion}$$

So, we have to eliminate all left-recursions from our grammar

So, the resulting equivalent grammar which is not left-recursive is:

$$S \rightarrow Aa \mid b$$

$$A \rightarrow bdA' \mid fA'$$

$$A' \rightarrow cA' \mid adA' \mid \varepsilon$$

Left-Factoring

When a non-terminal has two or more productions whose right-hand sides start with the same grammar symbols, then such a grammar is not LL(1) and cannot be used for predictive parsing. This grammar is called left factoring grammar.

Eg:

Replace productions

$$A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \dots \mid \alpha \beta_n \mid \gamma$$

with

$$A \rightarrow \alpha A' \mid \gamma$$

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

Hint: taking α common from the each production.

Predictive Parsing

A *predictive parser* tries to predict which production produces the least chances of a backtracking and infinite looping.

When re-writing a non-terminal in a derivation step, a predictive parser can uniquely choose a production rule by just looking the current symbol in the input string.

Two variants:

- Recursive (recursive-descent parsing)
- Non-recursive (table-driven parsing)

Non-Recursive Predictive Parsing

Non-Recursive predictive parsing is a table-driven parser.

Given an LL(1) grammar $G = (N, T, P, S)$ construct a table $M[A, a]$ for $A \in N$, $a \in T$ and use a driver program with a stack.

A table driven predictive parser has an input buffer, a stack, a parsing table and an output stream.

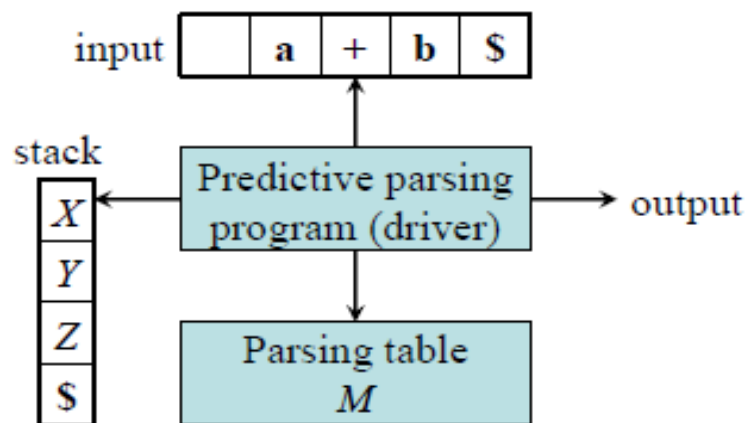


Fig model of a non-recursive predictive parser

Input buffer:

It contains the string to be parsed followed by a special symbol \$.

Stack:

A stack contains a sequence of grammar symbols with \$ on the bottom. Initially it contains the symbol \$.

Parsing table:

It is a two dimensional array $M[A, a]$ where 'A' is non-terminal and 'a' is a terminal symbol.

Output stream:

A production rule representing a step of the derivation sequence of the string in the input buffer.

Algorithm

Input : a string w .

Output: if w is in $L(G)$, a leftmost derivation of w ; otherwise error

1. Set ip to the first symbol of input stream
2. Set the stack to $\$S$ where S is the start symbol of the grammar
3. repeat
 - Let X be the top stack symbol and a be the symbol pointed by ip
 - If X is a terminal or $\$$ then
 - if $X = a$ then pop X from the stack and advance ip
 - else error()
 - else /* X is a non-terminal */
 - if $M[X, a] = X \rightarrow Y_1, Y_2, \dots, Y_k$ then
 - pop X from stack
 - push Y_k, Y_{k-1}, \dots, Y_1 onto stack (with Y_1 on top)
 - output the production $X \rightarrow Y_1, Y_2, \dots, Y_k$
 - else error()
4. until $X = \$$ /* stack is empty */

Example: Given a grammar,

$S \rightarrow aBa$

$B \rightarrow bB \mid \epsilon$

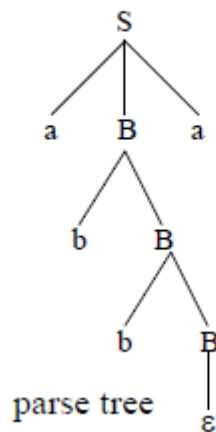
Input: abba

<u>stack</u>	<u>input</u>	<u>output</u>
$\$S$	abba\$	$S \rightarrow aBa$
$\$aBa$	abba\$	
$\$aB$	bba\$	$B \rightarrow bB$
$\$aBb$	bba\$	
$\$aB$	ba\$	$B \rightarrow bB$
$\$aBb$	ba\$	
$\$aB$	a\$	$B \rightarrow \epsilon$
$\$a$	a\$	
$\$$	$\$$	accept, successful completion

	a	b	S
S	$S \rightarrow aBa$		
B	$B \rightarrow \varepsilon$	$B \rightarrow bB$	

LL(1) Parsing Table

Outputs: $S \rightarrow aBa$ $B \rightarrow bB$ $B \rightarrow bB$ $B \rightarrow \varepsilon$
 Derivation(left-most): $S \Rightarrow aBa \Rightarrow abBa \Rightarrow abbBa \Rightarrow abba$



Constructing LL(1) Parsing Tables

- Eliminate left recursion from grammar
- Eliminate left factor of the grammar

a grammar $\xrightarrow{\text{eliminate left recursion}}$ $\xrightarrow{\text{eliminate left factor}}$ a grammar suitable for predictive parsing (a LL(1) grammar)

To compute LL (1) parsing table, at first we need to compute FIRST and FOLLOW functions.

Compute FIRST

FIRST(α) is a set of the terminal symbols which occur as first symbols in strings derived from α where α is any string of grammar symbols.

If α derives to ϵ , then ϵ is also in FIRST (α).

Compute FIRST algorithm:

1. If X is a terminal symbol then $\text{FIRST}(X) = \{X\}$
2. If X is a non-terminal symbol and $X \rightarrow \epsilon$ is a production rule then $\text{FIRST}(X) = \text{FIRST}(X) \cup \epsilon$.
3. If X is a non-terminal symbol and $X \rightarrow Y_1 Y_2 \dots Y_n$ is a production rule then
 - a. if a terminal a in $\text{FIRST}(Y_1)$ then $\text{FIRST}(X) = \text{FIRST}(X) \cup \text{FIRST}(Y_1)$
 - b. if a terminal a in $\text{FIRST}(Y_j)$ and ϵ is in all $\text{FIRST}(Y_j)$ for $j=1, \dots, i-1$ then $\text{FIRST}(X) = \text{FIRST}(X) \cup a$.
 - c. if ϵ is in all $\text{FIRST}(Y_j)$ for $j=1, \dots, n$ then $\text{FIRST}(X) = \text{FIRST}(X) \cup \epsilon$.
- If X is ϵ then $\text{FIRST}(X) = \{\epsilon\}$
- If X is $Y_1 Y_2 \dots Y_n$
 - a. if a terminal a in $\text{FIRST}(Y_j)$ and ϵ is in all $\text{FIRST}(Y_j)$ for $j=1, \dots, i-1$ then $\text{FIRST}(X) = \text{FIRST}(X) \cup a$
 - b. if ϵ is in all $\text{FIRST}(Y_j)$ for $j=1, \dots, n$ then $\text{FIRST}(X) = \text{FIRST}(X) \cup \epsilon$.

Compute FIRST: Example

$E \rightarrow TE'$
 $E' \rightarrow +TE' \mid \epsilon$
 $T \rightarrow FT'$
 $T' \rightarrow *FT' \mid \epsilon$
 $F \rightarrow (E) \mid id$

$\text{FIRST}(F) = \{ (, id \}$	$\text{FIRST}(TE') = \{ (, id \}$
$\text{FIRST}(T') = \{ *, \epsilon \}$	$\text{FIRST}(+TE') = \{ + \}$
$\text{FIRST}(T) = \{ (, id \}$	$\text{FIRST}(\epsilon) = \{ \epsilon \}$
$\text{FIRST}(E') = \{ +, \epsilon \}$	$\text{FIRST}(FT') = \{ (, id \}$
$\text{FIRST}(E) = \{ (, id \}$	$\text{FIRST}(*FT') = \{ * \}$
	$\text{FIRST}(\epsilon) = \{ \epsilon \}$
	$\text{FIRST}((E)) = \{ (\}$
	$\text{FIRST}(id) = \{ id \}$

Compute FOLLOW:

FOLLOW (A) is the set of the terminals which occur immediately after (follow) the *non-terminal A* in the strings derived from the starting symbol.

- a terminal a is in $\text{FOLLOW}(A)$ if $S \xRightarrow{*} \alpha A a \beta$
- $\$$ is in $\text{FOLLOW}(A)$ if $S \xRightarrow{*} \alpha A$

Compute FOLLOW algorithm:

Apply the following rules until nothing can be added to any FOLLOW set:

1. If S is the start symbol then $\$$ is in $\text{FOLLOW}(S)$
2. if $A \rightarrow \alpha B \beta$ is a production rule then everything in $\text{FIRST}(\beta)$ is placed in $\text{FOLLOW}(B)$ except ϵ
3. If $(A \rightarrow \alpha B \text{ is a production rule})$ or $(A \rightarrow \alpha B \beta \text{ is a production rule and } \epsilon \text{ is in } \text{FIRST}(\beta))$ then everything in $\text{FOLLOW}(A)$ is in $\text{FOLLOW}(B)$.

Compute FOLLOW: Example

$E \rightarrow TE'$
 $E' \rightarrow +TE' \mid \epsilon$
 $T \rightarrow FT'$
 $T' \rightarrow *FT' \mid \epsilon$
 $F \rightarrow (E) \mid \text{id}$

$\text{FOLLOW}(E) = \{ \$,) \}$
 $\text{FOLLOW}(E') = \{ \$,) \}$
 $\text{FOLLOW}(T) = \{ +,), \$ \}$
 $\text{FOLLOW}(T') = \{ +,), \$ \}$
 $\text{FOLLOW}(F) = \{ +, *,), \$ \}$

Constructing LL(1) Parsing Tables

Algorithm

Input: LL(1) Grammar G

Output: Parsing Table M

for each production rule $A \rightarrow \alpha$ of a grammar G

 for each terminal a in $\text{FIRST}(\alpha)$

 add $A \rightarrow \alpha$ to $M[A, a]$

 If ϵ in $\text{FIRST}(\alpha)$ then

 for each terminal a in $\text{FOLLOW}(A)$

 add $A \rightarrow \alpha$ to $M[A, a]$

 If ϵ in $\text{FIRST}(\alpha)$ and $\$$ in $\text{FOLLOW}(A)$ then

 add $A \rightarrow \alpha$ to $M[A, \$]$

Constructing LL(1) Parsing Tables: Example1

$E \rightarrow TE'$
 $E' \rightarrow +TE' \mid \varepsilon$
 $T \rightarrow FT'$
 $T' \rightarrow *FT' \mid \varepsilon$
 $F \rightarrow (E) \mid id$

$FIRST(F) = \{ (, id \}$
 $FIRST(T') = \{ *, \varepsilon \}$
 $FIRST(T) = \{ (, id \}$
 $FIRST(E') = \{ +, \varepsilon \}$
 $FIRST(E) = \{ (, id \}$
 $FIRST(TE') = \{ (, id \}$
 $FIRST(+TE') = \{ + \}$
 $FIRST(\varepsilon) = \{ \varepsilon \}$
 $FIRST(FT') = \{ (, id \}$
 $FIRST(*FT') = \{ * \}$
 $FIRST(\varepsilon) = \{ \varepsilon \}$
 $FIRST((E)) = \{ (\}$
 $FIRST(id) = \{ id \}$

$FOLLOW(E) = \{ \$, FIRST(') \} = \{ \$,) \}$
 $FOLLOW(E') = \{ FOLLOW(E) \} = \{ \$,) \}$
 $FOLLOW(T) = \{ FIRST(E'), FOLLOW(E') \} = \{ +,), \$ \}$
 $FOLLOW(T') = \{ FOLLOW(T) \} = \{ +,), \$ \}$
 $FOLLOW(F) = \{ FOLLOW(T'), FIRST(T') \text{ except } \varepsilon \} = \{ +,), \$, * \}$

Non-terminals	Terminal Symbols					
	+	*	()	id	\$
E			$E \rightarrow TE'$		$E \rightarrow TE'$	
E'	$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$		$E' \rightarrow \varepsilon$
T			$T \rightarrow FT'$		$T \rightarrow FT'$	
T'	$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$		$T' \rightarrow \varepsilon$
F			$F \rightarrow (E)$		$F \rightarrow id$	

Constructing LL(1) Parsing Tables: Example2

\rightarrow
 $S \rightarrow iEtSS' \mid a$
 \rightarrow
 $S' \rightarrow eS \mid \varepsilon$
 \rightarrow
 $E \rightarrow b$

Construct LL(1) parsing table for this grammar.

Soln:

$\text{FIRST}(S) = \{i, a\}$
 $\text{FIRST}(S') = \{e, \epsilon\}$
 $\text{FIRST}(E) = \{b\}$
 $\text{FIRST}(iEtSS') = \{i\}$
 $\text{FIRST}(a) = \{a\}$
 $\text{FIRST}(eS) = \{e\}$
 $\text{FIRST}(\epsilon) = \{\epsilon\}$

$\text{FOLLOW}(S) = \{\text{FIRST}(S')\} = \{e, \$\}$
 $\text{FOLLOW}(S') = \{\text{FOLLOW}(S)\} = \{e, \$\}$
 $\text{FOLLOW}(E) = \{\text{FIRST}(tSS')\} = \{t\}$
 $\text{FIRST}(b) = \{b\}$

Construct table itself.

LL(1) Grammars

A grammar whose parsing table has no multiply-defined entries is said to be LL(1) grammar.

What happen when a parsing table contains multiply defined entries?

– The problem is ambiguity

A left recursive, not left factored and ambiguous grammar cannot be a LL(1) grammar (i.e. left recursive, not left factored and ambiguous grammar may have multiply –defined entries in parsing table)

Properties of LL(1) Grammars

one input symbol used as a look-head symbol do determine parser action

$\text{LL}(1)$ left most derivation
 input scanned from left to right

A grammar G is LL(1) if and only if the following conditions hold for two distinctive production rules $A \rightarrow \alpha$ and $A \rightarrow \beta$

1. Both α and β cannot derive strings starting with same terminals.
2. At most one of α and β can derive to ϵ .
3. If β can derive to ϵ , then α cannot derive to any string starting with a terminal in $\text{FOLLOW}(A)$.

Exercise:

Q. For the grammar,

$S \rightarrow [C]S | \epsilon$
 $C \rightarrow \{A\}C | \epsilon$
 $A \rightarrow A() | \epsilon$

Construct the predictive top down parsing table (LL (1) parsing table)

Bottom-Up Parsing

Bottom-up parsing attempts to construct a parse tree for an input string starting from leaves (the bottom) and working up towards the root (the top).

Reduction:

The process of replacing a substring by a non-terminal in bottom-up parsing is called reduction.

It is a reverse process of production.

Eg: $S \xrightarrow{\quad} aA$

Here, if replacing aA by S then such a grammar is called reduction.

Shift-Reduce Parsing

The process of reducing the given input string into the starting symbol is called shift-reduce parsing.

A string the starting symbol
Reduced to

Example:

$S \rightarrow aABb$

$A \rightarrow aA \mid a$

$B \rightarrow bB \mid b$

$S \xRightarrow{m} aABb \xRightarrow{m} aAbb \xRightarrow{m} aaAbb \xRightarrow{m} aaabb$

input string: $aaabb$

$aaAbb$

$aAbb \quad \Downarrow \text{reduction}$

$aABb$

S

Handle

A substring that can be replaced by a non-terminal when it matches its right sentential form is called a **handle**.

If the grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.

Example: A Shift-Reduce Parser with Handle

$E \rightarrow E+T \mid T$
 $T \rightarrow T * F \mid F$
 $F \rightarrow (E) \mid id$

Right-Most Derivation of $id+id*id$
 $E \Rightarrow E+T \Rightarrow E+T * F \Rightarrow E+T * id \Rightarrow E+F * id$
 $\Rightarrow E+id * id \Rightarrow T+id * id \Rightarrow F+id * id \Rightarrow id+id * id$

<u>Right-Most Sentential Form</u>	<u>Reducing Production</u>	<u>Handle</u>
<u>id</u> +id*id	$F \rightarrow id$	id
<u>F</u> +id*id	$T \rightarrow F$	F
<u>T</u> +id*id	$E \rightarrow T$	T
E+ <u>id</u> *id	$F \rightarrow id$	id
E+ <u>F</u> *id	$T \rightarrow F$	F
E+T* <u>id</u>	$F \rightarrow id$	id
E+T* <u>F</u>	$T \rightarrow T * F$	T * F
<u>E+T</u>	$E \rightarrow E+T$	E+T
E		

Stack Implementation of Shift-Reduce Parser

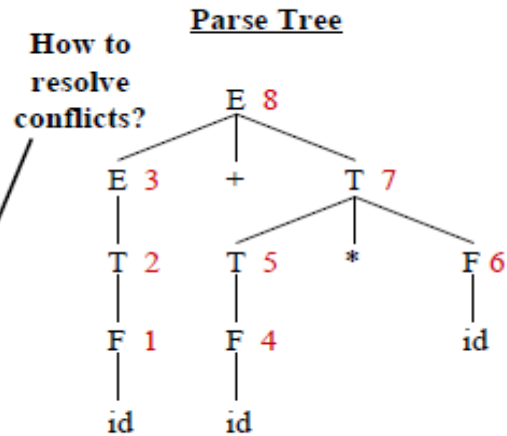
The stack holds the grammar symbols and input buffer holds the string w to be parsed.

- Initially stack contains only the sentinel $\$$, and input buffer contains the input string $w\$$.
- While stack not equal to $\$S$ or not **error** and input not $\$$ do
 - While there is no handle at the top of stack, do **shift** input buffer and push the symbol onto stack
 - If there is a handle on top of stack, then pop the handle and **reduce** the handle with its non-terminal and push it onto stack
- Done

Parser Actions:

- Shift:** The next input symbol is shifted onto the top of the stack.
- Reduce:** Replace the handle on the top of the stack by the non-terminal.
- Accept:** Successful completion of parsing.
- Error:** Parser discovers a syntax error, and calls an error recovery routine

<u>Stack</u>	<u>Input</u>	<u>Action</u>
\$	id+id*id\$	shift
\$id	+id*id\$	reduce by $F \rightarrow id$
\$F	+id*id\$	reduce by $T \rightarrow F$
\$T	+id*id\$	reduce by $E \rightarrow T$
\$E	+id*id\$	shift
\$E+	id*id\$	shift
\$E+id	*id\$	reduce by $F \rightarrow id$
\$E+F	*id\$	reduce by $T \rightarrow F$
\$E+T	*id\$	shift (or reduce?)
\$E+T*	id\$	shift
\$E+T*id	\$	reduce by $F \rightarrow id$
\$E+T*F	\$	reduce by $T \rightarrow T*F$
\$E+T	\$	reduce by $E \rightarrow E+T$
\$E	\$	accept



Conflicts in Shift-Reduce Parsing

Some grammars cannot be parsed using shift-reduce parsing and result in *conflicts*. There are two kinds of shift-reduce conflicts:

shift/reduce conflict:

Here, the parser is not able to decide whether to shift or to reduce.

Example:

\rightarrow
A \rightarrow ab | abcd

the stack contains \$ab, and

the input buffer contains cd\$, the parser cannot decide whether to reduce \$ab to \$A or to shift two more symbols before reducing.

reduce/reduce conflict:

Here, the parser cannot decide which sentential form to use for reduction.

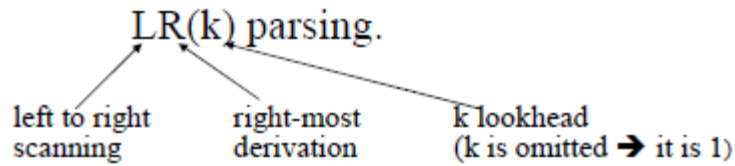
For example

\rightarrow
A \rightarrow bc

\rightarrow
B \rightarrow abc and the stack contains \$abc, the parser cannot decide whether to reduce it to \$aA or to \$B.

LR Parsers

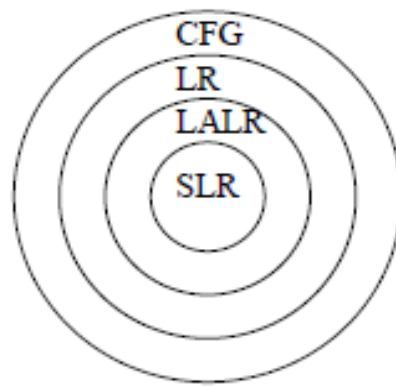
- LR parsing is most general non-backtracking, efficient and most powerful shift-reduce parsing.
- LL(1)-Grammars \subset LR(1)-Grammars
- An LR-parser can detect a syntactic error so fast.



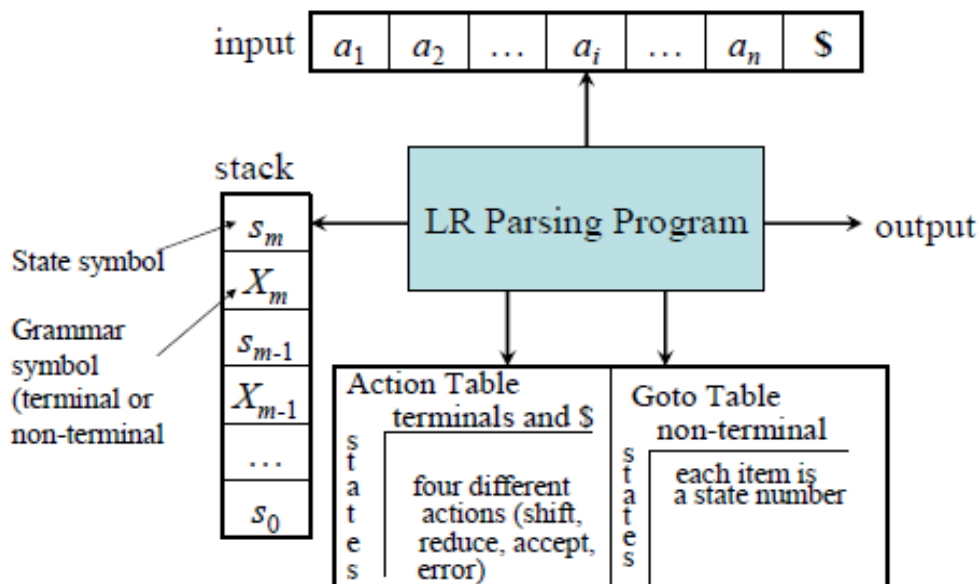
LR-Parsers cover wide range of grammars.

- SLR – simple LR parser
- LR – most general LR parser
- LALR – intermediate LR parser (look-head LR parser)

SLR, LR and LALR work same (they used the same algorithm), only their parsing tables are different.



LR Parsers: General Structure



Constructing SLR Parsing Tables

For constructing a SLR parsing table of given grammar we need,

To construct the canonical LR(0) collection of the grammar, which uses the ‘closure’ operation and ‘goto’ operation.

LR(0) Item:

An LR(0) item of a grammar G is a production of G with a dot (.) at some position of the right side.

Eg: the production

$A \xrightarrow{\quad} aBb$ yields the following 4 possible LR(0) items which are:
 $A \xrightarrow{\quad} .aBb$
 $A \xrightarrow{\quad} a.Bb$
 $A \xrightarrow{\quad} aB.b$
 $A \xrightarrow{\quad} aBb.$

Note: - The production $A \xrightarrow{\quad} \epsilon$ generates only one LR(0) item, $A \xrightarrow{\quad} .$

canonical LR(0) collection:

A collection of sets of LR(0) items is called canonical LR(0) collection.

To construct canonical LR(0) collection for a grammar we require augmented grammar and closure & goto functions.

Augmented grammar:

If G is a grammar with start symbol S, then the augmented grammar G' of G is a grammar with a new start symbol S' and production $S' \xrightarrow{\quad} S$

Eg: the grammar,

$E \rightarrow E + T \mid T$
 $T \rightarrow T * F \mid F$
 $F \rightarrow (E) \mid id$

Its augmented grammar is;

$E' \xrightarrow{\quad} E$
 $E \rightarrow E + T \mid T$
 $T \rightarrow T * F \mid F$
 $F \rightarrow (E) \mid id$

The Closure Operation:

If **I** is a set of LR(0) items for a grammar G, then **closure(I)** is the set of LR(0) items constructed from **I** by the two rules:

1. Initially, every LR(0) item in **I** is added to closure(I).
2. If $A \rightarrow \alpha.B\beta$ is in closure(I) and $B \rightarrow \gamma$ is a production rule of G then add $B \rightarrow .\gamma$ in the closure(I) repeat until no more new LR(0) items added to closure(I).

The Closure Operation: Example

Consider a grammar:

$E \rightarrow E + T \mid T$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid \text{id}$$

Its augmented grammar is;

$$E' \rightarrow E$$

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid \text{id}$$

Now closure ($E' \rightarrow E$) contains the following items:

$$E' \rightarrow .E$$

$$E \rightarrow .E + T$$

$$E \rightarrow .T$$

$$T \rightarrow .T * F$$

$$T \rightarrow .F$$

$$F \rightarrow .(E)$$

$$F \rightarrow .\text{id}$$

The goto Operation:

If I is a set of LR(0) items and X is a grammar symbol (terminal or non-terminal), then $\text{goto}(I, X)$ is defined as follows:

If $A \rightarrow \alpha X \beta$ in I then every item in $\text{closure}(\{A \rightarrow \alpha X \beta\})$ will be in $\text{goto}(I, X)$.

Example:

$$I = \{ E' \rightarrow .E, E \rightarrow .E+T, E \rightarrow .T, T \rightarrow .T*F, T \rightarrow .F, F \rightarrow .(E), F \rightarrow .\text{id} \}$$

$$\text{goto}(I, E) = \text{closure}(\{[E' \rightarrow E \cdot, E \rightarrow E \cdot + T]\}) = \{ E' \rightarrow E., E \rightarrow E.+T \}$$

$$\text{goto}(I, T) = \{ E \rightarrow T., T \rightarrow T.*F \}$$

$$\text{goto}(I, F) = \{ T \rightarrow F. \}$$

$$\text{goto}(I, () = \text{closure}(\{[F \rightarrow (\cdot E)]\})$$

$$= \{ F \rightarrow (.E), E \rightarrow .E+T, E \rightarrow .T, T \rightarrow .T*F, T \rightarrow .F, F \rightarrow .(E), F \rightarrow .\text{id} \}$$

$$\text{goto}(I, \text{id}) = \{ F \rightarrow \text{id}. \}$$

Construction of canonical LR(0) collection

Algorithm:

Augment the grammar by adding production $S' \rightarrow S$

$$C = \{ \text{closure}(\{S' \rightarrow .S\}) \}$$

repeat the followings until no more set of LR(0) items can be added to C .

for each I in C and each grammar symbol X

if $\text{goto}(I, X)$ is not empty and not in C

add $\text{goto}(I, X)$ to C

Example: The augmented grammar is:

$$C' \rightarrow C$$

$$C \rightarrow AB$$

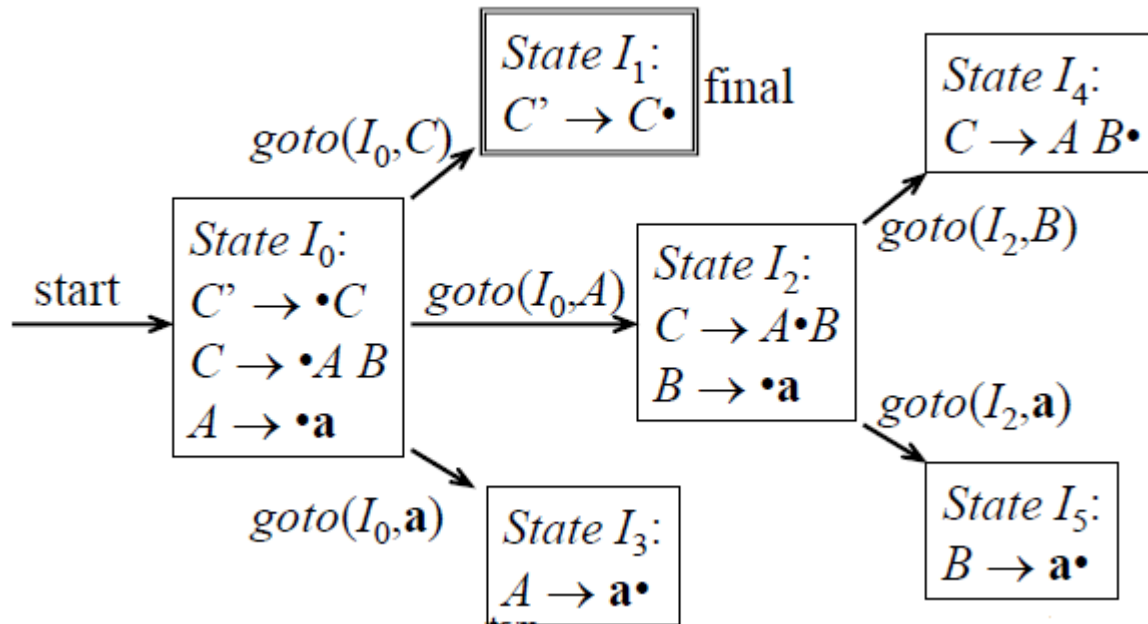
$$A \rightarrow a$$

$$B \rightarrow a$$

$$I_0 = \text{closure}(C' \rightarrow \bullet C)$$

$$I_1 = \text{goto}(I_0, C) = \text{closure}(C' \rightarrow C \bullet)$$

and so on



Example 2: Find canonical LR (0) collection for the following grammar:

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

Solution -----do itself-----

Constructing SLR Parsing Tables

Algorithm

- Construct the canonical collection of sets of LR(0) items for G' .
 $C \leftarrow \{I_0 \dots I_n\}$
- Create the parsing action table as follows
 - If $A \rightarrow \alpha.a\beta$ is in I_i and $\text{goto}(I_i, a) = I_j$ then set $\text{action}[i, a] = \text{shift } j$.
 - If $A \rightarrow \alpha.$ is in I_i , then set $\text{action}[i, a]$ to "reduce $A \rightarrow \alpha$ " for all 'a' in FOLLOW(A) where $A \neq S'$.
 - If $S' \rightarrow S.$ is in I_i , then $\text{action}[i, \$] = \text{accept}$.
 - If any conflicting actions generated by these rules, the grammar is not SLR(1).
- Create the parsing goto table
 - for all non-terminals A, if $\text{goto}(I_i, A) = I_j$ then $\text{goto}[i, A] = j$
- All entries not defined by (2) and (3) are errors.
- Initial state of the parser contains $S' \rightarrow .S$

Example: Construct the SLR parsing table for the grammar:

$C \rightarrow AB$

$A \rightarrow a$

$B \rightarrow a$

Soln: The augmented grammar of given grammar is:

1). $C' \rightarrow C$

2). $C \rightarrow AB$

3). $A \rightarrow a$

4). $B \rightarrow a$

Step 1:- construct the canonical LR(0) collection for the grammar as,

State I_0 :

$\text{closure}(C' \rightarrow \cdot C)$

$C' \rightarrow \cdot C$

$C \rightarrow \cdot AB$

$A \rightarrow \cdot a$

State I_1 :

$\text{closure}(\text{goto}(I_0, C))$

$\text{closure}(C' \rightarrow C \cdot)$

$C' \rightarrow C \cdot$

State I_2 :

$\text{closure}(\text{goto}(I_0, A))$

$\text{closure}(C \rightarrow A \cdot B)$

$C \rightarrow A \cdot B$

$B \rightarrow \cdot a$

State I_3 :

$\text{closure}(\text{goto}(I_0, a))$

$\text{closure}(A \rightarrow a \cdot)$

$A \rightarrow a \cdot$

State I_4 :

$\text{closure}(\text{goto}(I_2, B))$

$\text{closure}(C \rightarrow AB \cdot)$

$C \rightarrow AB \cdot$

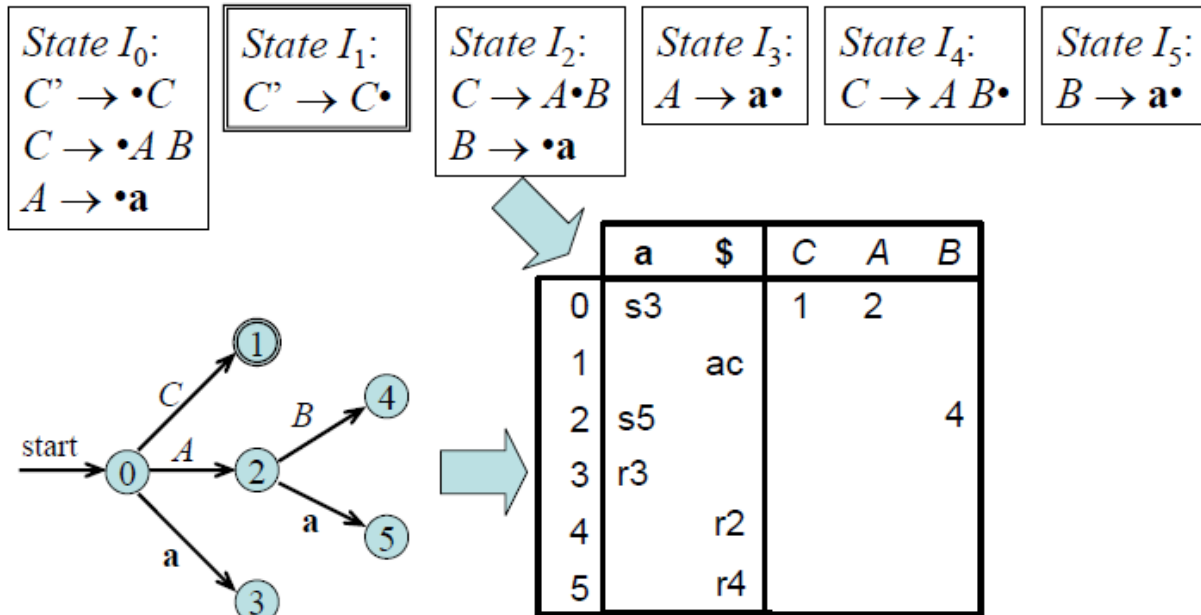
State I_5 :

$\text{closure}(\text{goto}(I_2, a))$

$\text{closure}(B \rightarrow a \cdot)$

$B \rightarrow a \cdot$

Step 2 : Construct SLR parsing table that contains both action and goto table as follows:



LR(1) Grammars

SLR is so simple and can only represent the small group of grammar

LR(1) parsing uses look-ahead to avoid unnecessary conflicts in parsing table

LR(1) item = LR(0) item + look-ahead

LR(0) item: LR(1) item:
 $[A \rightarrow \alpha \bullet \beta]$ $[A \rightarrow \alpha \bullet \beta, a]$

Constructing LR(1) Parsing Tables

Computation of Closure for LR(1) Items:

1. Start with $\text{closure}(I) = I$ (where I is a set of LR(1) items)
2. If $[A \rightarrow \alpha \bullet B \beta, a] \in \text{closure}(I)$ then
 add the item $[B \rightarrow \bullet \gamma, b]$ to I if not already in I , where $b \in \text{FIRST}(\beta a)$.
3. Repeat 2 until no new items can be added.

Computation of Goto Operation for LR(1) Items:

If I is a set of LR(1) items and X is a grammar symbol (terminal or non-terminal), then $\text{goto}(I, X)$ is computed as follows:

1. For each item $[A \rightarrow \alpha \bullet X \beta, a] \in I$, add the set of items
 $\text{closure}(\{[A \rightarrow \alpha X \bullet \beta, a]\})$ to $\text{goto}(I, X)$ if not already there
2. Repeat step 1 until no more items can be added to $\text{goto}(I, X)$

Construction of The Canonical LR(1) Collection:

Algorithm:

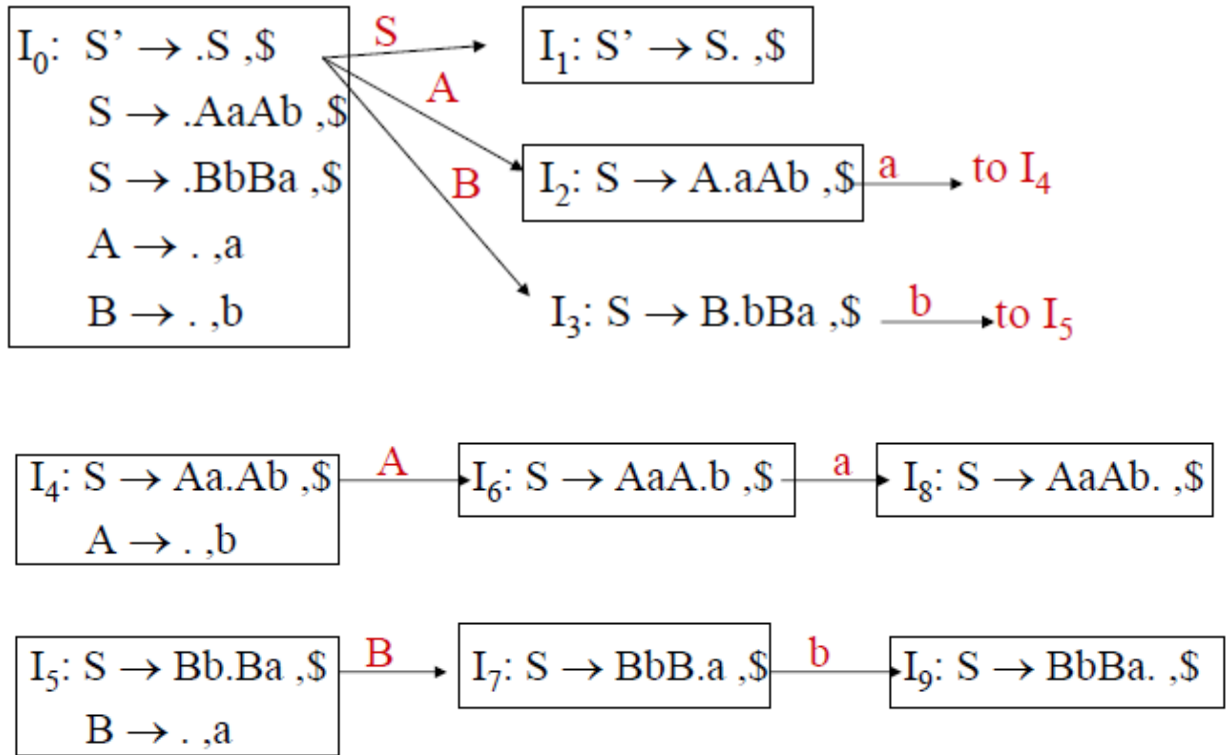
 Augment the grammar with production $S' \rightarrow S$
 $C = \{ \text{closure}(\{S' \rightarrow \cdot S, \$\}) \}$ (the start stat of DFA)
 repeat the followings until no more set of LR(1) items can be added to C .
 for each $I \in C$ and each grammar symbol $X \in (N \cup T)$
 $\text{goto}(I, X) \neq \emptyset$ and $\text{goto}(I, X) \notin C$ then
 add $\text{goto}(I, X)$ to C

Example: Construct canonical LR(1) collection of the grammar:

$S \rightarrow AaAb$
 $S \rightarrow BbBa$
 $A \rightarrow \epsilon$
 $B \rightarrow \epsilon$

Its augmented grammar is:

$S' \rightarrow S$
 $S \rightarrow AaAb$
 $S \rightarrow BbBa$
 $A \rightarrow \epsilon$
 $B \rightarrow \epsilon$



Constructing LR(1) Parsing Tables

Algorithm:

1. Construct the canonical collection of sets of LR(1) items for G' .
 $C = \{I_0 \dots I_n\}$
2. Create the parsing action table as follows
 - If $[A \rightarrow \alpha.a\beta, b]$ in I_i and $\text{goto}(I_i, a) = I_j$ then $\text{action}[i, a] = \text{shift } j$.
 - If $A \rightarrow \alpha., a$ is in I_i , then $\text{action}[i, a] = \text{reduce } A \rightarrow \alpha$ where $A \neq S'$.
 - If $S' \rightarrow S., \$$ is in I_i , then $\text{action}[i, \$] = \text{accept}$.
 - If any conflicting actions generated by these rules, the grammar is not LR(1).
3. Create the parsing goto table
 - for all non-terminals A , if $\text{goto}(I_i, A) = I_j$ then $\text{goto}[i, A] = j$
4. All entries not defined by (2) and (3) are errors.
5. Initial state of the parser contains $S' \rightarrow .S, \$$

LR(1) Parsing Tables: Example

Construct LR(1) parsing table for the augmented grammar,

1. $S' \rightarrow S$
2. $S \rightarrow L = R$
3. $S \rightarrow R$
4. $L \rightarrow * R$
5. $L \rightarrow \text{id}$
6. $R \rightarrow L$

Step 1: At first find the canonical collection of LR(1) items of the given augmented grammar as,

State I₀:

closure($S' \rightarrow .S, \$$)
 $S' \rightarrow .S, \$$
 $S \rightarrow .L = R, \$$
 $S \rightarrow .R, \$$
 $L \rightarrow .*R, \$$
 $L \rightarrow .Id, =$
 $R \rightarrow .L, \$$

State I₁:

closure (goto(I_0, S))
 closure($S' \rightarrow S., \$$)
 $S' \rightarrow S., \$$

State I₂:

closure (goto(I_0, L))
 closure($(S \rightarrow L. = R, \$), (R \rightarrow L., \$)$)
 $S \rightarrow L. = R, \$$
 $R \rightarrow L., \$$

State I₃:

closure (goto(I_0, R))
 closure($S \rightarrow R., \$$)
 $S \rightarrow R., \$$

State I₄:

closure(goto($I_0, *$))
 closure($L \rightarrow *.R, =$)
 $\{(L \rightarrow *.R, =), (R \rightarrow .L, =), (L \rightarrow .*R, =), (L \rightarrow .Id, =)\}$

State I₅:

closure (goto(I_0, id))
 closure($L \rightarrow Id., =$)
 $L \rightarrow Id., =$

State I₆:

closure(goto($(I_2, =)$))
 closure($S \rightarrow L. = R, \$$)
 $S \rightarrow L. = R, \$$
 $R \rightarrow .L, \$$
 $L \rightarrow .*R, \$$
 $L \rightarrow .Id, \$$

State I₇:

closure(goto((I_4, R)))
 closure($L \rightarrow *.R., =$)
 $L \rightarrow *.R., =$

State I₈:

closure (goto(I_4, L))
 closure($R \rightarrow L., =$)
 $R \rightarrow L., =$

State I₉:

closure(goto((I_6, R)))
 closure($S \rightarrow L=R., \$$)
 $S \rightarrow L=R., \$$

State I₁₀:

closure(goto((I_6, L)))
 $R \rightarrow L., \$$

State I₁₁:

Closure(goto($I_6, *$))
 Closure($L \rightarrow *.R, \$$)
 $L \rightarrow *.R, \$$
 $R \rightarrow .L, \$$
 $L \rightarrow .*R, \$$
 $L \rightarrow .id, \$$

State I₁₂:

closure(goto((I_6, id)))
 closure($L \rightarrow id., \$$)
 $L \rightarrow id., \$$

State I₁₃:

closure(goto((I_{11}, R)))
 $L \rightarrow *R., \$$

Step 2: Now construct LR(1) parsing table

	id	*	=	S	S	L	R
0	s5	s4			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s5	s4				8	7
5			r4	r4			
6	s12	s11				10	9
7			r3	r3			
8			r5	r5			
9				r1			
10				r5			
11	s12	s11				10	13
12				r4			
13				r3			

LALR(1) Grammars

It is an intermediate grammar between the SLR and LR(1) grammar.

A typical programming language generates thousands of states for canonical LR parsers while they generate only hundreds of states for LALR parser.

LALR(1) parser combines two or more LR(1) sets(whose core parts are same) into a single state to reduce the table size.

Example:

$I_1: L \rightarrow id, =$

$I_{12}: L \rightarrow id, =$

$I_2: L \rightarrow id, \$$

$L \rightarrow id, \$$

Constructing LALR Parsing Tables

1. Create the canonical LR(1) collection of the sets of LR(1) items for the given grammar $C = \{I_0, \dots, I_n\}$.

2. Find each core; find all sets having that same core; replace those sets having same cores with a single set which is their union.

$C = \{I_0, \dots, I_n\}$ then $C' = \{J_1, \dots, J_m\}$ where $m \leq n$

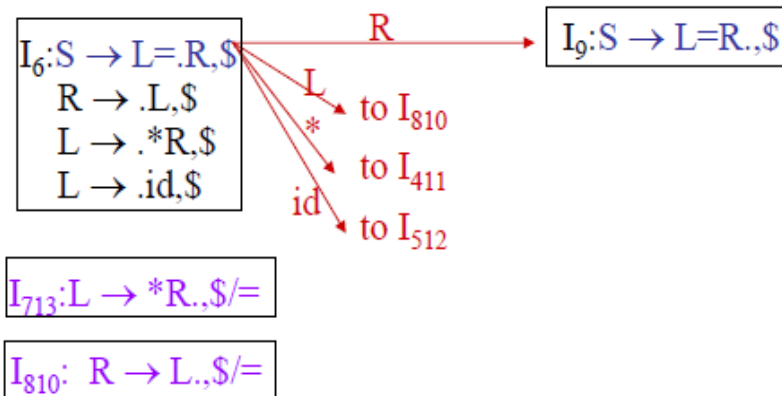
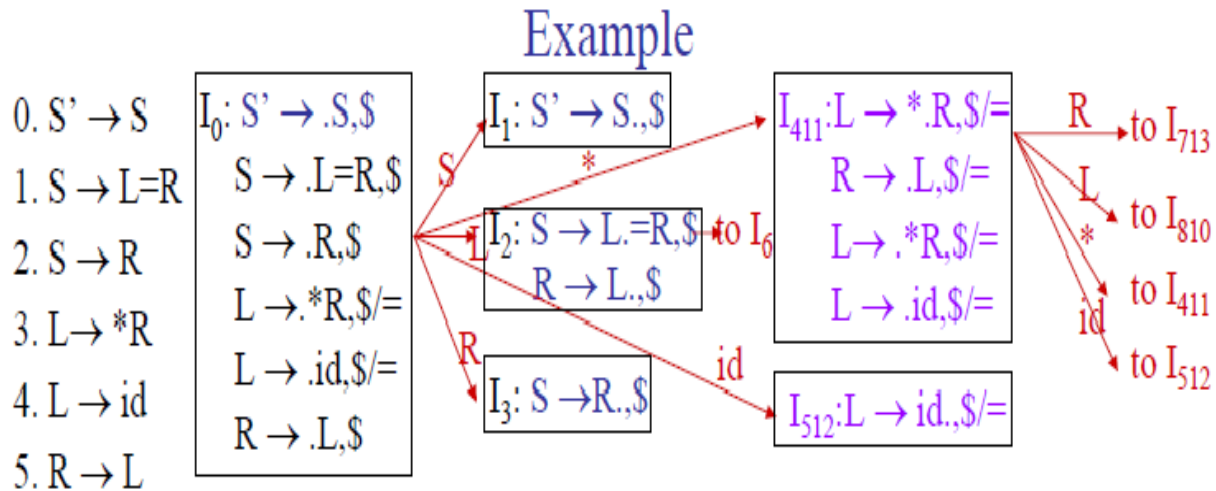
3. Create the parsing tables (action and goto tables) same as the construction of the parsing tables of LR(1) parser.

– Note that: If $J = I_1 \cup \dots \cup I_k$ since I_1, \dots, I_k have same cores then cores of $\text{goto}(I_1, X), \dots, \text{goto}(I_k, X)$ must be same.

– So, $\text{goto}(J,X)=K$ where K is the union of all sets of items having same cores as $\text{goto}(I_1,X)$.

4. If no conflict is introduced, the grammar is LALR(1) grammar.

(We may only introduce reduce/reduce conflicts; we cannot introduce a shift/reduce conflict)



Same Cores
 I_4 and I_{11}

I_5 and I_{12}

I_7 and I_{13}

I_8 and I_{10}

Grammar:

1. $S' \rightarrow S$
2. $S \rightarrow L = R$
3. $S \rightarrow R$
4. $L \rightarrow * R$
5. $L \rightarrow \text{id}$
6. $R \rightarrow L$

	id	*	=	S	S	L	R
0	s5	s4			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s5	s4				8	7
5			r4	r4			
6	s12	s11				10	9
7			r3	r3			
8			r5	r5			
9				r1			

no shift/reduce or
no reduce/reduce
conflict



so, it is a LALR
grammar

Kernel item:

This includes the initial items, $S' \rightarrow S$ and all items whose dot are not at the left end.

Non-Kernel item:

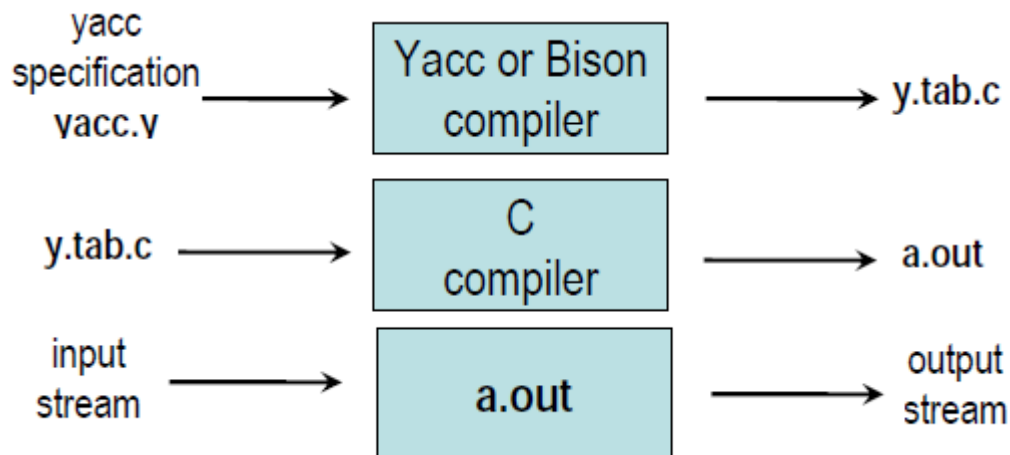
The productions of a grammar which have their dots at the left end are non-kernel items.

Parser Generators:

Introduction to Bison

Bison is a general purpose parser generator that converts a description for an *LALR(1)* context-free grammar into a C program file.

- ✓ The job of the Bison parser is to group tokens into groupings according to the grammar rules—for example, to build identifiers and operators into expressions.
- ✓ The tokens come from a function called the lexical analyzer that must supply in some fashion (such as by writing it in C).
- ✓ The Bison parser calls the lexical analyzer each time it wants a new token. It doesn't know what is “inside” the tokens.
- ✓ Typically the lexical analyzer makes the tokens by parsing characters of text, but Bison does not depend on this.
- ✓ The Bison parser file is C code which defines a function named *yyparse* which implements that grammar. This function does not make a complete C program: you must supply some additional functions.



Stages in Writing Bison program

1. Formally specify the grammar in a form recognized by Bison
2. Write a lexical analyzer to process input and pass tokens to the parser.
3. Write a controlling function that calls the Bison produced parser.
4. Write error-reporting routines.

Bison Specification

- A bison specification consists of four parts:

```

%{
    C declarations
}%
Bison declarations
%%
Grammar rules
%%
Additional C codes
Productions in Bison are of the form
Non-terminal: tokens/non-terminals {action}
              | Tokens/non | terminals {action}
              .....
              ;
  
```

Bison Declaration

Tokens that are single characters can be used directly within productions, e.g. '+', '-', '*'

Named tokens must be declared first in the declaration part using

```

%token Token Name (Upper Case Letter)
e.g    %token INTEGER IDENTIFIER
        %token NUM 100
  
```

– %left, %right or %nonassoc can be used instead for %token to specify the precedence & associativity (precedence declaration). All the tokens declared in a single precedence declaration have equal precedence and nest together according to their associativity.

- `%union` declares the collection data types
- `%type <non-terminal>` declares the type of semantic values of non-terminal
- `%start <non-terminal>` specifies the grammar start symbol (by default the start symbol of grammar)

Grammar Rules

- ✓ In order for Bison to parse a grammar, it must be described by a *Context-Free Grammar* that is LALR (1).
- ✓ A non-terminal in the formal grammar is represented in Bison input as an identifier, like an identifier in C. By convention, it is in *lower case*, such as *expr*, *declaration*.
- ✓ A Bison grammar rule has the following general form:
 - o `RESULT: COMPONENTS...;`
 where, `RESULT` is the non-terminal symbol that this rule describes and `COMPONENTS` are various terminal and non-terminal symbols that are put together by this rule.
 For example, `exp: exp '+' exp;` says that two groupings of type 'exp', with a '+' token in between, can be combined into a larger grouping of type 'exp'.
- ✓ Multiple rules for the same `RESULT` can be written separately or can be joined with the vertical-bar character '|' as follows:


```
RESULT:    RULE1-COMPONENTS...
          | RULE2-COMPONENTS...
          .....
          ;
```
- ✓ If `COMPONENTS` in a rule is empty, it means that `RESULT` can match the empty string. For example, here is how to define a comma-separated sequence of zero or more 'exp' groupings:


```
expseq:    /* empty */
          | expseq1
          ;
expseq1:   exp
          | expseq1 ',' exp
          ;
```

It is customary to write a comment '`/* empty */`' in each rule with no components.

Semantic Actions:

To make program useful, it must do more than simply parsing the input, i.e., must produce some output based on the input.

Most of the time the action is to compute semantics value of whole constructs from the semantic values associated with various tokens and groupings.

For Example, here is a rule that says an expression can be the sum of two sub-expression,

```
expr:  expr '+' expr { $$ = $1 + $3; }
      ;
```

The action says how to produce the semantic value of the sum expression from the value of two sub expressions.

In bison, the default data type for all semantics is int. i.e. the parser stack is implemented as an integer array. It can be overridden by redefining the macro YYSTYPE. A line of the form `#defines YYSTYPE double` in the C declarations section of the bison grammar file.

To use multiple types in the parser stack, a “union” has to be declared that enumerates all possible types used in the grammar.

Example:

```
%union{
    double val;
    char *str;
}
```

This says that there are two alternative types: double and char*.

Tokens are given specific types by a declaration of the form:

```
%token <val> exp
```

Interfacing with Flex

Bison provides a function called `yyparse()` and *expects* a function called `yylex()` that performs lexical analysis. Usually this is written in lex. If not, then `yylex()` should be written in the C code area of bison itself.

If `yylex()` is written in flex, then bison should first be called with the `-d` option: `bison -d grammar.y`

This creates a file `grammar.tab.h` that contains `#defines` of all the `%token` declarations in the grammar. The sequence of invocation is hence:

```
bison -d grammar.y
flex grammar.flex
gcc -o grammar grammar.tab.c lex.yy.c -lfl
```

Practice

- Get familiar with Bison: Write a desk calculator which performs '+' and '*' on unsigned integers
 1. Create a Directory: "mkdir calc"
 2. Save the five files ([calc.lex](#), [calc.y](#), [Makefile](#), [main.cc](#), and [heading.h](#)) to directory "calc"
 3. Command Sequence: "make"; "./calc"
 4. Use input programs (or stdin) which contain expressions with integer constants and operators + and *, then press Ctrl-D to see the result

Programming Example

```
/* Mini Calculator */
/* calc.lex */
%{
    #include "heading.h"
    #include "tok.h"
    int yyerror(char *s);
    int yylineno = 1;
```

```

%}

digit      [0-9]
int_const  {digit}+

%%
{int_const}{ yylval.int_val = atoi(yytext); return
INTEGER_LITERAL; }
"+"      { yylval.op_val = new std::string(yytext); return
PLUS; }
"*"      { yylval.op_val = new std::string(yytext); return
MULT; }

[\\t]*      {}
[\\n]      { yylineno++; }

.          { std::cerr << "SCANNER "; yyerror(""); exit(1); }

%%
-----
-

/* Mini Calculator */
/* calc.y */
%{
    #include "heading.h"
    int yyerror(char *s);
    int yylex(void);
%}

%union{
    int      int_val;
    string*  op_val;
}

%start      input
%token      <int_val> INTEGER_LITERAL
%type       <int_val> exp
%left      PLUS
%left      MULT

%%

input:      /* empty */
            | exp      { cout << "Result: " << $1 << endl; }
            ;

```

```
exp:      INTEGER_LITERAL      { $$ = $1; }
      | exp PLUS exp { $$ = $1 + $3; }
      | exp MULT exp { $$ = $1 * $3; }
      ;
```

```
%%
```

```
int yyerror(string s)
{
    extern int yylineno;      // defined and maintained in lex.c
    extern char *yytext;      // defined and maintained in lex.c
    cerr << "ERROR: " << s << " at symbol \"" << yytext;
    cerr << "\" on line " << yylineno << endl;
    exit(1);
}

int yyerror(char *s)
{
    return yyerror(string(s));
}
```

unit: 3

Syntax Directed Translation

To translate a programming language construct, compiler needs to keep track of many quantities to the grammar symbol.

There are two notations for associating semantic rules with productions, which are:

- Syntax- directed definitions and
- Translation schema.

Syntax-directed definition:

Syntax-Directed Definitions are high level specifications for translations. They hide many implementation details and free the user from having to explicitly specify the order in which translation takes place.

A syntax-directed definition is a generalization of a context-free grammar in which each grammar symbol is associated with a set of attributes. This set of attributes for a grammar symbol is partitioned into two subsets **synthesized** and **inherited** attributes of that grammar.

Mathematically,

Given a production

$$A \rightarrow \alpha$$

then each semantic rule is of the form

$$b = f(c_1, c_2, \dots, c_k)$$

where f is a function and c_i are attributes of A and α , and either

- b is a *synthesized* attribute of A
- b is an *inherited* attribute of one of the grammar symbols in α .

Example: The syntax directed definition for a simple desk calculator

Production	Semantic Rules
$L \rightarrow E \text{ return}$	$\text{print}(E.\text{val})$
$E \rightarrow E_1 + T$	$E.\text{val} = E_1.\text{val} + T.\text{val}$
$E \rightarrow T$	$E.\text{val} = T.\text{val}$
$T \rightarrow T_1 * F$	$T.\text{val} = T_1.\text{val} * F.\text{val}$
$T \rightarrow F$	$T.\text{val} = F.\text{val}$
$F \rightarrow (E)$	$F.\text{val} = E.\text{val}$
$F \rightarrow \text{digit}$	$F.\text{val} = \text{digit}.\text{lexval}$

Note: all attributes in this example are of the synthesized type.

Annotated Parse Tree:

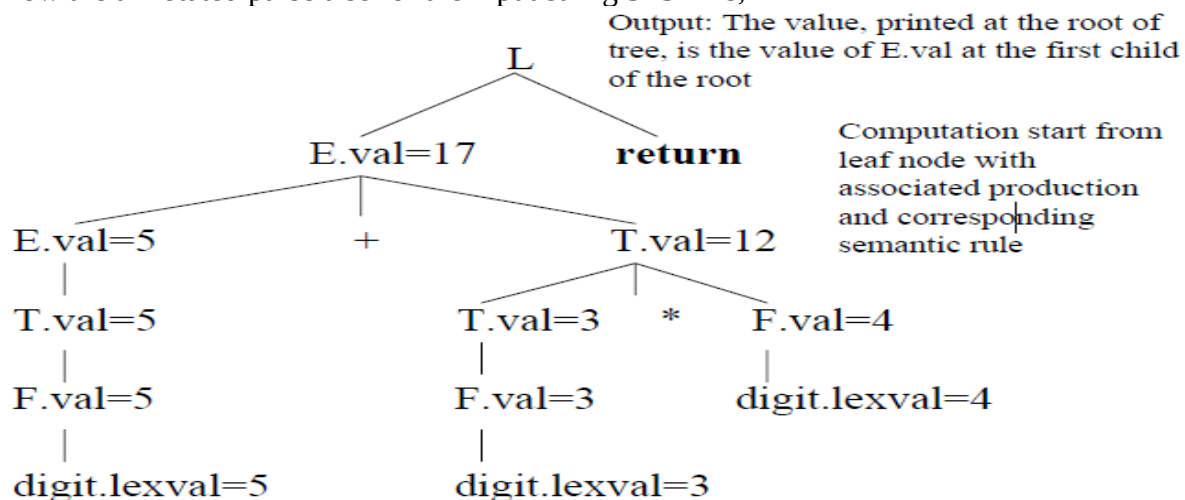
A parse tree constructing for a given input string in which each node showing the values of attributes is called an annotated parse tree.

Example:

Let's take a grammar,

$L \rightarrow E \text{ return}$
 $E \rightarrow E_1 + T$
 $E \rightarrow T$
 $T \rightarrow T_1 * F$
 $T \rightarrow F$
 $F \rightarrow (E)$
 $F \rightarrow \text{digit}$

Now the annotated parse tree for the input string 5+3*4 is,



Inherited and Synthesized Attributes:

A node in which attributes are derived from the parent or siblings of the node is called inherited attribute of that node.

The attributes of a node that are derived from its children nodes are called synthesized attributes.

Example:

<u>Production</u>	<u>Semantic Rules</u>	
$D \rightarrow T L$	$L.in = T.type$	inherited
$T \rightarrow \text{int}$	$T.type = \text{integer}$	
$T \rightarrow \text{real}$	$T.type = \text{real}$	synthesized
$L \rightarrow L_1 \text{ id}$	$L_1.in = L.in, \text{ addtype}(\text{id.entry}, L.in)$	
$L \rightarrow \text{id}$	$\text{addtype}(\text{id.entry}, L.in)$	

Dependency Graph:

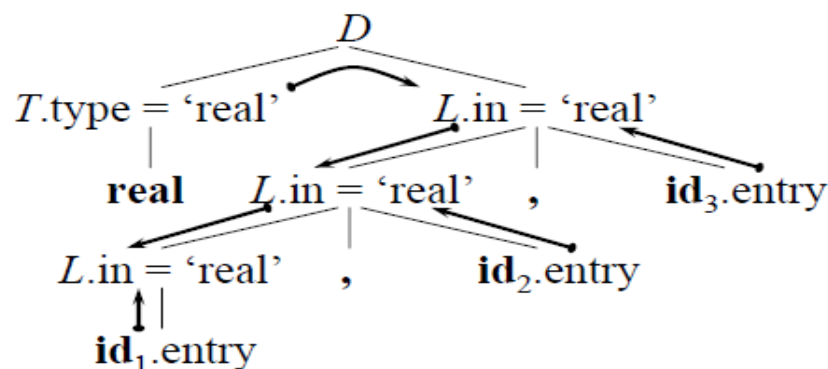
If interdependencies among the inherited and synthesized attributes in an annotated parse tree are specified by arrows then such a tree is called dependency graph.

In order to correctly evaluate attributes of syntax tree nodes, a *dependency graph* is useful. A dependency graph is a directed graph that contains attributes as nodes and dependencies across attributes as edges.

Example: let's take a grammar,

$D \rightarrow T L$
 $T \rightarrow \text{int}$
 $T \rightarrow \text{real}$
 $L \rightarrow L_1 \text{ id}$
 $L \rightarrow \text{id}$

Input : real id1 , id2 , id3



S-Attributed Definitions:

- ✓ A syntax-directed definition that uses synthesized attributes exclusively is called an *S-attributed definition* (or *S-attributed grammar*).
- ✓ A parse tree of an S-attributed definition is annotated by evaluating the semantic rules for the attribute at each node in bottom-up manner.
- ✓ Yacc/Bison only support S-attributed definitions.

Example: Bottom up evaluation of S-Attributed definition:

Let's take a grammar:

$L \rightarrow E \mathbf{n}$
 $E \rightarrow E1 + T$
 $E \rightarrow T$
 $T \rightarrow T1 * F$
 $T \rightarrow F$
 $F \rightarrow (E)$
 $F \rightarrow \mathbf{digit}$

Input: 3*5+4 n

Stack	val	Input	Action
S	—	3*5+4nS	shift
S 3	3	*5+4nS	reduce $F \rightarrow \mathbf{digit}$
S F	3	*5+4nS	reduce $T \rightarrow F$
S T	3	*5+4nS	shift
S T *	3 —	5+4nS	shift
S T * 5	3 — 5	+4nS	reduce $F \rightarrow \mathbf{digit}$
S T * F	3 — 5	+4nS	reduce $T \rightarrow T * F$
S T	15	+4nS	reduce $E \rightarrow T$
S E	15	+4nS	shift
S E +	15 —	4nS	shift
S E + 4	15 — 4	nS	reduce $F \rightarrow \mathbf{digit}$
S E + F	15 — 4	nS	reduce $T \rightarrow F$
S E + T	15 — 4	nS	reduce $E \rightarrow E + T$
S E	19	nS	shift
S E n	19 —	S	reduce $L \rightarrow E \mathbf{n}$
S L	19	S	accept

L-Attributed Definitions:

An inherited attribute which can be evaluated in a left-to right fashion is called a L-attributed definition.

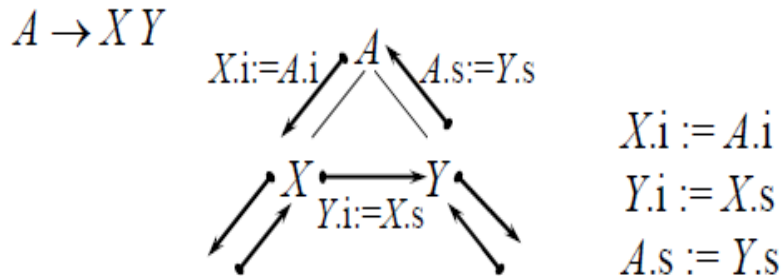
L-attributed definitions can be evaluated using a *depth-first* evaluation order. Mathematically,

A syntax-directed definition is *L-attributed* if each inherited attribute of X_j , $1 \leq j \leq n$ on right side of $A \rightarrow X_1 X_2 \dots X_n$ and it depends on,

1. The attributes of the symbols X_1, X_2, \dots, X_{j-1} to the left of X_j and
2. The inherited attributes of A .

Every S-attributed definition is L-attributed; the restrictions only apply to the inherited attributes (not to synthesized attributes).

Example:

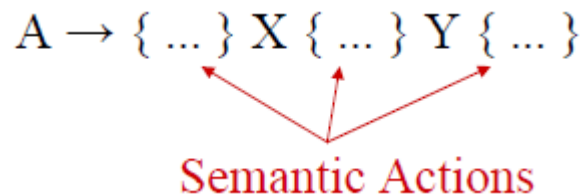


Translation Schema:

A **translation scheme** is a context-free grammar in which:

- Attributes are associated with the grammar symbols and
- Semantic actions are inserted within the right sides of productions and are enclosed between braces $\{ \}$.

Example:



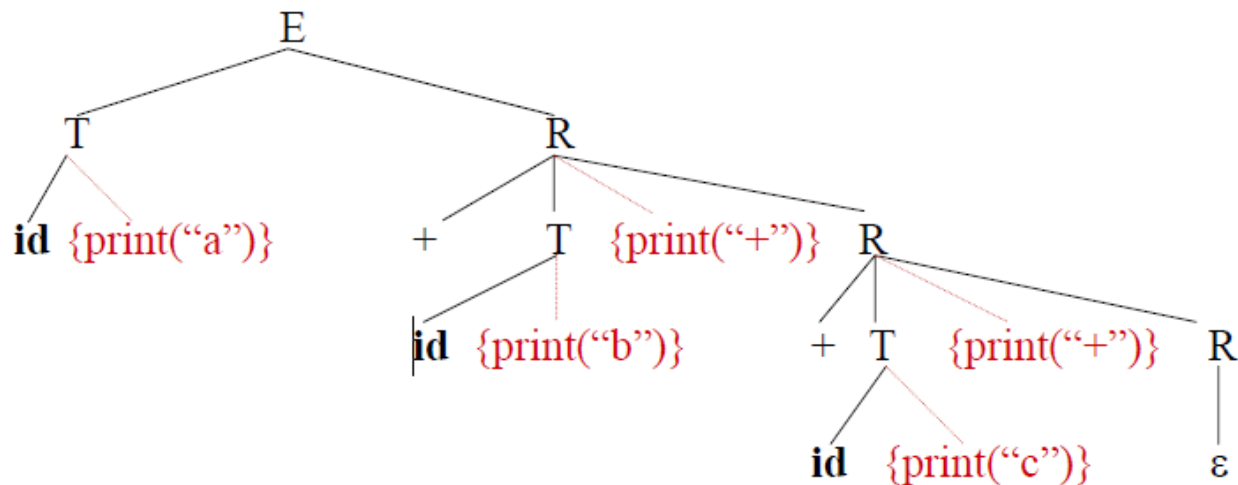
Q:- Define the syntax directed definitions with an example. How definitions are different from translation schemas?

- ✓ In syntax directed definition each grammar symbols associated with the semantic rules while in translation schema we use semantic actions instead of semantic rules.

Example: A simple translation schema that converts infix expression into corresponding postfix expressions.

$$\begin{aligned}
 E &\rightarrow T R \\
 R &\rightarrow + T \{ \text{print}("+") \} R_1 \\
 R &\rightarrow \varepsilon \\
 T &\rightarrow \text{id} \{ \text{print}(\text{id.name}) \}
 \end{aligned}$$

$a+b+c$ \rightarrow $ab+c+$
 infix expression postfix expression



Eliminating Left Recursion from a Translation Scheme

Let us take a left recursive translation schema:

$$A \rightarrow A1 Y \{ A.a = g(A1.a, Y.y) \}$$

$$A \rightarrow X \{ A.a = f(X.x) \}$$

In this grammar each grammar symbol has a synthesized attribute written using their corresponding lower case letters.

Now eliminating left recursion as,

$$A \rightarrow XR$$

$$R \rightarrow YR1$$

$$R \rightarrow \varepsilon$$

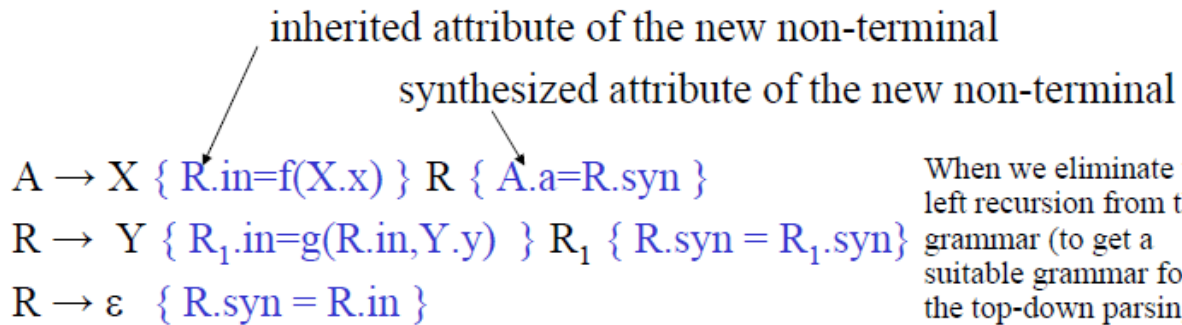
Hint: $A \rightarrow A\alpha \mid \beta$



$$A \rightarrow \beta A'$$

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

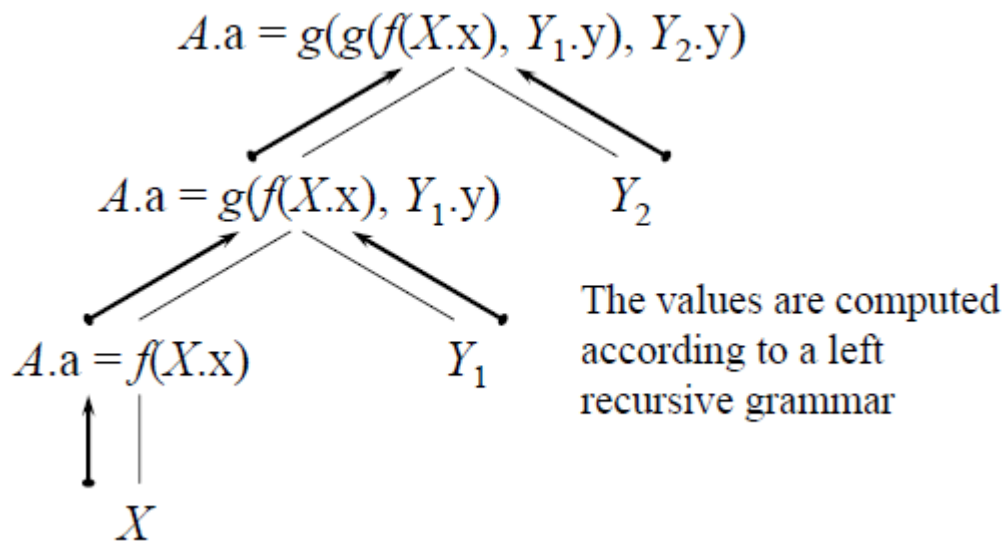
Now taking the new semantic actions for each symbols as follows,



When we eliminate the left recursion from the grammar (to get a suitable grammar for the top-down parsing) we also have to change semantic actions

Evaluating attributes

Evaluation of string XY₁Y₂



Q. For the following grammar:

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

Annotate the grammar with syntax directed definitions using synthesized attributes. Remove left recursion from the grammar and rewrite the attributes correspondingly.

-

Soln:

First part:

$E \rightarrow E + E \{ E.val = E1.val + E2.val \}$

$E \rightarrow E * E \{ E.val = E1.val * E2.val \}$

$E \rightarrow (E) \{ E.val = E1.val \}$

$E \rightarrow id \{ E.val = id.lexval \}$

Second part:

Removing left recursion as,

$$E \rightarrow (E)R \mid id R$$

$$R \rightarrow +ER1 \mid *ER1 \mid \epsilon$$

Now add the attributes within this non-left recursive grammar as,

$$E \rightarrow (E) \{R.in=E1.val\} R \{E.val=R.syn\}$$

$$E \rightarrow id \{R.in=id.lexval\} R1 \{E.val=R.syn\}$$

$$R \rightarrow +E \{R1.in=E.val+R.in\} R1 \{R.syn=R1.syn\}$$

$$R \rightarrow *E \{R1.in=E.val*R.in\} R1 \{R.syn=R1.syn\}$$

$$R \rightarrow \epsilon \{R.syn=R.in\}$$

Q. For the following grammar:

$$E \rightarrow E + E \mid E - E \mid T$$

$$T \rightarrow (E) \mid num$$

Annotate the grammar with syntax directed definitions using synthesized attributes. Remove left recursion from the grammar and rewrite the attributes correspondingly.

Soln: do itself

Q. For the following grammar:

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

At first remove the left recursion then construct an annotated parse tree for the expression $2*(3+5)$ using the modified grammar.

Trace the path in which semantic attributes are evaluated.

-

$$E \rightarrow E + E \{E.val = E1.val + E2.val\}$$

$$E \rightarrow E * E \{E.val = E1.val * E2.val\}$$

$$E \rightarrow (E) \{E.val = E1.val\}$$

$$E \rightarrow id \{E.val=id.lexval\}$$

Removing left recursion as,

$$E \rightarrow (E)R \mid id R$$

$$R \rightarrow +ER1 \mid *ER1 \mid \epsilon$$

Now add the attributes within this non-left recursive grammar as,

$$E \rightarrow (E) \{R.in=E1.val\} R \{E.val=R.syn\}$$

$$E \rightarrow id \{R.in=id.lexval\} R1 \{E.val=R.syn\}$$

$$R \rightarrow +E \{R1.in=E.val+R.in\} R1 \{R.syn=R1.syn\}$$

$$R \rightarrow *E \{R1.in=E.val*R.in\} R1 \{R.syn=R1.syn\}$$

$$R \rightarrow \epsilon \{R.syn=R.in\}$$

Second part:

An annotated parse tree for $2*(3+5)$ is,

$E.syn=16$

Id

R1

2	*	E	R.in=16
			ϵ
(E)	R.in=8
Id		R	
3			
	+	E	R.in=8
	Id		R.in=5
	5		