Course Title: Compiler Design and Construction
Course no: CSC-352
Credit hours: 3
Full Marks: 60+20+20
Pass Marks: 24+8+8

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: -6 Hrs.

1.1 Introduction to compiling: Compilers, Analysis of source program, the phases of compiler, compiler-construction tools.
(9 page)

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

Unit 2: -19 Hrs.

2.1 Lexical Analysis: The role of the lexical analyzer, Input buffering, Specification of tokens, Recognition of tokens, Finite Automata, Regular Expression to an NFA, Design of a lexical analyzer generator.
(24 page)

2.2 Syntax Analysis: The role of parser, Context free grammars, Writing a grammars, Top-down parsing, Bottom-up parsing, Operator-preceding parsing, LR parsing, Ambiguous grammar.

Scanned with CamScanner

Comp. remaining 16-29

2.3 Syntax Directed Translation : Syntax-directed definition, Syntax tree and its construction, Evaluation of S-attributed definitions, L-attributed, Top-down translation, Recursive evaluators
(8 page)

2.4 Type Checking: Type systems, specification of a simple type checker, Type conversions.
4 page

Unit 3: -13 Hrs.

3.1 Intermediate Code Generation: Intermediate languages, Declaration Assignments Statements, Boolean Expressions, Back Patching.

3.2 Code Generator: Issues in design of a code generator, the target machine, Run-time storage management, Basic blocks and flow graphs, a simple code generator, Peephole organization, Generating code from dags, Dynamic programming code-generating algorithm, Code generator generators.

3.3 Code optimization: The principal sources of optimization, optimization of basic blocks, loops in flow graphs.

Unit 4: -7 Hrs.

4.1 Writing a compilers: Planning a compiler, Approaches to compiler development, the compiler development environment, Testing and Maintenance.

4.2 Comparing some compilers: Pascal Compiler, C compiler, C++ compiler.

Scanned with CamScanner

# UNIT 1:

## 1.1) Introduction to Compiling

### Compiler :-

Compiler is a program that can read a program in one language - the source language and translates it into an equivalent program in another language - the target language. An important role of compiler is to report any errors in the source program that it detects during the translation process.

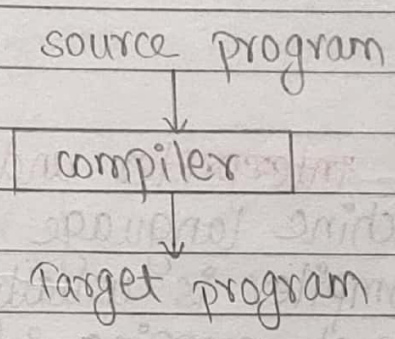



Fig:- Compiler

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

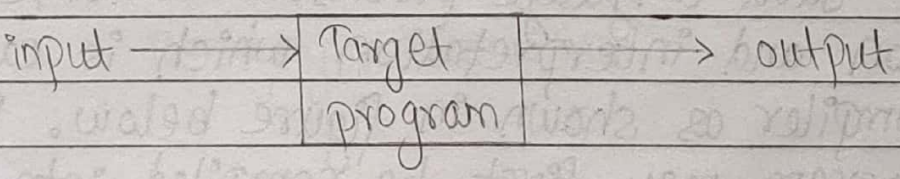



Fig:- Running the target program

Scanned with CamScanner

## Interpreter :-

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

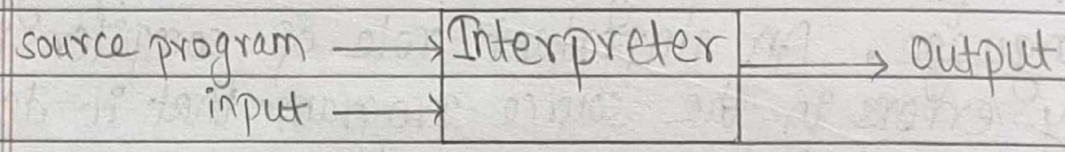



Fig:- An interpreter.

## Comparison between interpreter and compiler:

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

## Hybrid compiler:-

Java language processors combine compilation and interpretation which is called hybrid compiler as shown in figure below. A Java source program may first be compiled into an

Scanned with CamScanner

intermediate form called byte code. The byte codes are then interpreted by a virtual machine. The benefits of this arrangement is that byte codes compiled on one machine can be interpreted on another machine.

Inorder to achieve faster processing of inputs to outputs, some java compilers, called just-in-time compilers, translate the byte code into machine language immediately before they run the intermediate program to process the input.

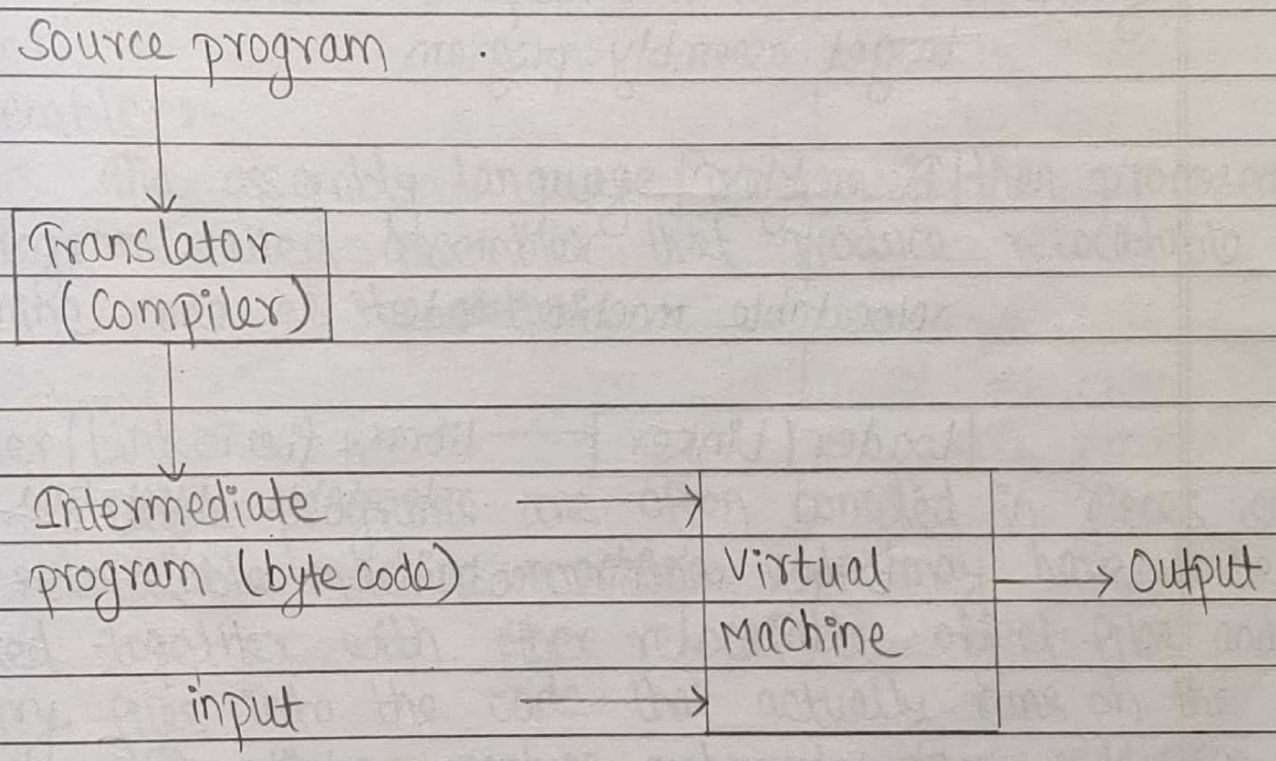



Fig:- Hybrid compiler

Scanned with CamScanner

## Language Processing System

In addition to a compiler, several other programs may be required to create an executable target program.

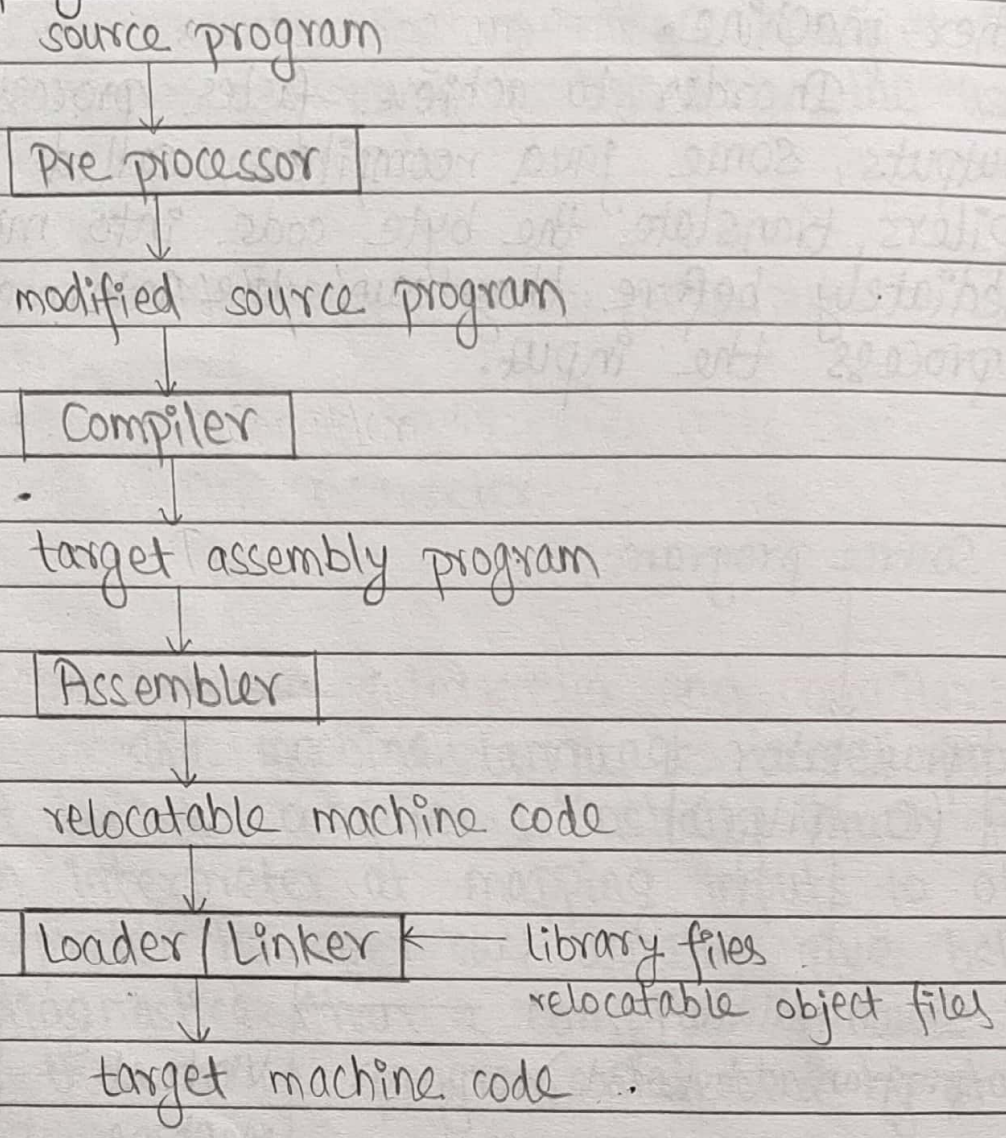



Fig:- A language processing system.

Scanned with CamScanner

Pre processor :-

A source program may be divided into modules stored in a seperate file. The task of collecting source program is sometimes assigned to a separate program called a preprocessor. The preprocessor may also expands short hands, called macros, into source language statements.

Compiler :-

The modified source program is then fed to a compiler. The compiler may produce an assembly language program as its output because assembly language is easier to produce as output and is easier to debug

Assembler :-

The assembly language program is then processed by a program called assembler that produce relocatable machine code as its output.

Loader / Linker :-

Large programs are often compiled in pieces so the relocatable machine machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine. The linker resolves external memory addresses where the code in one file may refer to a location in another file.

The loader then puts together all of the executable object files into memory for execution.

Scanned with CamScanner

# Structure of a compiler

## Major Components of a Compiler :-

A compiler is a program that maps a source program into a semantically equivalent target program. There are two major parts of this mapping.

1) Analysis
2) Synthesis

### 1) Analysis :

Analysis part breaks off the source program into constituent pieces and imposes grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill form or semantically unsound, then it must provide informative messages so that the user can take the corrective action. The analysis part also collects information about the source program and stores it into a data structure called symbol table which is passed along with the intermediate representation to the synthesis part.

### 2) Synthesis :

The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the front-end of the compiler

Scanned with CamScanner

and the synthesis part is called back-end.

Imp 6 marks

## Phases of compiler:

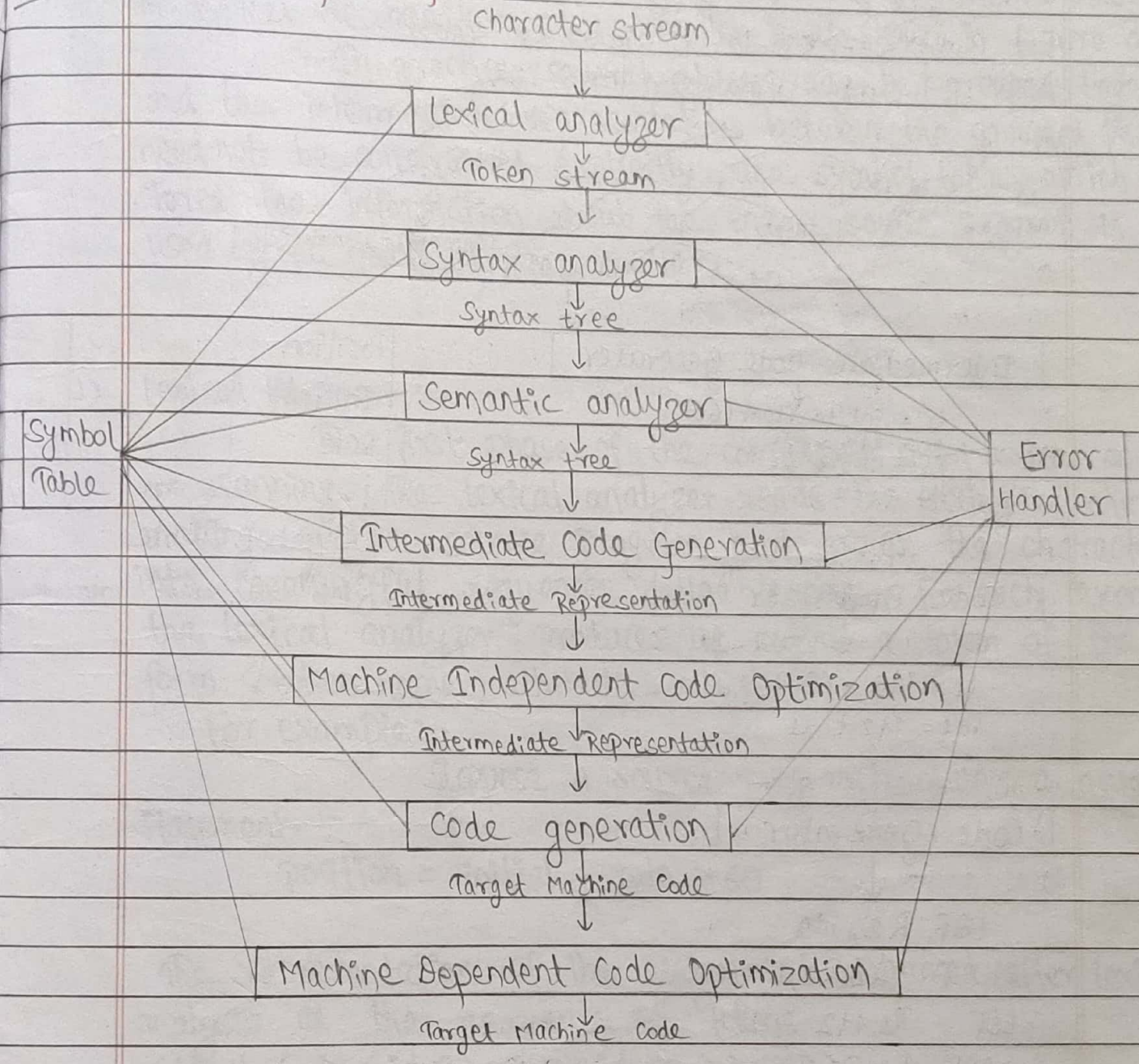



Fig:- Phases of compiler

Scanned with CamScanner

Position = initial + rate * 60

↓

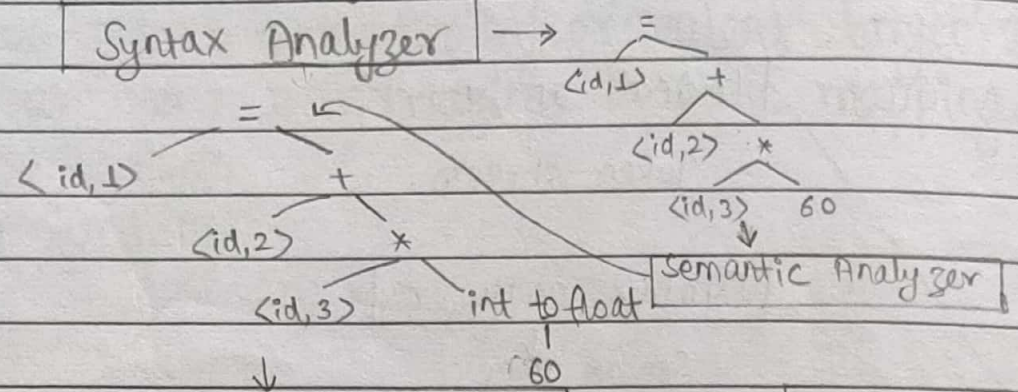
**Lexical Analyzer**

↓

<id,1> <==> <id,2> <+> <id,3> <*> <60>

↓

**Syntax Analyzer** →




↓

**Intermediate Code Generator**

↓

$t_1$ = int to float(60)
$t_2$ = id3 * $t_1$
$t_3$ = id2 * $t_2$
id1 = $t_2$

| Position | ------ |
|---|---|
| Initial | ------ |
| Rate | ------ |
| ⋮ | ------ |

Symbol Table
↳ information about source code

↓

**Code Optimizer**

↓

$t_1$ = id3 * 60.0
id1 = id2 + $t_1$

↓

**Code Generator**

↓

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

Fig:- Translation of an assignment statement

Scanned with CamScanner

If we examine that compilation process in more detail, we see that it operates as a sequence of phases, each of which transforms one representation of the source program to another. A typical decomposition of a compiler is as shown in figure above.

In practice, several phases may be grouped together and the intermediate representation between the grouped phases need not be constructed explicitly. The symbol table, which stores the information about the entire source program is used by all phases of the compiler.

1) Lexical Analyzer :-

The first phase of the compiler is called lexical analysis or scanning. The lexical analyzer reads the string of character making up the source program and groups the characters into meaningful sequences called lexemes. For each lexemes, the lexical analyzer produces as output a token of the form <token_name, attribute_value>

For example:-

Suppose a source program contains a assignment statement.

position = initial + rate * 60

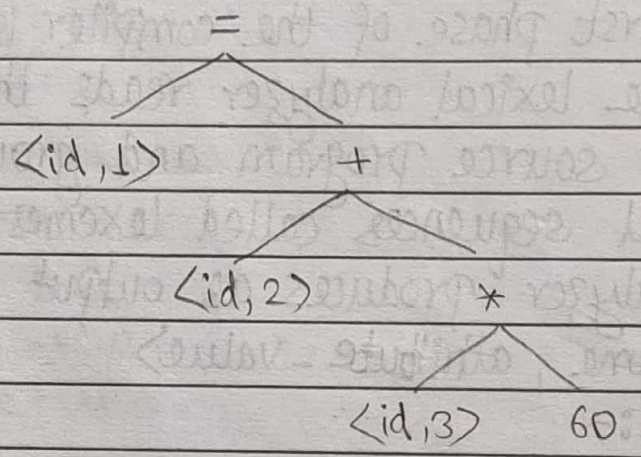
The representation of the assignment statement after lexical analysis as the sequence of tokens

<id,1> <=> <id,2> <+><id,3><*><60>.

Scanned with CamScanner

2) Syntax Analyzer :-

The second phase of the compiler is syntax analysis or parsing. Parser uses tokens produced by lexical analyzer to create a tree like intermediate representation that depicts the grammatical structure of the token stream. A typical representation is a syntax tree in which interior node represents an operation and children of the node represents the argument of the operation. A syntax tree for the token stream of assignment operation is as the output of the syntax analyzer.




The tree shows the order in which the operation in the assignment

position = initial + rate * 60

are to be performed.

The syntax of a language is specified by a context free grammer.

Scanned with CamScanner

3) Semantic Analyzer:-

The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either syntax tree or the symbol table for subsequent use during intermediate code generation.

An important component of semantic analysis is type checking. Here the compiler checks that each operator has operands that are permitted by the source language specification. The language specification may permit some type conversion called coercions. In above figure, coercions appear in converting integer value to float value i.e. inttofloat 60.

4) Intermediate code generation:-

It is fourth phase of the compiler. It gets input from the semantic analyzer and converts the input into the intermediate code such as three-address code. The output of the intermediate code generator in above figure consists of 3-address code sequence as

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

Scanned with CamScanner

5) Code optimization:-

It is the fifth phase of compiler. It gets intermediate code as input and produces optimized intermediate code as output. This phase reduce the redundant code and attempts to improve the intermediate code so that faster running machine code will result. During the code optimization the result of the program is not affected. To improve the code generation, different techniques of optimization are involved. For example:

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
   ↓
t1 = id3 * 60.0
id1 = id2 + t1
```

6) Code Generation:-

It is the final phase of compiler. The code generator takes an intermediate representation of the source program as the input and maps it into the target language. If the target language is the machine code, registers or memory locations are selected for each of the variable used by the program. Then the intermediate instructions are translated into sequences of machine instructions

Scanned with CamScanner

that perform the same task. The code generation involves:

i) allocation of registers and memory
ii) generation of correct reference
iii) generation of correct data types
iv) generation of missing code

Example:

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

## Symbol Table Management:

Symbol Table is an important data structure created and manipulated by compilers inorder to store information about the occurrence of various entities such as variable names, function names, objects classes, etc. Symbol table is used by both the analysis and the synthesis part of the compiler.

A symbol table may serve the following purposes depending upon the language in use.

i) To store the names of all entities in a structured form at one place.
ii) To verify if a variable has been declared.
iii) To implement type checking by verifying assignments and expressions in the source code are semantically correct.

Scanned with CamScanner

iv) To determine the scope of a name.

A symbol table is simply a table which can be either linear or hash table. It maintains an entry for each name in the following format.

<symbol_name, type, ~~form~~ attribute>

For example: if a symbol table has to store information about the following variable declaration.

static int interest;

then it should store the entry such as:

<interest, int, static>

The attribute clause contains the entries related to the name.

## Error Handling / Error Handler:

Each phase can encounters errors in a compiler. After detecting an error, the phase must handle the error so that the compilation can proceed.

In lexical analysis error occurs in seperation of tokens. In syntax analysis error occurs during the construction of syntax tree. In semantic analysis error occurs when the compiler detects constructs with right syntax but no meaning and during type conversion. In code optimization error occurs when the result is affected by the optimization. In code generation error occurs when the code is missing.

Scanned with CamScanner

## Compiler Passes :

A collection of phases is done only once (single pass) or multiple times (multi-pass).

### Single pass :-

One pass compiler is a compiler that passes through the parts of each compilation unit only once, immediately translating each parts into its final machine code. Single pass compiler usually requires everything to be defined before being used in source program.

### Multipass

A multi pass compiler is a type of compiler that processes the source code or abstract syntax tree of a program several times. Multi pass compiler may have to keep entire program representation in memory.

Several phases can be grouped into one single pass and the activities of these phases are interleaved during the pass. For example: lexical analysis, syntax analysis, semantic analysis and intermediate code generation might be grouped into one pass.

## Compiler construction tools :-

The compiler writer, like any other software developer, can profitably use modern software development environment containing tools such as language editors, debuggers, version managers, profilers and so on. These

Scanned with CamScanner

tools use specialized languages for specifying and implementing specific components and many use quite sophisticated algorithms. The most successful tools are those that hides the details of the generation algorithm and produce the components that can be easily integrated into the remaining part of the compiler. These are the specialized tools that have been developed for implementing various phases of a compiler. The following are the compiler construction tools.

i) Parser Generator:

Parser generator automatically produce syntax analyzer from a grammatical descriptions of a programming language. For example: YACC (Yet Another Compiler Compiler).

ii) Scanner Generator:

It produces lexical analyzer from a regular expression description of a token of a language. For example: Lex.

iii) Syntax Directed Translation Engine:

It produce collection of routines for working on parse tree and generate intermediate code.

iv) Code Generator Generator:

Code generator generator produce a code

Scanned with CamScanner

generator from a collection of rules translating each operation of the intermediate language into the machine language for a target machine.

v) Data flow analysis engine:

They facilitate the gathering of information about how the values are transmitted from one part of the program to other part. Dataflow analysis is the key part of code optimization.

vi) Compiler construction toolkits:

They provide an integrated set of routines for constructing various phases of the compiler.

Scanned with CamScanner

हरेक token lexeme हुनुपर्छ tt हरेक lexeme token हुन्न



# Unit 2:

## 2.1) Lexical Analysis

### The Role of Lexical Analyzer :-

The lexical analyzer is the first phase of compiler. Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis. It is common for lexical analyzer to interact with symbol table as well. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.

Since the lexical analyzer is the part of compiler that reads the source text, it may perform certain other task besides identification of lexeme. One such task is stripping out comments and white spaces. Another task is correlating error messages generated by compiler with the source program.

Sometimes lexical analyzer is divided into a cascade of two processes:

a) Scanning :-

It consists of simple processes that do not require tokenization of the input such as deletion of comments and compaction of consecutive white space characters into one.

Scanned with CamScanner

```
void main()
{
  printf("Hello world");
}
```

lexical ⟹

void → keyword
main → identifier
( → special symbol
) → "  "
{ → "  "
printf → identifier
( → special symbol
" → special symbol
Hello → constant (string)
world
" → special symbol
) → "  "
; → "  "
} → "  "



b) Lexical analysis :-

It is the more complex portion, it produces the sequence of tokens as output.

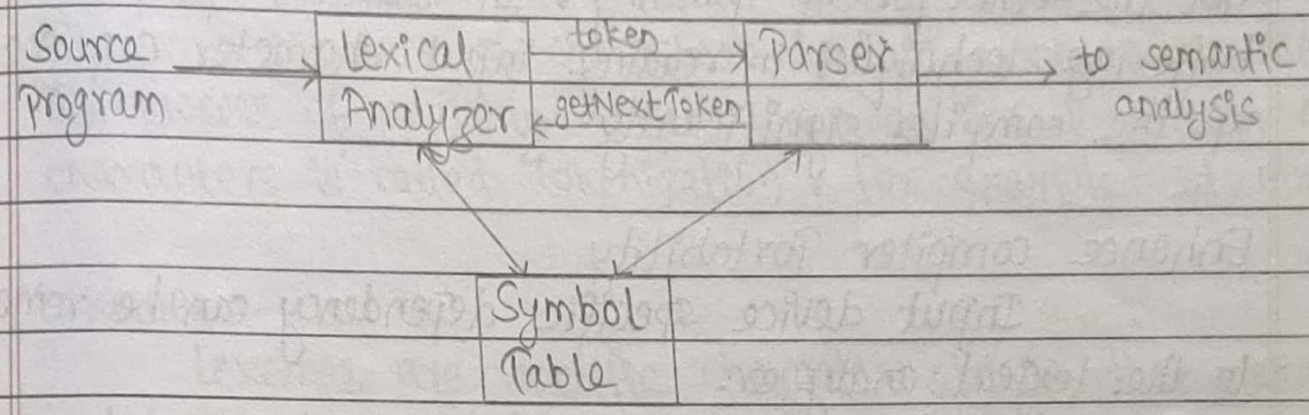



Fig :- Interaction between Lexical analyzer & parser

## Benefits of Seperate Lexical Analyzer :

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

1. Simplicity of Design

The seperation of lexical and syntatic analysis often allows us to simplify task. Such as a parser that had to deal with comments and white spaces as syntactic units would be considered more complex than on that can assume comments and white space have already been removed by lexical analyzer.

Scanned with CamScanner

2. Efficiency of compiler

A seperate lexical analyzer allows us to apply specialized techniques that serve only the lexical task not the ~~other~~ job of parsing. In addition, specialized buffering techniques in reading input character can speed up the compiler significantly.

3. Enhance compiler Portability

Input device specific dependency can be removed to the lexical analyzer.

## Syntax Analyzer Vs. Lexical Analyzer

- Lexical analyzer deals with simple non-recursive constructs of the language.
- The syntax analyzer deals with the recursive construct of the language.
- The lexical analyzer simplifies the job of syntax analyzer.
- The lexical analyzer recognizes the smallest unit (tokens) of the source program.
- The syntax analyzer works on the smallest meaningful units (tokens) in the source program to recognize the meaningful structure in programming language.

Scanned with CamScanner

smallest meaningful unit of programming language — sequence of characters that make up token — set of definitions that define particular lexeme (or token)

# Tokens, Lexemes and Patterns

## Tokens :-

A 'token' is a logical building block of a language. They are sequence of characters having a collective meaning. The process of making tokens from an input stream of characters is called tokenization. For example: id, num, etc.

## Lexemes :-

Lexemes are specific character strings that make up a token. For example; abc, 123, etc. A token can represent more than one lexeme.

## Patterns :-

Patterns are rules describing the set of lexemes belonging to a token. For example: "letter followed by letter and digits" and "non-empty sequence of digits". 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 the attribute of the token.

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

For example:- the token names and associated attribute values for the following statement are :

Scanned with CamScanner

A = B*C + 12

<id, pointer to the symbol table entry for A>
<assign_op>
<id, pointer to the symbol_table entry for B>
<mult_op>
<id, pointer to the symbol_table entry for C>
<add_op>
<num, integer value 12>

## Error Recovery Strategies in Lexical Analysis:-

The following are the error recovery actions in the lexical analysis:

i) Delete one character from remaining input.
ii) Insert a missing character into the remaining input.
iii) Replace one character by another character.
iv) Transforming two adjacent characters.
v) Deletion of successive characters from the token until error is resolved. This technique is called panic mode recovery.

## Lexical Analyzer Implementation Strategies:

There are three general strategies to the implementation of a lexical analyzer.

1) Use a lexical analyzer generator like Flex to produce lexical analyzer from a regular expression based specification. The generator provides routine for reading and buffering the input.

Scanned with CamScanner

ii) Design a lexical analyzer in high level language such as C, using the I/O facility of the language to read and buffering the input.

iii) Write the lexical analyzer in assembly language and explicitly manage the input and buffering.

buffer – to make efficient reading of source program

## Input Buffering:-

Many times a scanner has to look ahead several characters from the current characters inorder to recognize the tokens. For example; int is a 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. Inorder to efficiently move back and forth in the input stream, input buffering is used.

## *. Buffer pair:-

This scheme involves two buffers that are alternatively reloaded as shown in figure below.

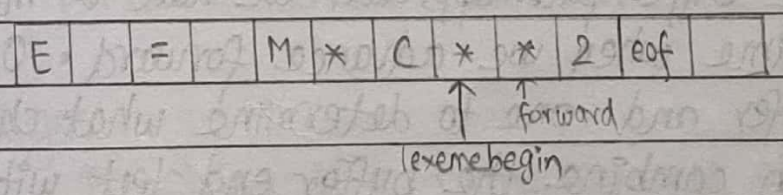



Fig:- Using the pair of input Buffer

Each buffer is of size N (same size) and N is usually the size of disk block (4096 bytes). Using one system read

command we can read N characters into a buffer rather than using one system call per character. If fewer than N character remains in the input file then a special character represented by eof marks the end of the source file and is different from any possible character of the source program.

Two pointers to the input are maintained.

i) Pointer lexeme begin marks the beginning of the current lexeme whose extent we are attempting to determine.

ii) Pointer 'forward' scan ahead until a pattern match is found.

Once the match for the pattern is found, both the pointers point at the same place and the forward pointer is moved again. Advancing 'forward' requires that we first test whether we have reached the end of one of the buffer and if so we must reload the other buffer from the input and move forward to the beginning of the newly reloaded buffer.

*. Sentinels :-

If we use buffer pair approach we have to make two test for each time when we advance forward. One for the end of buffer and one to determine what character is read. We can combine the buffer end test with the test for the current character if we extent each buffer to hold the sentinel character at the end. The sentinel is a special character that cannot be part of the source program and a natural choice is the character eof.

Scanned with CamScanner

Figure below shows the same example as above but with the sentinel added. Note that eof represents both end of the input file and end of the buffer. Any eof that appears other than at the end of a buffer means that the input is at an end

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| E | | = | | M | * | eof | C | * | * | 2 | eof | | eof |

forward

lexeme begin

Fig:- Sentinel at the end of the Buffer

## Specification of Tokens:

### Alphabet:-

An alphabet is a finite, non-empty set of symbols. Conventionally, we use the symbol $\Sigma$ for an alphabet. Common alphabet includes:

1) $\Sigma = \{0,1\}$, the binary alphabet
2) $\Sigma = \{a,b,c,\ldots,z\}$, the set of all lower case letters
3) The set of all ASCII characters or the set of all printable ASCII characters.

### Strings:-

A string is a finite sequence of symbols choosen from some alphabet. For example; 001101 is a string from the binary alphabet $\Sigma = \{0,1\}$.

Scanned with CamScanner

Empty string:-

The empty string is a string with zero occurrance of symbols. It is denoted by $\epsilon$.

Length of string:-

The number of positions in a string is called length of string. The notation for length of string s is $|s|$. For example;

$s = 010110$

$|s| = 6$ $\quad$ $|\epsilon| = 0$.

Power of an alphabet:-

If $\Sigma$ is an alphabet, we can express the set of all strings of a certain length from that alphabet by using an exponential notation. We define $\Sigma^k$ to be the set of strings of length k each of whose symbols is in $\Sigma$. For example;

If $\Sigma = \{a, b, c\}$ then

$\Sigma^0 = \epsilon$

$\Sigma^1 = \{a, b, c\}$

$\Sigma^2 = \{ab, ac, bc, ba, \ldots\}$

The set of all strings over an alphabet is conventionally denoted by $\Sigma^*$. For example;

- Let $\Sigma = \{0, 1\}$ then

$\Sigma^* = \{0,1\}^* = \{\epsilon, 0, 1, 00, 01, \ldots\}$

It can also be written as; $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$

Scanned with CamScanner

Sometimes we wish to exclude empty string from the set of strings. The set of non-empty strings from alphabet $\Sigma$ is denoted by $\Sigma^+$. Thus, two appropriate equivalences are:

1) $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots$
2) $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$

Concatenation of strings:-

Let $x$ and $y$ be strings then $xy$ denotes the concatenation of $x$ and $y$ i.e. the string formed by making a copy of $x$ and following it by a copy of $y$. For example;

Let $x = 0010$ and $y = 1101$, then $xy = 00101101$

For any string $s$, the equation $\epsilon s = s\epsilon = s$ is hold i.e. $\epsilon$ is identity for concatenation.

Language:-

A language is a specific set of strings over some fixed alphabet $\Sigma$. For example;

- $\phi$ is the empty set, is a language
- $\{\epsilon\}$ the set containing empty string is a language
- the set of well formed C programs is a language
- the set of all possible identifiers is a language.

Scanned with CamScanner

## Operations on Languages:-

i) Concatenation : $L_1L_2 = \{s_1s_2 \mid s_1 \in L_1 \ \& \ s_2 \in L_2\}$
ii) Union : $L_1 \cup L_2 = \{s \mid s \in L_1 \text{ or } s \in L_2\}$
iii) Exponentiation : $L^0 = \{\epsilon\}$, $L^1 = L$, $L^2 = LL$ .....
iv) Kleen closure : $L^* = \bigcup_{i=0}^{\infty} L^i$
v) Positive closure : $L^+ = \bigcup_{i=1}^{\infty} L^i$

Example:- Let $L_1 = \{a,b,c,d\}$ & $L_2 = \{1,2\}$ then
$L_1L_2 = \{a1, a2, b1, b2, c1, c2, d1, d2\}$
$L_1 \cup L_2 = \{a,b,c,d,1,2\}$
$L_1^3$ = {all strings with length three (using a,b,c, & d)}
$L_1^*$ = {all strings using letter a,b,c,d and empty string}
$L_1^+$ = {all strings using letter a,b,c,d but not including empty string}.

## Regular Expression:-

Regular expressions are algebric expressions that are used to describe tokens of a programming language. Let $\Sigma$ be an alphabet, the regular expression over the alphabet $\Sigma$ are defined inductively as follows:

Basis step:
- $\phi$ is a regular expression representing the empty language i.e. $L(\phi) = \phi$.
- $\epsilon$ is regular expression representing the language of empty string.
  i.e. $L(\epsilon) = \{\epsilon\}$

Scanned with CamScanner

- If 'a' is a symbol in $\Sigma$ then a is a regular expression representing the language $\{a\}$ i.e. $L(a) = \{a\}$

Induction:

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

- $(r|s)$ is a regular expression denoting $L_1(r) \cup L_2(s)$.
- $(r)(s)$ 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 $L_1(r)$.

A language defined by a regular expression is called a regular set.

## Algebraic Properties of Regular Expression:

The algebraic laws for regular expression r, s, t are as follows:

| | Axioms | Descriptions |
|---|---|---|
| 1. | $r\|s = s\|r$ | • \| is cammutative |
| 2. | $r\|(s\|t) = (r\|s)\|t$ | • \| is associative |
| 3. | $(rs)t = r(st)$ | • concatenation is associative |
| 4. | $r(s\|t) = rs\|rt$<br>$(s\|t)r = sr\|tr$ | • concatenation is distributive over \| |
| 5. | $\epsilon r = r\epsilon = r$ | • $\epsilon$ is identity element for concatenation |
| 6. | $r^* = (r\|\epsilon)^*$ | • relation between * and $\epsilon$ ($\epsilon$ is guaranteed in a closure. |
| 7. | $r^{**} = r^*$ | • * is idempotent. |

Scanned with CamScanner

Examples: (Write string of given RE).

1) $1(0+1)^*0$
- The set of all strings start with 1 and ends with 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 strings that describes alternative 1's and 0's.

4) $0^*10^*10^*10^*$
- Denotes the set of strings that contains exactly three 1's.

5) $11^*(0+\epsilon)11^*(0+\epsilon)11^*$
- Denotes the set of strings the contains at least 3 1's and at most 2 0's.

6) $(A|B|C|....|Z|a|b|c|....|z|_).((A|B|C|....|Z|a|b|c|...|z|_|(0|1|...|9))^*$
- Denotes the regular expression to specify the identifiers of C.

Scanned with CamScanner

$1^*(00)^* 1^*(00)^* 0\ 1^*(00)^*$



$1^*(01^*\ 0\ 1^*)^*\ 01^*$

$1^*\ 01^*\ 01^*\ 01^*\ 1^*$

Example 2: Write Regular expressions for following languages.

1) Strings that either have sub string 001 or 100

- $(0+1)^*\ 001\ (0+1)^* + (0+1)^*\ 100\ (0+1)^*$

2) Strings where no two 1's occurs consecutively

- $00^*\ (01)^*\ 0^* + 10^*\ (10)^*\ 0^*$

3) Strings which have an odd no. of 0's.

-

Scanned with CamScanner

regular expression bati name diyera regular def^ huncha.

## Regular Definitions:

Giving names to regular expression is referred to as a regular definition. Let $\Sigma$ is an alphabet of basic symbols then a regular definition is a sequence of definitions of the form

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\vdots$$
$$d_n \rightarrow r_n$$

where,

each $d_i$ is a distinct names

each $r_i$ is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \ldots d_n\}$

Regular definitions are not recursive

Example:

C identifiers are strings of letters, digits and underscores. Here is a regular definition for the language of C identifier.

letter → A|B|C|....|Z|a|b|c|...|z.

underscore → _

digit → 0|1|2|...|9

id → (letter|underscore) (letter|underscore|digit)*

Example 2:

The following regular definition provide a precise specification for unsigned numbers (int or floating point) such as 5280, 0.01243, 6.336E4, 1.89E-4.

digit → 0|1|2|...|9

Scanned with CamScanner

digits → digit digit*
optionalfraction → . digits | ϵ
optionalexponent → (E (+|-|ϵ) digits) | ϵ
number → digits optional Fraction optional Exponent

Example 3 : Write a regular definition of identifiers of Pascal language.

letter → A|B|C|....|Z|a|b|c|...|z
digit → 0|1|2|....|9
id → (letter) (letter|digit)*

Notational Shorthands:-

*. One or more instance
$r^+ = rr^* = r^*r$ & $r^* = r^+ | \epsilon$

*. zero or one instance
$r? = r | \epsilon$

*. Character Classes
[a-z] = a|b|c|...|z.

Write a regular definition of identifier of pascal using shorthands.

letter → [A-Z, a-z]
digit → [0-9]
id → (letter) (letter|digit)*

Scanned with CamScanner

## Recognition of Tokens:

A recognizer for a language is a program that takes a string w from input stream and answer 'yes' if w is a sentence of that language otherwise 'No'. The tokens that are specified using regular expressions are recognized by using transition diagram or finite automata. In a finite automata, starting from the start state and we follow the transition defined. If the transition leads to the accepting state then the token is matched and the lexeme is returned otherwise, other transition diagram tried out until we process all the transition diagram or the failure is detected. Recognization of tokens takes the language 'L' and the string 's' as input and try to verify whether $s \in L$ or not.

## Finite Automata:

A finite automaton has a set of states and its control moves from state to state in response to external inputs. Recognition of tokens implies implementing a regular expression recognizer that leads the implementation of finite automata. A finite automata can be deterministic (DFA) or non-deterministic (NFA). This means that we may use a deterministic or non-deterministic automaton as a lexical analyzer. Both deterministic and non-deterministic finite automata are capable of recognizing the regular language that regular expression can describe.

Deterministic: faster recognizer but it may take more space.
Non-deterministic: slower but it may take less space.

Scanned with CamScanner

Deterministic automata are widely used in lexical analyzer.

## Design of Lexical analyzer :-

First, we define regular expressions for tokens then we convert them into a DFA to get a lexical analyzer for our tokens. There are two approaches:

**Algorithm 1:**

Regular Expression → NFA → DFA (two steps: first NFA then to DFA).

**Algorithm 2:**

Regular Expression → DFA (directly convert RE into a DFA)

## Non-Deterministic Finite Automata (NFA):

An NFA is a 5-tuple $(S, \Sigma, \delta, S_0, F)$ where,

$S$ is a finite set of states

$\Sigma$ is a finite set of symbols

$\delta$ is a mapping from $S \times \Sigma$ to a set of states

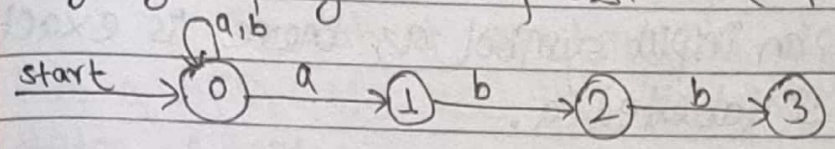
$S_0 \in S$ is the start state.

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

An NFA accepts a string $x$ if and only if there is a path from the starting state to one of accepting state such that edge labels along this path spells out $x$.

Scanned with CamScanner

Example: The transition graph for an NFA recognizing the language of regular expression $(a|b)^*abb$




Here,

$\Sigma = \{a,b\}$

$S = \{0,1,2,3\}$

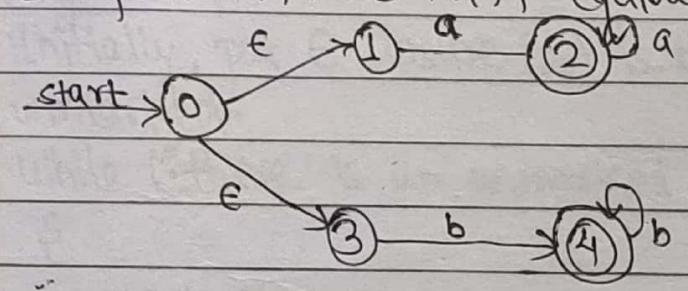
$S_0 = 0$

$F = \{3\}$

## ε-NFA

In NFA if a transition made without any input symbol is called ε-NFA. Regular expressions are easily convertable to ε-NFA.

Example: An ε-NFA equivalent to RE $aa^* + bb^*$




## Deterministic Finite Automata (DFA):

A DFA is a 5-tuple $(S, \Sigma, \delta, S_0, F)$ where

$S$ is a finite set of states

$\Sigma$ is a finite set of symbols

$\delta$ is a transition function that maps state-~~single~~ symbol pair to a state.

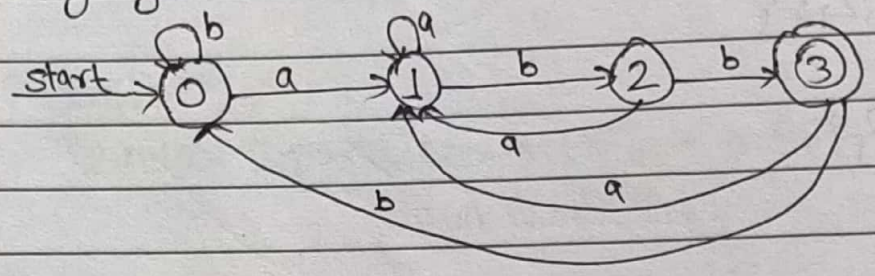
$S_0 \in S$ is the start state

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

Scanned with CamScanner

A DFA is a special case of an ε-NFA where there are no moves on input ε.
For each state s, an input symbol a, there is exactly one edge out of s labelled a.

Example: The transition diagram of a DFA accepting a language (a|b)* abb




## Simulating a DFA:-

Algorithm:

```
S = S0
c = nextchar();
while (c! = eof)
{
    s = move(s,c)
    c = nextchar();
}
if (s is in F)
    return "Yes";
else
    return "No";
```

## Conversion of an NFA to a DFA (subset construction method):

The subset construction algorithm converts an NFA into a DFA. We use the following operations to track of states of NFA.

$\epsilon$-closure(s) :- Set of NFA states reachable from NFA state 's' on $\epsilon$-transition alone.

$\epsilon$-closure(T) :- Set of NFA states reachable from some NFA state s in T on $\epsilon$-transition alone i.e. $\bigcup_{s \in T}$ $\epsilon$-closure(s).

move(T, a) :- Set of NFA states to which there is a transition on input symbol 'a' from some state s in T.

**Algorithm:**

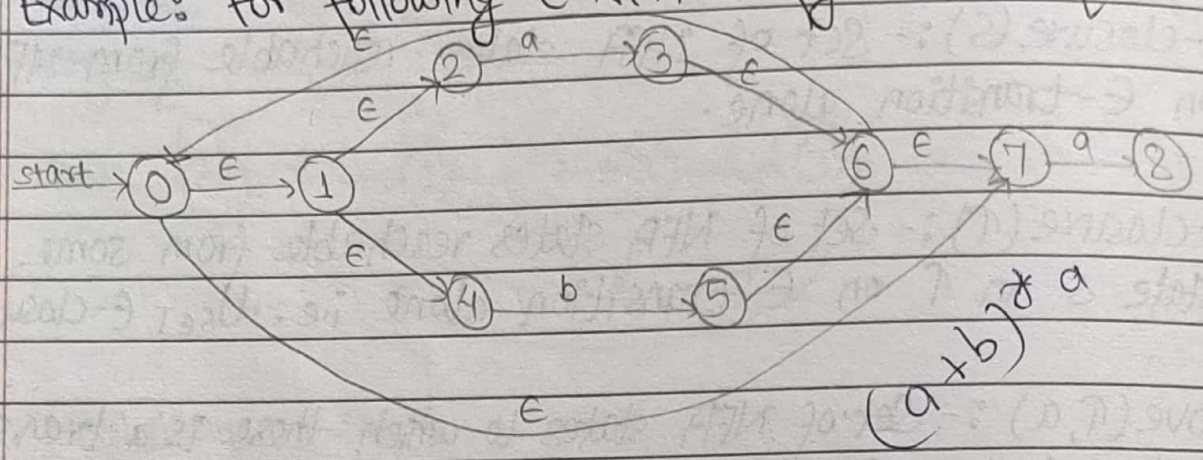
- Initially, put $\epsilon$-closure($s_0$) is the only state in Dstates, & it is unmarked.
- while (there is an unmarked state in Dstates)

```
{
  mark T;
  for(each input symbol a)
  {  U = ε-closure(move(T,a))
     if (U is not in Dstates)
     {
        add U as an unmarked state to Dstates
     }
     Dtran[T,a] = U;
  }
}
```

Scanned with CamScanner

- A state of Dstates is an accepting state of DFA if it is a set of NFA states containing at least one accepting state of NFA.
- The start state of DFA is $\epsilon$-closure$(S_0)$.

Example: For following $\epsilon$-NFA configure the equivalent DFA.




Sol:

$S_0' = \epsilon\text{-closure}(s_0)$ $= \epsilon\text{-closure}(0)$

$= \{0,1,2,4,7\}$

put $S_0$ in the Dstates make it unmarked. Dstate $= \{S_0\}$

$\Downarrow$ mark $S_0$

$\epsilon\text{-closure}(\text{move}(S_0', a)) = \epsilon\text{-closure}(\{3,8\})$

$= \{0,1,2,3,4,6,7,8\} = S_1$

Dstates $= \{S_0, S_1\}$
Dtran $[S_0', a] = S_1$

$\epsilon\text{-Closure}(\text{move}(S_0', b)) = \epsilon\text{-Closure}(\{5\})$

$= \{0,1,2,4,5,6,7\} = S_2$

Dstates $= \{S_0, S_1, S_2\}$
Dtran $[S_0', b] = S_2$

$\Downarrow$ mark $S_1$

$\epsilon\text{-closure}(\text{move}(S_1, a)) = \epsilon\text{-closure}(\{3,8\})$

$= \{0,1,2,3,4,6,7,8\} = S_1$

Dtran $[S_1, a] = S_1$

Scanned with CamScanner

$\epsilon$-closure (move($S_1$, b)) = $\epsilon$-closure ({5}) $\quad$ $D_{tran}[S_1, b] = S_2$

$= \{0,1,2,4,5,6,7\} = S_2$.

$\Downarrow$ mark $S_2$.

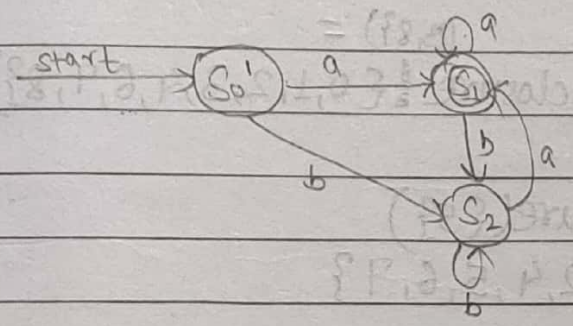
$\epsilon$-closure (move($S_2$, a)) = $\epsilon$-closure ({3,8}) $\quad$ $D_{tran}[S_2, a] = S_1$

$= \{0,1,2,3,4,6,7,8\} = S_1$

$\epsilon$-closure (move($S_2$, b)) = $\epsilon$-closure ({5}) $\quad$ $D_{tran}[S_2, b] = S_2$

$= \{0,1,2,4,5,6,7\} = S_2$

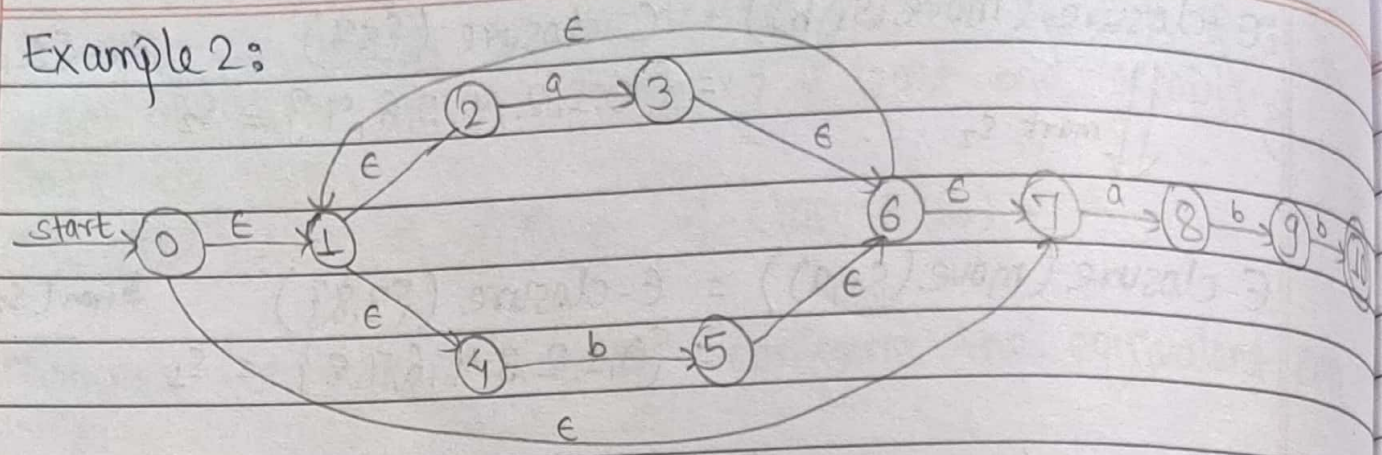
So, the transition diagram of equivalent DFA is :




OR

| States | a | b |
|---|---|---|
| {0,1,2,4,7} | {0,1,2,3,4,6,7,8} | {0,1,2,4,5,6,7} |
| {0,1,2,3,4,6,7,8} | {0,1,2,3,4,6,7,8} | {0,1,2,4,5,6,7} |
| {0,1,2,4,5,6,7} | {0,1,2,3,4,6,7,8} | {0,1,2,4,5,6,7} |

Example 2:




Sol$^n$:

The starting state of NFA is 0, so the starting state of DFA will be:

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

$\Downarrow$ mark $S_0$

$\epsilon\text{-closure}(move(S_0', a)) = \epsilon\text{-closure}(\{3,8\}) = \{0,1,2,3,4,6,7,8\} = S_1$

$\epsilon\text{-closure}(move(S_0', b)) = \epsilon\text{-Closure}(\{5\})$
$= \{0,1,2,4,5,6,7\}$

Scanned with CamScanner

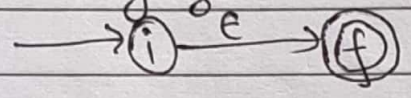
# Construction of an NFA from Regular Expression:

## Thomson's Construction:-

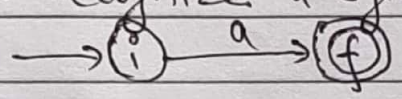
Thomson's construction is a simple and systematic method to convert a regular expression into an NFA. It guarantees that the resulting NFA will have one final state and one start state. Construction starts from simplest part of regular expression. To create an NFA for a complex regular expression, NFA of its sub expressions are combined to create its NFA.

Basis:

1. To recognize an empty string $\epsilon$



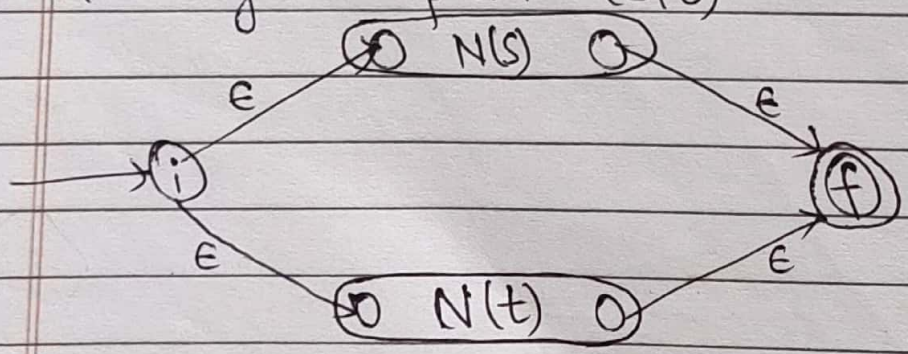

2. To recognize a symbol 'a' in the alphabet $\Sigma$




Induction:

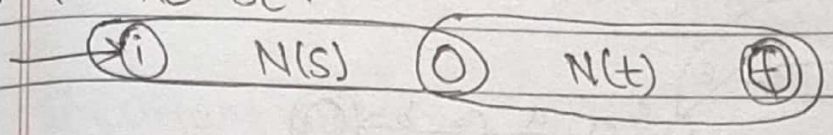
Suppose N(s) and N(t) are NFA's for regular expressions s and t respectively then,

1. For regular expression (s|t)

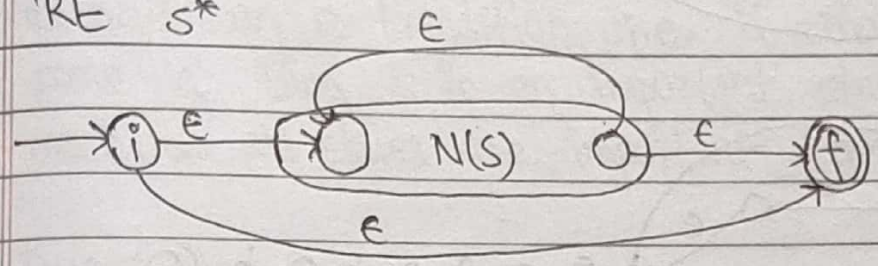



Scanned with CamScanner

2. For RE st.



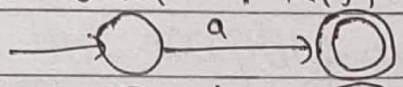

3. RE $s^*$



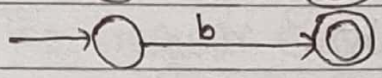

Example:

Construct an NFA for following RE $(a|b)^*$ abb.

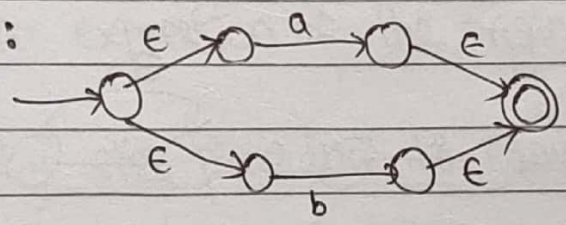
i) a:



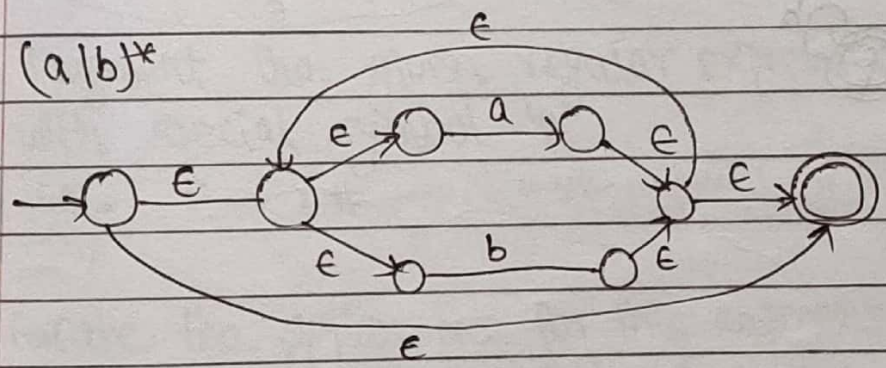

ii) b:



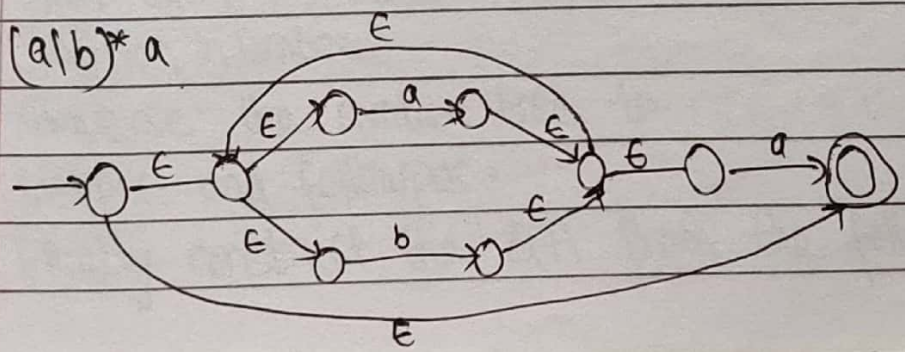

iii) $(a|b)$:




iv) $(a|b)^*$

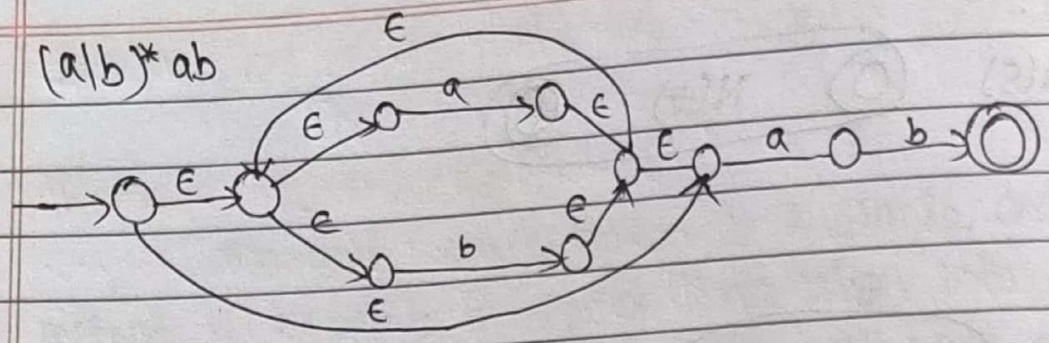



v) $(a|b)^*a$

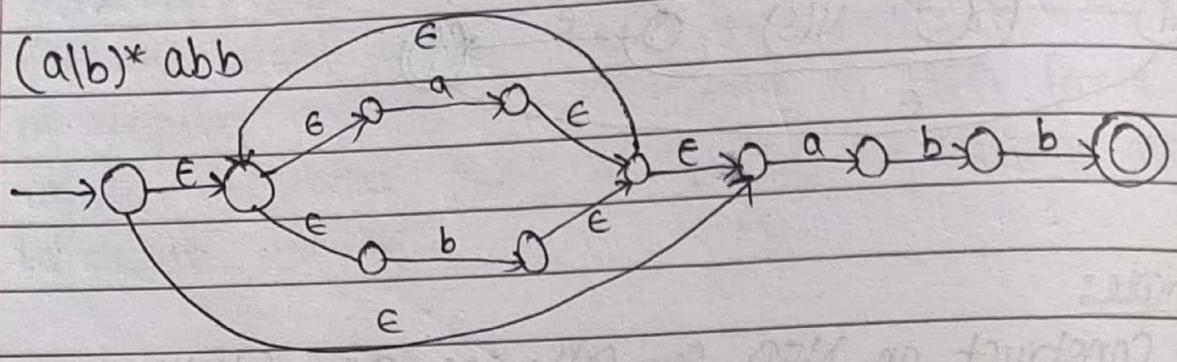



Scanned with CamScanner

vi) $(a|b)^* ab$



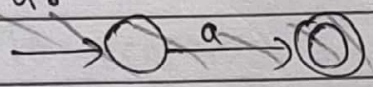

vii) $(a|b)^* abb$

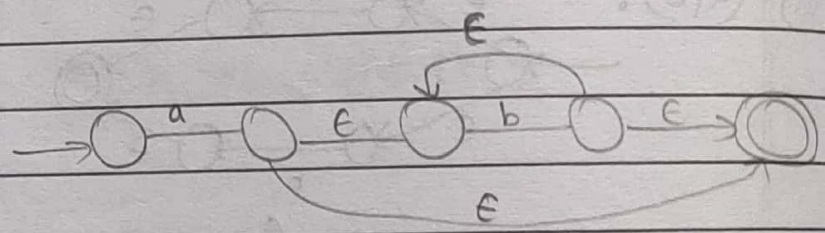



Example 2:

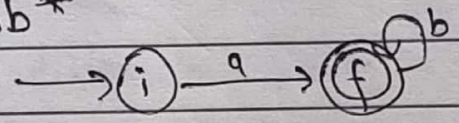
Construct an NFA for RE $ab^*$.

i) a:




ii) $ab^*$



## Conversion of RE to DFA directly:

**Important states:-**

The important states of an NFA are those states without an ε-transition i.e. if move $(\{s\}, a) \neq \phi$ for some 'a' then s is an important state. In an optimal state machine all states are important states.

**Augmented RE:-**

Since the accepting state doesnot have any transition i.e. it is not important state generated from the subset construction method. We introduce an augmented character '#' from the accepting state to make accepting state important state. The regular expression r# is called the augmented regular expression of the original regular expression r.

**Step by step procedure to convert RE to DFA directly:**

1. Augment the given regular expression by concatenating it with special symbol '#'
   i.e. $r \rightarrow r\#$ '#' → final state. to importance rakhna use hunxa.
2. Create the syntax tree for this augmented regular expression.
3. Then each alphabet symbol (including #) will be numbered (position number)
4. Traverse the syntax tree to construct functions nullable, firstpos, lastpos and followpos.
5. Finally construct the DFA from the followpos.

Scanned with CamScanner

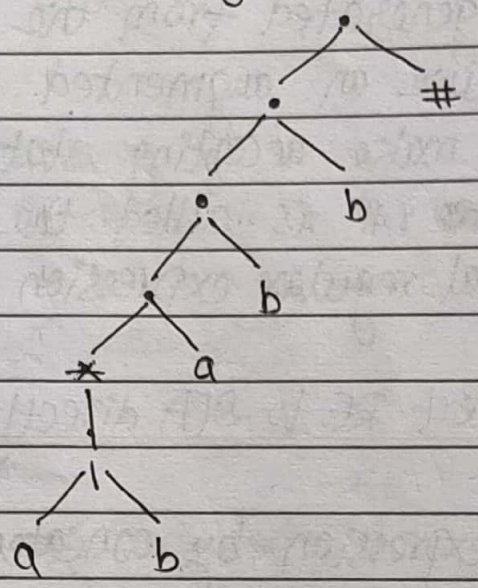
## Syntax tree construction:-

In syntax tree, all alphabets symbols (including # and the empty string ($\epsilon$)) in the augmented regular expression will be in the leaves, and all inner nodes will be the operators in that augmented regular expression.

**For example:**

Let $(a|b)^*abb$ is a regular expression then the augmented regular expression of given regular expression is $(a|b)^*abb\#$

Now, the syntax tree of augmented RE is:




## Computing the functions firstpos, lastpos, nullable & followpos:-

**nullable(n):-**

It is true for a syntax tree of node n if and only if the sub-expression represented by n has $\epsilon$ in its language.

Scanned with CamScanner

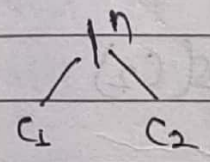
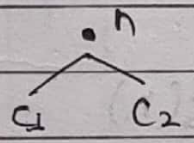
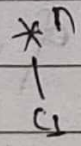
**firstpos(n):-**

It is the set of the positions of the first symbols of strings generated by sub-expression rooted by n.

**lastpos(n):-**

It is the set of the positions of the last symbols of strings generated by sub-expression rooted by n.

Rules for calculating nullable, firstpos & lastpos:-

| | node n | nullable(n) | firstpos(n) | lastpos(n) |
|---|---|---|---|---|
| 1. | A leaf labelled $\epsilon$ | true | $\phi$ | $\phi$ |
| 2. | A leaf with position i | false | $\{i\}$ | $\{i\}$ |
| 3. |  | nullable($c_1$) OR nullable($c_2$) | firstpos($c_1$) $\cup$ firstpos($c_2$) | lastpos($c_1$) $\cup$ lastpos($c_2$). |
| 4. |  | nullable($c_1$) AND nullable($c_2$) | if(nullable($c_1$)) firstpos($c_1$) $\cup$ firstpos($c_2$) else firstpos($c_1$) | if(nullable($c_2$)) lastpos($c_1$) $\cup$ lastpos($c_2$) else lastpos($c_2$) |
| 5. |  | true | firstpos($c_1$) | lastpos($c_1$) |

Scanned with CamScanner

Rules for computing followpos :-

There are only two ways that a position of a regular expression can be made to follow another.

1. If n is a cat-node with left child $c_1$ and right child $c_2$ then for every position i in last pos($c_1$), all positions in firstpos($c_2$) are in followpos(i).

2. If n is a star node, and i is a position in lastpos(n) then all positions in first pos(n) are in followpos(i)

Algorithm to evaluate followpos :-

```
for each node n in the tree do
    if n is a cat-node with left child c1 & right child c2
    then
        for each i in the lastpos (c1) do
            followpos (i) = followpos(i) U firstpos( c2)
        end for
    else if n is a star-node
        for each i in lastpos(n) do
            followpos(i) = followpos(i) U firstpos(n)
        end for
    end if
end for
```

Note: Compute followpos in bottom-up for each node of syntax tree.

Scanned with CamScanner

Example:

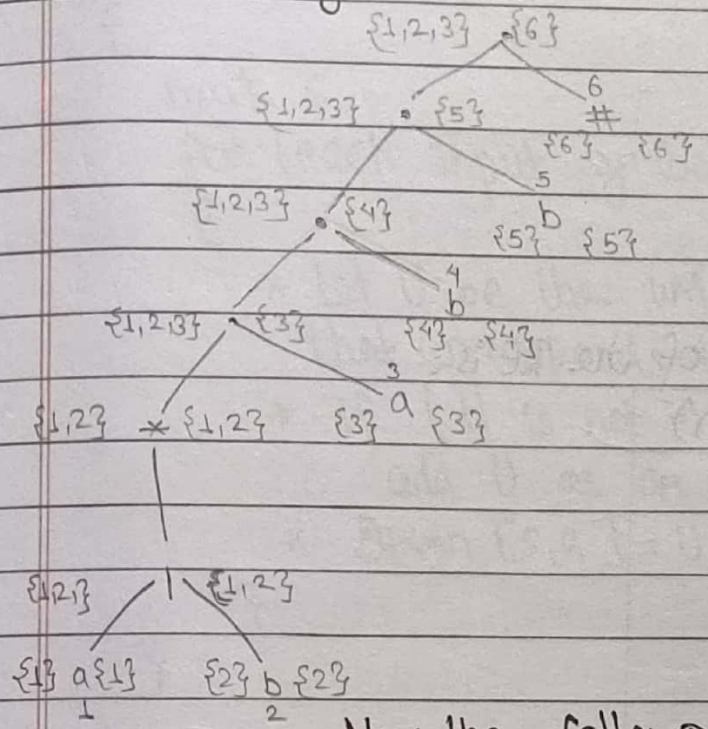
Compute the nullable, firstpos, lastpos & followpos of a regular expression $(a|b)^* abb$.

Sol:

The augmented RE of given RE is:

$(a|b)^* abb\#$

Then the syntax tree is:




Now, the followpos of above RE:

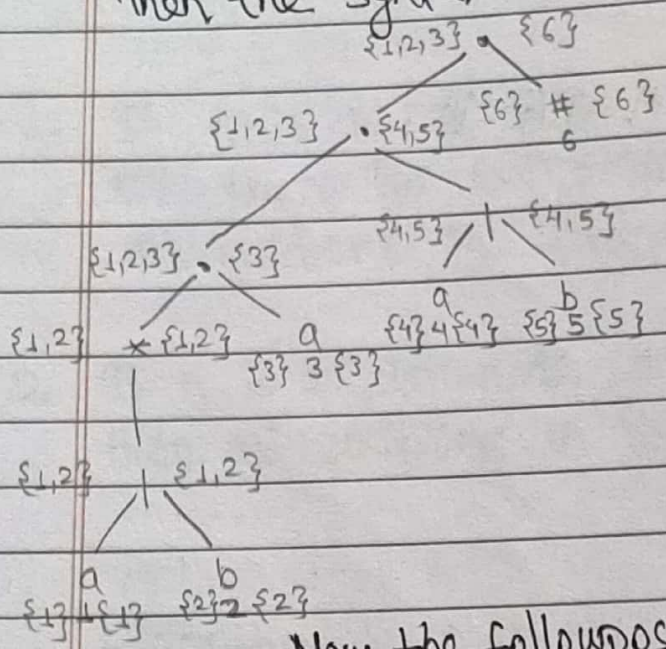
| position n | followpos(n) |
|---|---|
| 1 | {1,2,3} |
| 2 | {1,2,3} |
| 3 | {4} |
| 4 | {5} |
| 5 | {6} |
| 6 | $\phi$ |

Scanned with CamScanner

2) $(a|b)^* a (a|b)$

Sol: The augmented RE of given RE is : $(a|b)^* a (a|b)\#$

Then the syntax tree of augmented RE is:

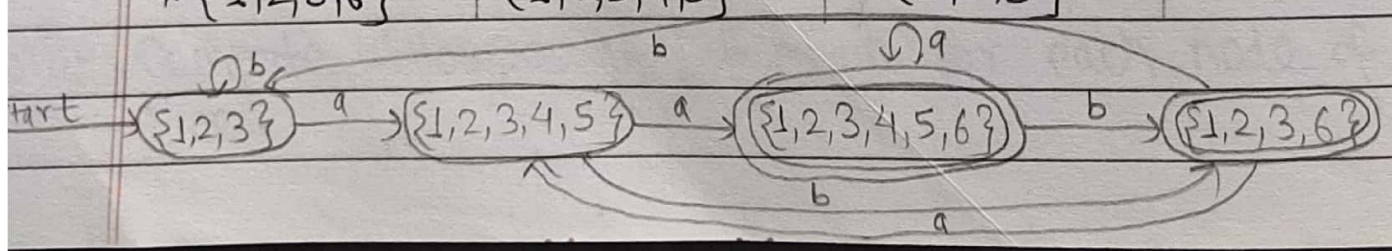



Now, the followpos of the RE is:

| position n | followpos(n) |
|---|---|
| 1 | {1,2,3} |
| 2 | {1,2,3} |
| 3 | {4,5 |
| 4 | {6} |
| 5 | {6} |
| 6 | $\phi$ |

Now, the transition table of the DFA is:

| Dstates | a | b |
|---|---|---|
| → {1,2,3} | {1,2,3,4,5} | {1,2,3} |
| {1,2,3,4,5} | {1,2,3,4,5,6} | {1,2,3,6} |
| * {1,2,3,4,5,6} | {1,2,3,4,5,6} | {1,2,3,6} |
| * {1,2,3,6} | {1,2,3,4,5} | {1,2,3} |




Scanned with CamScanner

## Algorithm to convert a regular expression to DFA directly:

i) Construct a syntax tree from the augmented regular expression r.

ii) Compute nullable, firstpos, lastpos and followpos

iii) Initialize Dstates to contain only the unmarked state firstpos($n_0$), where $n_0$ is the root of the syntax tree.

iv) while (there is an unmarked states s in Dstates)

```
{
    mark s
    for (each input symbol a)
    {
        * let U be the union of followpos(p) for all p in S
          that corresponds to a.
        * if (U is not in Dstates)
            add U as an unmarked state to Dstates
        * Dtran [s,a] = U
    }
}
```

**Note:**

i) The start state of DFA is firstpos (root)

ii) The accepting state of DFA are all states containing the position of #.

Scanned with CamScanner

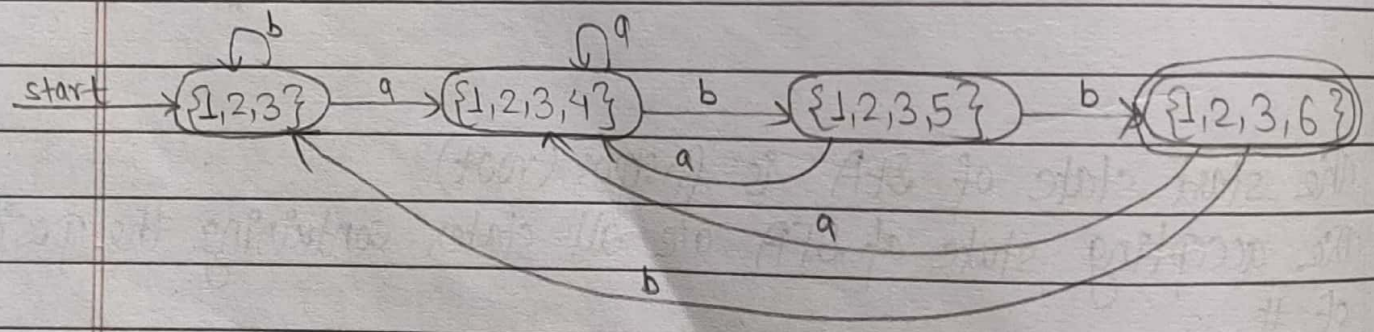
Example :-

The followpos of RE $(a|b)^*abb$ is:

| position n | followpos(n) |
|---|---|
| 1 | {1,2,3} |
| 2 | {1,2,3} |
| 3 | {4} |
| 4 | {5} |
| 5 | {6} |
| 6 | $\phi$ |

Now, the transition table of the DFA is:

| DStates | a | b |
|---|---|---|
| → {1,2,3} | {1,2,3,4} | {1,2,3} |
| {1,2,3,4} | {1,2,3,4} | {1,2,3,5} |
| {1,2,3,5} | {1,2,3,4} | {1,2,3,6} |
| * {1,2,3,6} | {1,2,3,4} | {1,2,3} |




Scanned with CamScanner

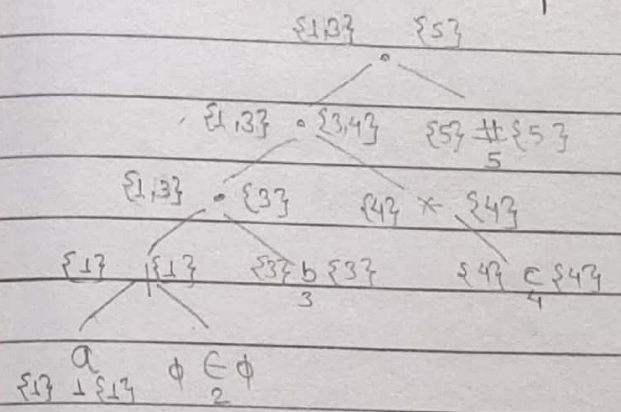
Example:

a) $(a|b)^* \; a(a|b)(a|b)$

b) $(a|\epsilon)\, b\, c^*$

b) ⇒ The augmented RE of given RE is : $(a|\epsilon)\, bc^* \#$

Then the syntax tree of augmented RE is:



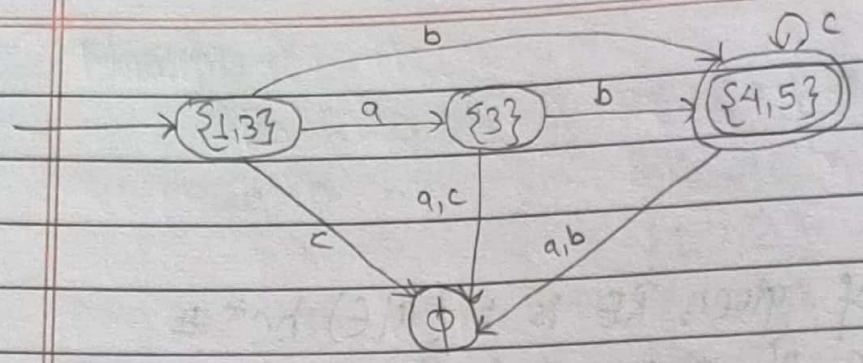

Now, the followpos of the RE is:

| position n | followpos (n) |
|---|---|
| 1 | {3} |
| 2 | ϕ |
| 3 | {4,5} |
| 4 | {4,5} |
| 5 | ϕ |

Now the transition table of the DFA is:

| Dstates | a | b | c |
|---|---|---|---|
| {1,3} | {3} | {4,5} | ϕ |
| {3} | ϕ | {4,5} | ϕ |
| {4,5} | ϕ | ϕ | {4,5} |

Scanned with CamScanner




a)

Scanned with CamScanner

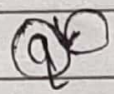
## Minimizing the number of states in DFA :

For every DFA, there is a minimum state DFA accepting the same language. Moreover, the minimum state DFA for a given language is unique except for the names given to the various states. To minimize DFA, following four terms are required:

i) Dead state:

All those non-final state which transit to itself for all input symbols in $\Sigma$ are called dead states.

For example:

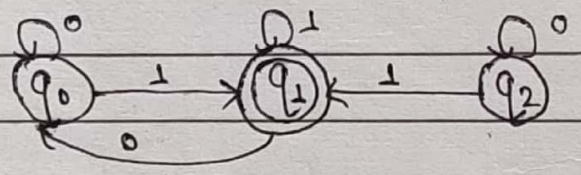



$\Sigma = \{a, b\}$

ii) Inaccessible states:

Those states which cannot be reachable from an initial state are called inaccessible state.

For example:




Here, $q_2$ is an inaccessible state.

iii) Indistinguishable state :

Two states $q$ and $r$ of DFA are said to be indistinguishable if

$\hat{\delta}(q, w) \in F \Rightarrow \hat{\delta}(r, w) \in F$ &

$\hat{\delta}(q, w) \notin F \Rightarrow \hat{\delta}(r, w) \notin F$ for all $w \in \Sigma^*$.

Scanned with CamScanner

iv) Distinguishable state:

Two states q and r of DFA are said to be distinguishable if

$$\hat{\delta}(q,w) \in F \Rightarrow \hat{\delta}(r,w) \notin F \ \&$$
$$\hat{\delta}(q,w) \notin F \Rightarrow \hat{\delta}(r,w) \in F \quad \text{for all } w \in \Sigma^*$$

## Steps in minimizing DFA:

1. Partition the set of states into two groups:
   *. $G_1$ : set of accepting states
   *. $G_2$ : set of non-accepting states.

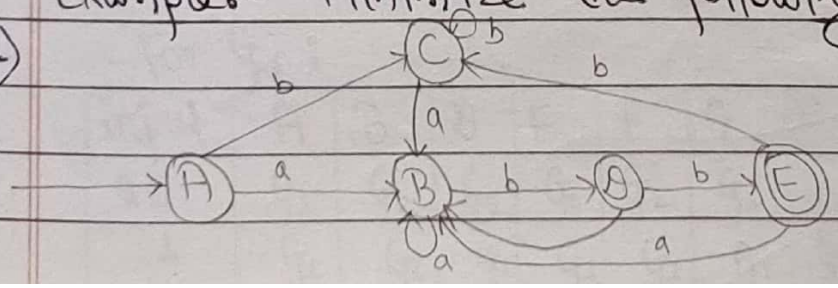
2. For each new group G:
   *. Partition G into sub-groups such that state $S_1$ and $S_2$ are in the same group if and only if for all input symbol a, state $S_1$ and $S_2$ have transition to states in the same group.

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

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

Example:- Minimize the following DFA.

1)



Sol:

Initially construct a group of states of accepting and non-accepting.

$G_1 : \{A, B, C, D\}$
$G_2 : \{E\}$

Now,

For $G_1$:

| input | A | B | C | D |
|---|---|---|---|---|
| a | $G_1$ | $G_1$ | $G_1$ | $G_1$ |
| b | $G_1$ | $G_1$ | $G_1$ | $G_2$ |

Now, new formed groups are:

$G_1 : \{A, B, C\}$
$G_2 : \{E\}$
$G_3 : \{D\}$

Again, for $G_1$:

| input | A | B | C |
|---|---|---|---|
| a | $G_1$ | $G_1$ | $G_1$ |
| b | $G_1$ | $G_3$ | $G_1$ |

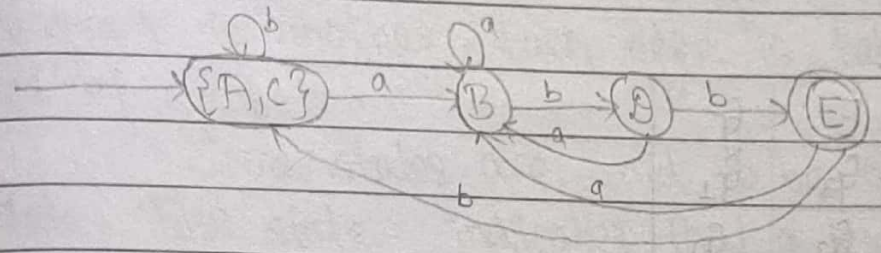
New groups are:

$G_1 : \{A, C\}$
$G_2 : \{E\}$
$G_3 : \{D\}$
$G_4 : \{B\}$

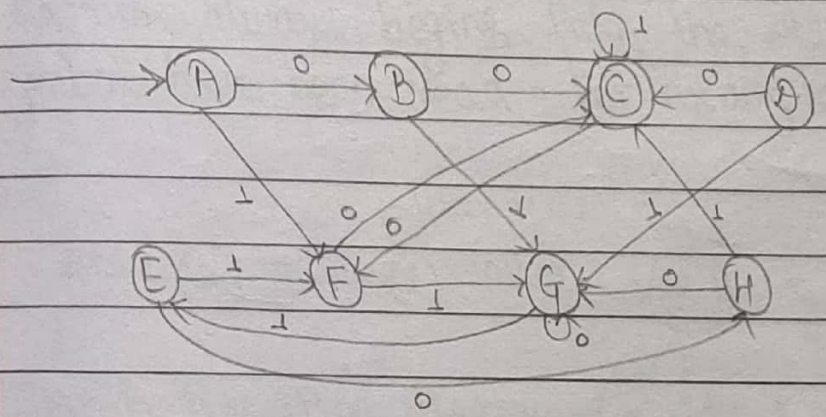
Scanned with CamScanner

Again, for $G_1$:

| input | A | C |
|---|---|---|
| a | $G_4$ | $G_4$ |
| b | $G_1$ | $G_1$ |




Example 2:




Sol:

Initially construct a group of states of accepting and non-accepting.

$G_1 : \{A, B, D, E, F, G, H\}$
$G_2 : \{C\}$

Now,

For $G_1$:

| input | A | B | D | E | F | G | H |
|---|---|---|---|---|---|---|---|
| 0 | $G_1$ | $G_2$ | $G_2$ | $G_1$ | $G_2$ | $G_1$ | $G_1$ |
| 1 | $G_1$ | $G_1$ | $G_1$ | $G_1$ | $G_1$ | $G_1$ | $G_2$ |

Scanned with CamScanner

Now, new formed groups are:

$G_1$ : {A, E, G}
$G_2$ : {B, D, F}
$G_3$ : {H}
$G_4$ : {C}

Again, for $G_1$ :

| input | A | E | G |
|---|---|---|---|
| 0 | $G_2$ | $G_3$ | $G_1$ |
| 1 | $G_2$ | $G_2$ | $G_1$ |

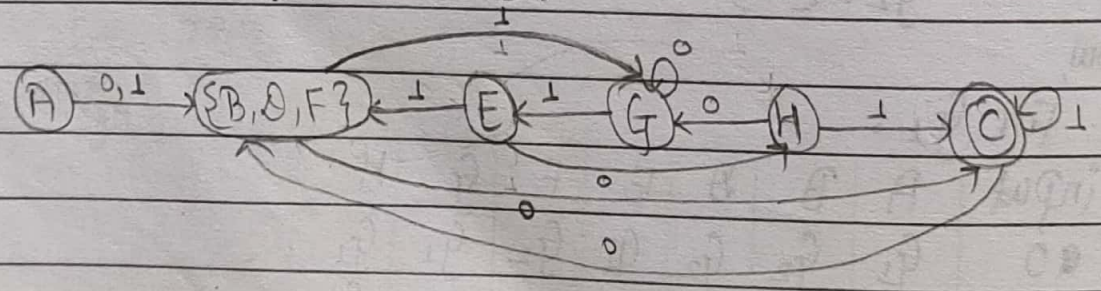
New groups are:

$G_1$ : {A}
$G_2$ : {E}
$G_3$ : {G}
$G_4$ : {B, D, F}
$G_5$ : {H}
$G_6$ : {C}

Now, for $G_4$ :

| input | B | D | F |
|---|---|---|---|
| 0 | $G_6$ | $G_6$ | $G_6$ |
| 1 | $G_3$ | $G_3$ | $G_3$ |

Now, minimized DFA becomes:



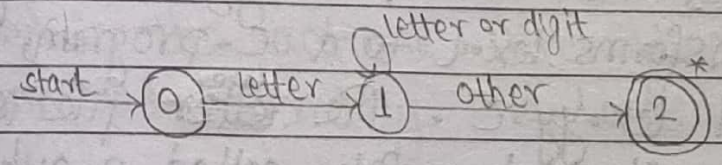
## Transition Diagram :-

Transition diagram have a collection of nodes or circles called states. Each state represents a collection that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns.

Edge are directed from one state of the transition diagram to another. Each edge is labelled by a symbol or set of symbols.

Some states are said to be accepting state or final state. Those state indicates that a lexeme has been found.

One state is designed the start state or initial state. It is indicated by an edge labelled start. The transition diagram always begins from the start state before any input symbol have been read. For example:




[Fig:- A transition diagram for id's & keywords]

## Lex/Flex (Lexical Analyzer Generator) :

Lex/Flex is a tool that allows one to specify a lexical analyzer by specifying regular expressions to describe patterns for tokens.

Use of lex:-

Lex

com

Use of lex:

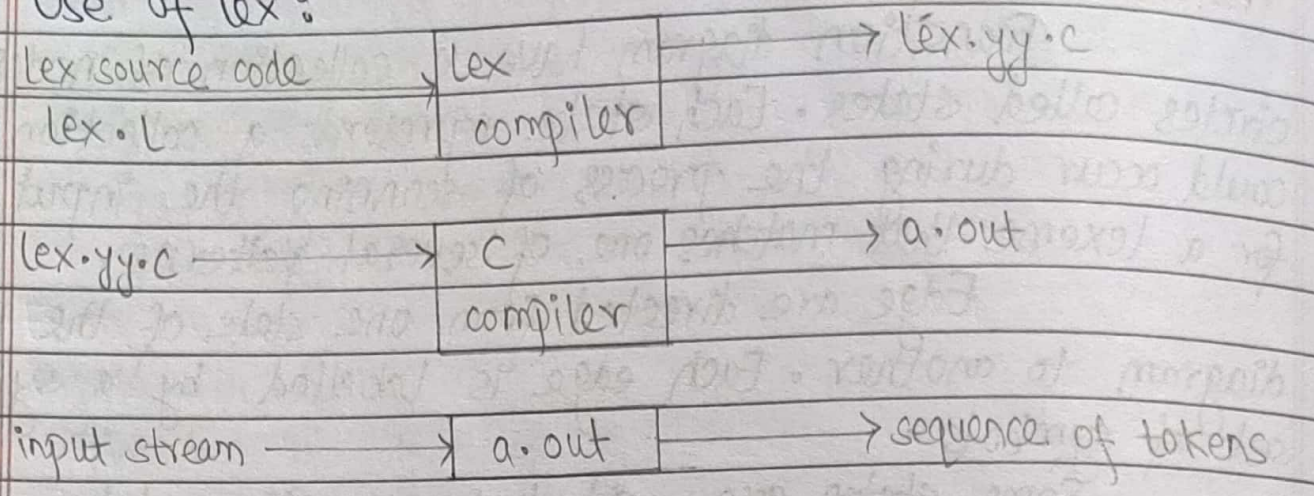



Fig:- Creating a lexical analyzer with lex.

Figure above suggests how lex is used.
An input file which we call lex.l is written in the lex language and describes the lexical analyzer to be generated. The lex compiler transforms lex.l to a C-program, in a file that is always named lex.yy.c. The later file is compiled by the C-compiler into a file called a.out. The C-compiler output is a working lexical analyzer that can take a stream of input characters and produce a stream of tokens.

Structure of lex program:

A lex program has the following form:

declarations
%%
translation rule
%%
auxiliary functions.

1. Declarations:

The declaration session includes the declarations of variables, manifest constant (identifiers declared to stand for a constant. eg: the name of a token) and regular definitions.

2. Translation rule:

The translation rules each have the form:

Pattern {action} :- Each pattern is a regular expression which may use the regular definitions of the declaration section. The actions are the fragment of code typically written in syntax of C.

3. Auxiliary functions:

This section holds whatever additional functions are used in the actions. Alternatively these functions can be compiled seperately and loaded with the lexical analyzer.

Example: Program that returns any integer numbers given by user.

sol:

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

```
main()
{
    yylex();
}
```

} Auxiliary functions

Example 2: Count the number of character and number of lines.

Sol:

```
%{
    #include <stdio.h>
    int num_lines = 0;
    int num_chars = 0;
%}
%%
  \n      {num_lines++ ; num_chars++ ;}
  .       {num_chars++}
%%
main()
{
    yylex();
    printf("Number of character = %d", num_chars);
    printf("Number of lines = %d", num_lines);
}.
```

Scanned with CamScanner

Example 3: Write a lex program to identify whether the given input is number, identifier or other.

Sol:

```
%{
    #include <stdio.h>
%}
  digit    [0-9]
  letter   [A-Z a-z]
   id      (letter) (letter|digit)*
%%
  digit +    {printf("This is number: %s", yytext);}
   id        {printf("This is identifier: %s", yytext;}
    .        {printf("Other : %s", yytext;}
%%
 main()
 {
   yylex();
 }
```

Scanned with CamScanner

## 2.2. Syntax Analysis :

### Syntax Analyzer (Parser) :-

Syntax analysis is the second phase of the compiler. It gets the input from the tokens and generates a syntax tree or parse tree. Syntax analyzer is also called parser.

The syntax of a programming language constructs can be specified by context free grammer.

### Advantages of grammar for syntactic analysis :

- A grammar gives a precise and easy to understand syntatic specification of a programming language.
- An efficient parser can be constructed automatically from a properly designed grammar.
- A grammar imparts a structure to a source program. i.e. useful for its translation into object code and for the detection of errors.
- New constructs can be added to a language more easily when there is a grammatical description of the language.

### The Role of Parser:

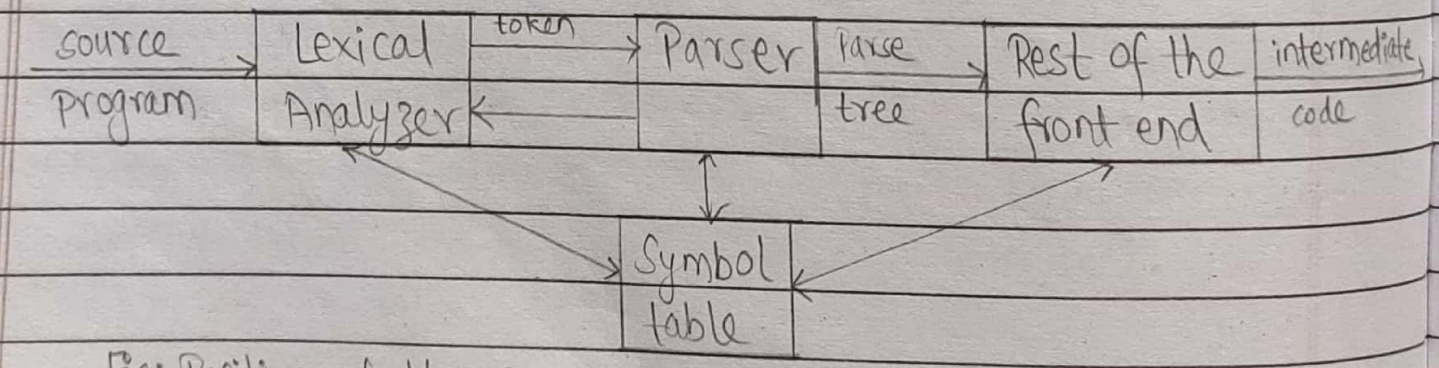



Fig: Position of the parser in compiler model

The parser or syntax analyzer obtains a string of tokens from the lexical analyzer and verifies that token names can be generated by the grammar for the source language. It reports any syntax errors in the program. It also recovers from commonly occuring errors so that it can continue processing its input.

## Syntax Error Handling (Programming errors):-

Programming errors can occur at many different levels such as

1. Lexical error :- It includes mis-spelling of identifiers, keywords or operators.

2. Syntatic errors :- It includes misplaced semi-colons or extra or missing braces.

3. Semantic errors :- It includes type mismatching between operators and operands.

4. Logical errors :- It can be anything from incorrect reasoning such as an infinitely recursive call.

The error handler in a parser has goals that are simple to state but challenging to implement. Functions of error handler are:

i) Report the presence of errors clearly and accurately
ii) Recover from each error quickly enough to detect subsequent

errors.

iii) Add minimal overhead to the processing of correct programs.

## Error Recovery Strategies:-

The different strategies that a parser uses to recover from a syntactic errors are:

i) Panic Mode Recovery
ii) Phrase Level Recovery
iii) Error Production
iv) Global correction

### i) Panic Mode Recovery:-

On discovering an error the parser discards input symbols one at a time until one of a designated set of synchronizing tokens is found. The synchronizing tokens are usually delimiters such as semi colons or closing bracs '}' whose role in the source program is clear enough and unambiguous. It has the advantage of simplicity and does not go into an infinite loop. It is useful when multiple errors in the same statement are rare.

### ii) Phrase Level Recovery:-

On discovering an error, a parser may perform local correction on the remaining input. A typical local correction is to replace a commas by semi-colon, delete an extraneous semi-colon or insert a missing semi-colon.

Scanned with CamScanner

iii) Error Production :-

The parser is constructed using augmented grammar with error productions. If an error production is used by the parser, appropriate error diagnostic can be ~~used~~ generated to indicate the erroneous constructs recognized by the input.

iv) Global correction :-

There are algorithms for choosing a minimal sequence of changes to obtain a globally least cost corrections. Given an incorrect input string $x$ and grammar $G$, these algorithms will find a parse tree for a related string $y$ such that the number of insertions, deletions and changes of tokens required to transform $x$ to $y$ is as small as possible. However, these methods are in general too costly in terms of time and space.

## Context Free Grammar (CFG) :-

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 set of non-terminals
- $P$ is a set of productions of the form $\alpha \rightarrow \beta$ where, $\alpha \in N$ & $\beta \in (N \cup T)^*$
- $S \in N$ is the designated start symbol.

The CFG generates context free language.

Scanned with CamScanner

## Components of Context Free Grammar:

A context free grammar consists of terminals, non-terminals, a start symbol and productions.

- Terminals are the basic symbols from which strings are formed. The token name is a synonym for terminals.
- Non-terminals are syntactic variables that denotes set of strings. The set of strings denoted by non-terminals helps to define the language generated by the grammar. Non-terminals impose a hierarchical structure on the language that is key to syntactic analysis and translation.
- In a grammar, one non-terminal is distinguished as the start symbol and the set of strings it denotes is the language generated by the grammar. Conventionally, the production of the start symbol are listed first.
- The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of:
  * a non-terminal called the head or left side of the production, this production defines some of the strings denoted by the head.
  * the symbol arrow, sometimes (::=) has been used in place of the arrow (→)
  * a body or right side consisting of zero or more terminals and non-terminals. The components of the body describes one way in which strings of the terminals at the head can be constructed.

Scanned with CamScanner

## Notational Conventions :

To avoid always having state that "these are the terminals", "these are the non-terminals" and so on, the following notational conventions for grammar will be used.

1. Terminals are denoted by lowercase letters and symbols (single atoms) and bold strings (tokens).

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

   specific terminal : 0, 1, id, +, etc.

2. Non-terminals are denoted by lowercase italicized letters or uppercase letter symbols.

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

   specific non-terminals : expr, term, stmt, etc.

3. Production rules are of the form $A \rightarrow \alpha$ that is read as A produces $\alpha$.

4. Strings comprising of both terminals and non-terminals are denoted by Greek letters $\alpha, \beta$, etc.

Example :

$S \rightarrow \epsilon$

$S \rightarrow 0S1$

A CFG defining the grammar of language with equal number of zeros followed by equal number of one's.

Here,

i) $T = \{0, 1, \epsilon\}$

ii) $N = \{S\}$

iii) $S = \{S\}$

iv) $P = \{S \rightarrow \epsilon, S \rightarrow 0S1\}$

Scanned with CamScanner

## Derivation :-

Derivation is a process that generates a valid string with the help of grammar by replacing the non-terminals on the left with the string on the right side of the production. Each step of the derivation modifies an intermediary string into a new one by replacing a substring of it that matches with the left hand side of a production rule by the string on right hand side of a rule.

Example:- Consider the following grammar for arithmetic expressions:

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

To generate a valid string -(id+id) from the grammar the steps are:

$$E \Rightarrow -E$$
$$E \Rightarrow -(E)$$
$$E \Rightarrow -(E+E)$$
$$E \Rightarrow -(id+E)$$
$$E \Rightarrow -(id+id)$$

In the above derivation

*. E is the start symbol.
*. -(id+id) is the required sentence (only terminals).
*. strings such as E, -E, -(E), --- are called sentential forms.

Scanned with CamScanner

## Types of derivations:

The two types of derivations are:

1. Left most derivation
2. Right most derivation

1. In leftmost derivations, the leftmost non-terminal in each sentential is always chosen. It can be written as $\alpha \underset{lm}{\Rightarrow} \beta$

Example :- The above derivation is leftmost derivation that can be written as:

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

2. In rightmost derivation, the rightmost non-terminal is always choosen. It can be written as $\alpha \underset{rm}{\Rightarrow} \beta$

Example :- The rightmost form of above derivation is:

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

## Context-Free Languages:-

- The language generated by G, denoted L(G), is

$$L(G) = \{ w \mid w \in T^* \text{ and } S \overset{*}{\Rightarrow} w \}$$

i.e. a string is in L(G), if
- the string consist only terminals
- the string can be derived from S.

- Context-free grammars generates context-free languages.
- Two grammars are equivalent if they produce the same language.
- A string $\alpha \in (N \cup T)^*$ is a sentential form if $S \overset{*}{\Rightarrow} \alpha$.
- A string $\alpha \in T^*$ is a sentence if $S \overset{*}{\Rightarrow} \alpha$.

Scanned with CamScanner

Examples:-

i) $L(G_1)$ = set of all expressions with +, *, names, and balanced '(' and ')'.

ii) $L(G_2) = \{a^n b^n \mid n \geq 0\}$

iii) $L(G_3) = \{x \mid x$ has equal no. of a's and b's$\}$.

## Parse Trees :-

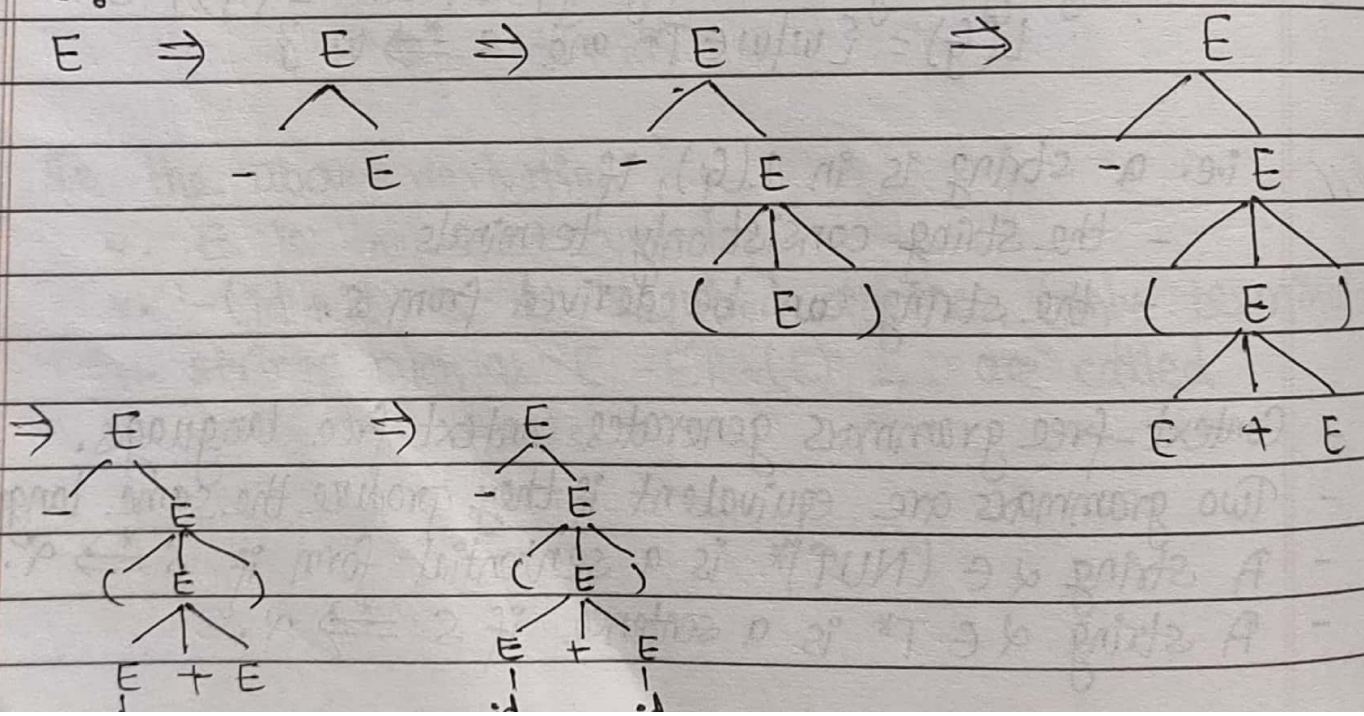
A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace non-terminals. It has following properties:

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

Example :- Let us consider the following CFG.

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

Then the sequence of parse tree for derivation of -(id+id) is :



## Ambiguity :

A grammar that produces more than one parse tree for a sentence is called as an ambiguous grammar. In another word, an ambiguous grammar is one that produces more than one leftmost derivation or more than one right most derivation for the same sentence.

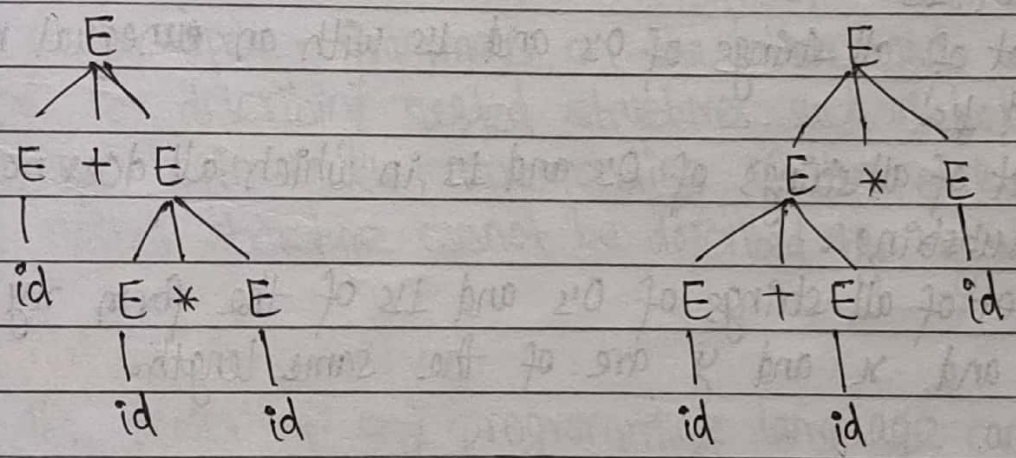
Example :- Let us consider the CFG.

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

Let's check whether the above grammar is ambiguous or not by using string id+id*id.

$E \Rightarrow E+E$
$\Rightarrow id+E$
$\Rightarrow id+E*E$
$\Rightarrow id+id*E$
$\Rightarrow id+id*id$

$E \Rightarrow E*E$
$\Rightarrow E+E*E$
$\Rightarrow id+E*E$
$\Rightarrow id+id*E$
$\Rightarrow id+id*id.$



H.W.

Q.1. Consider the CFG.

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

and the string $aa+a*$

a) Give a leftmost derivation for the string.
b) Give a rightmost derivation for the string.
c) Give a parse tree for the string.
d) Is the grammar is ambiguous or unambiguous? Justify your answer.
e) Describe the language generated by this grammar.

Q.2. Design the grammar for the following languages.

a) The set of all strings of 0's and 1's such that every 0 immediately followed by at least one 1.
b) The set of all strings of 0's and 1's that are palindromes, that is, the string reads the same backward as forwards.
c) The set of all strings of 0's and 1's with an equal number of 0's and 1's.
d) The set of all strings of 0's and 1's with an unequal number of 0's and 1's.
e) The set of all strings of 0's and 1's in which all does not appear as a substring.
f) The set of all strings of 0's and 1's of the form $xy$, where $x \neq y$ and $x$ and $y$ are of the same length.

Q.3. bexpr → bexpr or bterm | bterm
bterm → bterm and bfactor | bfactor
bfactor → not bfactor | (bexpr) | true | false.

Scanned with CamScanner

## Lexical Versus Syntactic Analysis:

We know that everything that can be described by a regular expression can also be described by a grammar. But why regular expressions are used to define the lexical syntax of a language. There are several reasons:

1. Seperating the syntactic structure of a language into lexical and non-lexical parts provides a convenient way of modularizing the front end of a compiler into two manageable sized components.
2. The lexical rules of a language are frequently quite simple to describe them, we do not need a notation as powerful as grammars.
3. Regular expressions generally provide a more concise and easy-to-understand notation for tokens than grammars.
4. More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.

Regular expressions are most useful for describing the structure of constructs such as identifiers, constants, keywords and white space. Grammars, on the other hand, are most useful for describing nested structures such as balanced parenthesis, matching begin-end's, corresponding if-then-else's and so on. Those nested structure cannot be described by regular expressions.

## Eliminating Ambiguity:

The parser of any programming language cannot be generated using ambiguous grammar. Thus the ambiguous grammar should be converted into an unambiguous grammar.

Scanned with CamScanner

precedence - level.
associativity - left



Example:

Let us consider the grammar that describes the mathematical expression.

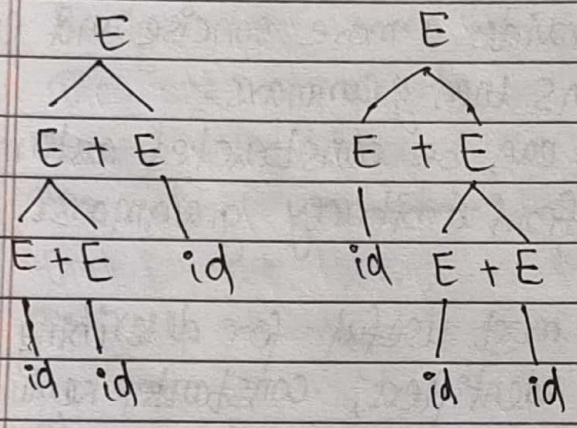
$$E \rightarrow E+E \mid E*E \mid (E) \mid -E \mid id.$$

and strings id+id+id and id+id*id. Check the given grammar is ambiguous or not if it is then convert it into an unambiguous grammar.

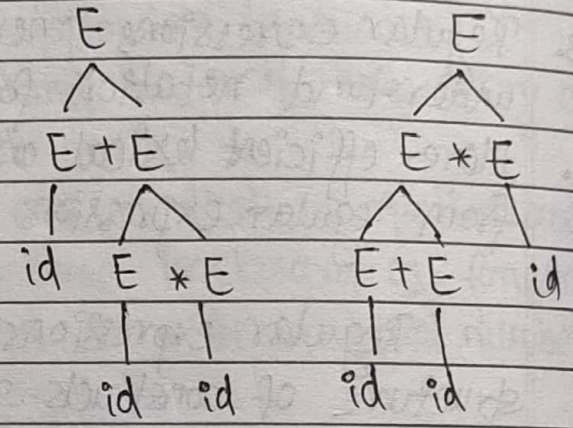
Sol:

To check the given grammar is ambiguous or not, we constructs parse tree of given strings.

id+id+id




id+id*id




The grammar is ambiguous because it is not taking care of both associativity and precedence. Then the unambiguous grammar of above ambiguous grammar is:

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

Note: To define the left associativity of the operator, we use left recursion and to define the precedence of the operator we use the level of the grammar.

Scanned with CamScanner

Example:

$R \rightarrow R+R \mid RR \mid R* \mid a \mid b \mid c$

Scanned with CamScanner

## Eliminating Left Recursion:

- A grammar is left recursive if it has a non-terminal 'A' such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string $\alpha$.
- Top-down parsing method cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion.
- Let us consider left-recursive pair of production.

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

which could be replaced by the non-left-recursive productions:

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

without changing the strings derivable from A.

- In general, left recursion can be eliminated with any number of A-productions is as follows:
- First, group the productions as

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

where no $\beta_i$ begins with an A. Then, replace the A-production by,

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots \mid \alpha_n A' \mid \epsilon$$

- The non-terminal A generates the same strings as before but is no longer left recursive.

Scanned with CamScanner

Example 1:- Consider the following left recursive grammar of the form:

$$E \rightarrow \underbrace{E+T}_{\alpha} \mid \underbrace{T}_{\beta}$$
$$T \rightarrow T*F \mid F$$
$$F \rightarrow (E) \mid id$$

The non-left-recursive form of above grammar is:

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

Example 2:-

$$\underset{A}{S} \rightarrow \underbrace{S0S1S}_{A\ \alpha} \mid \underbrace{01}_{\beta}$$

The non-left-recursive form of grammar is:

$$S \rightarrow 01S'$$
$$S' \rightarrow 0S1SS' \mid \epsilon$$

Example 3:-

$$S \Rightarrow (L) \mid x$$
$$L \rightarrow L, S \mid S$$
$$\Downarrow$$
$$S \rightarrow (L) \mid x$$
$$L \rightarrow SL'$$
$$L' \rightarrow , SL' \mid \epsilon$$

Scanned with CamScanner

Example 4:-

$$A \rightarrow Aa \mid Ab \mid Ac \mid d \mid e \mid f$$

$$\Downarrow$$

$$A \rightarrow dA' \mid eA' \mid fA'$$
$$A' \rightarrow aA' \mid bA' \mid cA' \mid \epsilon$$

Example 5:-

$$S \rightarrow S+S \mid SS \mid (S) \mid S* \mid a$$

Scanned with CamScanner

## Left Factoring:

Left factoring is removing the common left factor that appears in two productions of the same non-terminal. It is done to avoid back-tracking by the parser. It is useful for producing a grammar suitable for predictive or top-down parsing.

If $A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3$ are A-productions, and the input begins with a non-empty string derived from $\alpha$, we do not know whether to expand A to $\alpha\beta_1$, $\alpha\beta_2$ or $\alpha\beta_3$. However, we may differ the decision by expanding A to $\alpha A'$. Then, after seeing the input derived from $\alpha$, we expand $A'$ to $\beta_1, \beta_2$ or $\beta_3$.

i.e. original production : $A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3 | \gamma$

The left factored production : $A \rightarrow \alpha A' | \gamma$

$A' \rightarrow \beta_1 | \beta_2 | \beta_3$

where $\gamma$ represents all alternatives that do not begins with $\alpha$.

Example 1:- Consider the following grammar.

$S \rightarrow iEtS | iEtSeS | a$

$E \rightarrow b$

Left-factor the above grammar.

Sol$^n$:

Given grammar

$S \rightarrow iEtS | iEtSeS | a$

$E \rightarrow b$

Here, $\alpha = iEtS$

Scanned with CamScanner

The left factored grammar for this grammar is:

$S \rightarrow iEtSS' \mid a$

$S \rightarrow eS \mid \epsilon$

$E \rightarrow b$

Example 2:- Consider the following grammar

$S \rightarrow aSSbS$
$\mid aSaSb$
$\mid abb$
$\mid b$

Give the left-factored form of above grammar.

Sol:

Here, the prefix in three productions is a. Therefore, the left-factored for above grammar is:

$S \rightarrow aS' \mid b$

$S' \rightarrow SSbS \mid SaSb \mid bb$

The grammar is still ambiguous (non-deterministic), therefore, we need again to convert it into left-factored grammar.

$S \rightarrow aS' \mid b$

$S' \rightarrow SS'' \mid bb$

$S'' \rightarrow SbS \mid aSb$

Example 3:- Consider the following grammar

$S \rightarrow bSSaaS$
$\mid bSSaSb$
$\mid bSb$
$\mid a$

Left-factor the above grammar.

Scanned with CamScanner

## Parsing :-

- Parsing is the process of determining how a string of terminals can be generated by a grammar.
- A parser builds a parse tree in which the root is labelled with the start symbol, each non-leaf corresponds to a production and each leaf is labelled with a terminal or the empty string $\epsilon$.
- The parse tree derives the string of terminals at the leaves, read from left to right.

## Types of parsing:

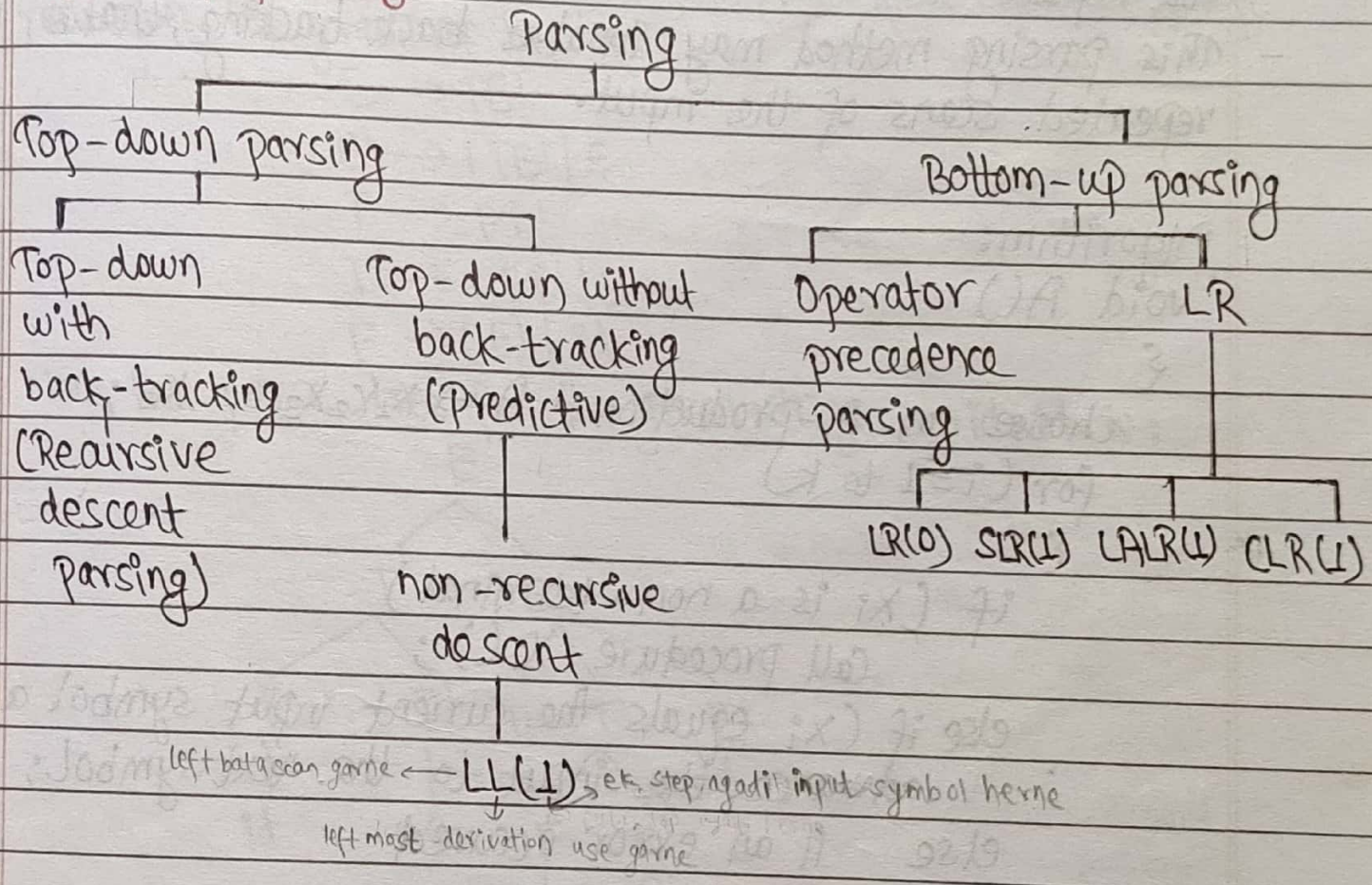



Fig:- Types of parsing.

Scanned with CamScanner

## Top-down parsing:

- Top-down parsing is the process of constructing a parse tree from top to bottom, starting from the root and creating the nodes of the parse tree in preorder.
- Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.

## Recursive Descent Parsing:

- Recursive descent parsing is one of the top-down parsing technique that uses a set of recursive procedure to scan its input.
- This parsing method may involve back-tracking, that is, making repeated scans of the input.

Algorithm:

```
void A()
 {
   choose an A-production, A → X1X2X3 ---Xk ;
   for (i=1 to k)
    {
      if (Xi is a non-terminal)
            Call procedure Xi();
      else if (Xi equals the current input symbol a)
            advance the input to the next symbol;
      else    // an error occured
    }
 }
```

Scanned with CamScanner

$E \rightarrow TE'$
$\Rightarrow FT'E'$
$\Rightarrow FT'+TE'$
$\Rightarrow id+FT'E'$
$\Rightarrow id+id*FT'E'$
$\Rightarrow id+id*id$



Example 1:

Consider the grammar for arithmetic expression.

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

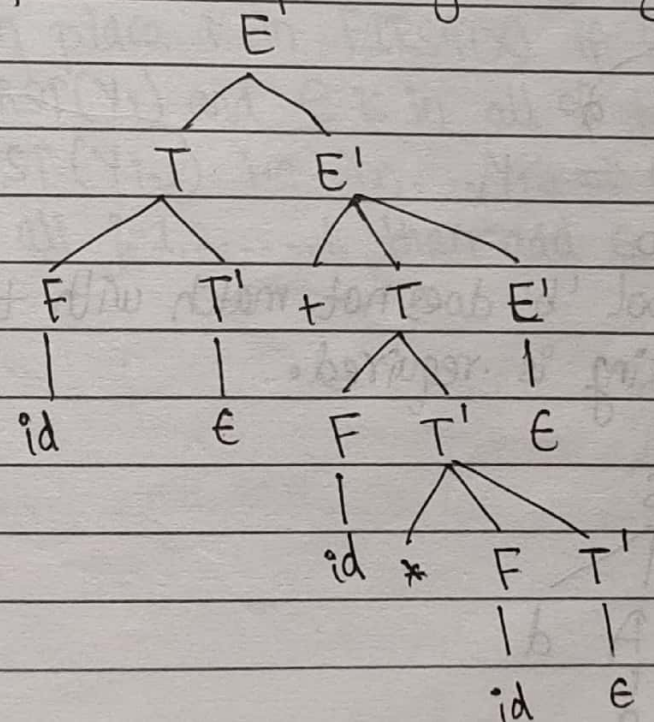
Generate a parse tree according to recursive descent parsing for string id+id*id.

Sol:

A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop. Hence, elimination of left-recursion must be done before parsing. After eliminating left recursion the grammar becomes;

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

Now, the parse tree for the given string id+id*id is:




Scanned with CamScanner

Example 2:- (With backtracking)

Consider the grammar

$$S \rightarrow cAd$$
$$A \rightarrow ab \mid a$$

Construct a parse tree using top-down approach for the input string $w = cad$

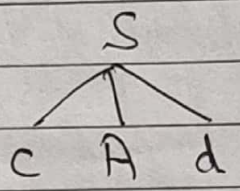
Sol:

Given,

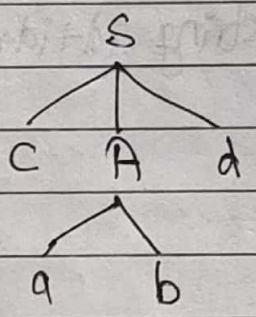
$$S \rightarrow cAd$$
$$A \rightarrow ab \mid a \quad \& \quad w = cad$$

The parse tree can be constructed in following ways:

Step 1:

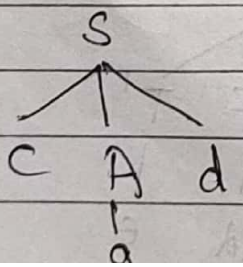



Step 2:




Hence, the symbol 'b' does not match with the input string w.
Hence, backtracking is required.

Step 3:




Scanned with CamScanner

## First and follow:

Construction of both top-down and bottom-up parser is aided by two functions associated with a grammar G.

1. FIRST
2. FOLLOW

## FIRST:

FIRST ($\alpha$) to be the set of terminals that begin strings derived from $\alpha$. If $\alpha \overset{*}{\Rightarrow} \epsilon$, then $\epsilon$ is also in FIRST($\alpha$), where $\alpha$ is any string of grammar symbols.

### Rules for computing FIRST(X)

To compute FIRST(X) for all grammar symbol X, apply the following rules until no more terminals or $\epsilon$ can be added to any FIRST set.

1. If X is terminal, then FIRST(X) is {X}.
2. If $X \rightarrow \epsilon$ is a production, then add $\epsilon$ to FIRST(X).
3. If X is non-terminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production, then place 'a' in FIRST(X) if for some 'i', 'a' is in FIRST($Y_i$) and $\epsilon$ is in all of FIRST($Y_1$), FIRST($Y_2$)..... FIRST($Y_{i-1}$) i.e. $Y_1, \ldots, Y_{i-1} \Rightarrow \epsilon$. If $\epsilon$ is in FIRST($Y_j$) for all $j = 1, 2, \ldots, k$ then add $\epsilon$ to FIRST(X).

Scanned with CamScanner

## FOLLOW(A):

For non-terminal A, FOLLOW(A) to be the set of terminals 'a' that can appear immediately to the right of A in some sentential form.

### Rules for computing FOLLOW(A):

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

1. Place $ in FOLLOW(S), if S is start symbol. $ is the input endmarker.
2. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST($\beta$) except $\epsilon$ is in FOLLOW(B).
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where FIRST($\beta$) contains $\epsilon$, then everything in FOLLOW(A) is in FOLLOW(B).

### Example 1:-

Consider the following grammer.

$S \rightarrow ABCDE$

$A \rightarrow a \mid \epsilon$

$B \rightarrow b \mid \epsilon$

$C \rightarrow c$

$D \rightarrow d \mid \epsilon$

$E \rightarrow e \mid \epsilon$

Find FIRST and FOLLOW of grammar symbols.

Scanned with CamScanner

Sol:

| Grammar | FIRST() | FOLLOW() |
|---|---|---|
| $S \to ABCDE$ | $\{a, b, c\}$ | $\{\$\}$ |
| $A \to a \mid \epsilon$ | $\{a, \epsilon\}$ | $\{b, c\}$ |
| $B \to b \mid \epsilon$ | $\{b, \epsilon\}$ | $\{c\}$ |
| $C \to c$ | $\{c\}$ | $\{d, e, \$\}$ |
| $D \to d \mid \epsilon$ | $\{d, \epsilon\}$ | $\{e, \$\}$ |
| $E \to e \mid \epsilon$ | $\{e, \epsilon\}$ | $\{\$\}$ |

Example 2:-

Consider the following grammar and find out FIRST & FOLLOW of grammar symbols.

$E \to TE'$

$E' \to +TE' \mid \epsilon$

$T \to FT'$

$T' \to *FT' \mid \epsilon$

$F \to id \mid (E)$

Sol:

| Grammar productions | FIRST() | FOLLOW() |
|---|---|---|
| $E \to TE'$ | $\{id, (\}$ | $\{\$, )\}$ |
| $E' \to +TE' \mid \epsilon$ | $\{+, \epsilon\}$ | $\{\$, )\}$ |
| $T \to FT'$ | $\{id, (\}$ | $\{+, \$, )\}$ |
| $T' \to *FT' \mid \epsilon$ | $\{*, \epsilon\}$ | $\{+, \$, )\}$ |
| $F \to id \mid (E)$ | $\{id, (\}$ | $\{*, +, \$, )\}$ |

Scanned with CamScanner

Example 3:-

Consider the following grammar and find the FIRST and FOLLOW of grammar symbols.

$S \rightarrow Bb \mid Cd$

$B \rightarrow aB \mid \epsilon$

$C \rightarrow cC \mid \epsilon$

Sol:

| Grammar Productions | FIRST() | FOLLOW() |
|---|---|---|
| $S \rightarrow Bb \mid Cd$ | $\{a, b, c, d\}$ | $\{\$\}$ |
| $B \rightarrow aB \mid \epsilon$ | $\{a, \epsilon\}$ | $\{b\}$ |
| $C \rightarrow cC \mid \epsilon$ | $\{c, \epsilon\}$ | $\{d\}$ |

Example 4:-

Find FIRST and FOLLOW of given grammar.

$S \rightarrow ACB \mid CbB \mid Ba$

$A \rightarrow da \mid BC$

$B \rightarrow g \mid \epsilon$

$C \rightarrow h \mid \epsilon$

Sol:

| Grammar Productions | FIRST() | FOLLOW() |
|---|---|---|
| $S \rightarrow ACB \mid CbB \mid Ba$ | $\{d, g, h, \epsilon, b, a\}$ | $\{\$\}$ |
| $A \rightarrow da \mid BC$ | $\{d, g, h, \epsilon\}$ | $\{g, \$, h\}$ |
| $B \rightarrow g \mid \epsilon$ | $\{g, \epsilon\}$ | $\{\$, a, h, g\}$ |
| $C \rightarrow h \mid \epsilon$ | $\{h, \epsilon\}$ | $\{g, \$, b, h\}$ |

Scanned with CamScanner

Example 5:-

Find FIRST and FOLLOW of given grammar.

$S \rightarrow aBDh$

$B \rightarrow cC$

$C \rightarrow bC \mid \epsilon$

$D \rightarrow EF$

$E \rightarrow g \mid \epsilon$

$F \rightarrow f \mid \epsilon$

Sol:

| Grammar productions | FIRST() | FOLLOW() |
|---|---|---|
| $S \rightarrow aBDh$ | $\{a\}$ | $\{\$\}$ |
| $B \rightarrow cC$ | $\{c\}$ | $\{g, f, h\}$ |
| $C \rightarrow bC \mid \epsilon$ | $\{b\}$ | $\{g, f, h\}$ |
| $D \rightarrow EF$ | $\{g, f, \epsilon\}$ | $\{h\}$ |
| $E \rightarrow g \mid \epsilon$ | $\{g, \epsilon\}$ | $\{f, h\}$ |
| $F \rightarrow f \mid \epsilon$ | $\{f, \epsilon\}$ | $\{h\}$ |

Scanned with CamScanner

back-tracking n ahune

## Predictive Parsing :-

- Predictive parsing is a special case of recursive descent parsing where no backtracking is required.
- The goal of predictive parsing is to construct a top-down parser that never backtracks.
- The key problem of predictive parsing is to determine the production to be applied for a non-terminal in case of alternatives.

## LL(1) Grammar :-

A grammar in which it is possible to choose the correct production to expand of given non-terminal, looking only the next input symbol is called LL(1) grammar. The first 'L' stands for scanning the input from left to right, the second 'L' stands for producing the leftmost derivation, & the '1' for using one input symbol of look ahead in each step to make parsing action decisions. These grammars allow us to construct a predictive parsing table that gives the correct choice of production.

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

1. For non-terminal $\alpha$ and $\beta$, they do not derive any string beginning with a.
2. At most one of $\alpha$ and $\beta$ can derive the empty string.
3. If $\beta \overset{*}{\Rightarrow} \epsilon$, then $\alpha$ does not derive any string beginning with a terminal in FOLLOW(A). Likewise, if $\alpha \overset{*}{\Rightarrow} \epsilon$, then $\beta$ does not derive any string beginning with a terminal in FOLLOW(A).

Scanned with CamScanner

## Construction of predictive parsing table (Algorithm):

Input: Grammar G
Output: Parsing table M

Method:

1. For each production $A \to \alpha$ of the grammar, do steps 2 & 3.
2. For each terminal a in FIRST($\alpha$), add $A \to \alpha$ to M[A, a].
3. If $\epsilon$ is in FIRST($\alpha$), add $A \to \alpha$ to M[A, b] for each terminal b in FOLLOW(A). If $\epsilon$ is in FIRST($\alpha$) and \$ is in FOLLOW(A) add $A \to \alpha$ to M[A, \$].
4. Make each undefined entry of M be error.

Example:- Construct a predictive parsing table for the following grammar.

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

Sol:

| | |
|---|---|
| FIRST(E) = {id, (} | FOLLOW(E) = {\$, )} |
| FIRST(E') = {+, $\epsilon$} | FOLLOW(E') = {\$, )} |
| FIRST(T) = {id, (} | FOLLOW(T) = {+, \$, )} |
| FIRST(T') = {*, $\epsilon$} | FOLLOW(T') = {+, \$, )} |
| FIRST(F) = {id, (} | FOLLOW(F) = {*, +, \$, )} |

Scanned with CamScanner

Now, the predictive parsing table of above grammar is:

| non-terminal \ input symbol | id | + | * | ( | ) | $ |
|---|---|---|---|---|---|---|
| E | $E \to TE'$ | | | $E \to TE'$ | | |
| E' | | $E' \to +TE'$ | | | $E' \to \epsilon$ | $E' \to \epsilon$ |
| T | $T \to FT'$ | | | $T \to FT'$ | | |
| T' | | $T' \to \epsilon$ | $T' \to *FT'$ | | $T' \to \epsilon$ | $T' \to \epsilon$ |
| F | $F \to id$ | | | $F \to (E)$ | | |

Fig:- M table of above grammar

## How to check grammar is LL(1) or not?

A LL(1) grammar is a grammar which is not
- ambiguous
- left recursive
- non-deterministic

If a grammar have any of the above problem simply it can't be LL(1). To check whether the given grammar is LL(1) or not we use the following method.

**Method 1:**

Draw predictive parsing table if we found multiple entries in a table, we can say given grammar is not LL(1).

**Method 2:**

i) Productions which have only single derivation is always parse by LL(1) parser so we don't need to see these derivation. We simply ignore them while checking grammar is LL(1) or not.

Scanned with CamScanner

ii) Production with multiple derivation but no $\epsilon$ move such as

$$S \rightarrow A|B|C$$

Then calculate FIRST(A), FIRST(B) & FIRST(C), if they are pair-wise disjoint then the given grammar is LL(1).

iii) Production with multiple derivation with $\epsilon$-move such as

$$S \rightarrow A|B|\epsilon$$

Then calculate FIRST(A), FIRST(B) & FOLLOW(S), if they are pair wise disjoint, then the given grammar is LL(1).

Example 1:

Show that the following grammar is not LL(1) grammar.

$S \rightarrow cAd$

$A \rightarrow Ab|a$

[TU 2068]

Sol:

Given grammar is:

$S \rightarrow cAd$

$A \rightarrow Ab|a$

Now, the first and follow of the grammar symbol is:

| Grammar productions | FIRST() | FOLLOW() |
|---|---|---|
| $S \rightarrow cAd$ | {c} | {$} |
| $A \rightarrow Ab\|a$ | {a} | {d, b} |

Then, the parsing table of above grammar is:

| non-terminal \ terminal | a | b | c | d |
|---|---|---|---|---|
| S | | | $S \rightarrow cAd$ | |
| A | $A \rightarrow Ab$<br>$A \rightarrow a$ | | | |

Since, the entry for M[A, a] contains both $A \rightarrow Ab$ and $A \rightarrow a$, thus the grammar is not LL(1) grammar.

Scanned with CamScanner

Example 2:

Consider the following grammar.

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \epsilon$

$E \rightarrow b$

Check whether the given grammar is LL(1) or not?

Sol:

To construct a parsing table, we need FIRST() and FOLLOW() for all the non-terminals.

FIRST(S) = {i, a}  FOLLOW(S) = {$, e}

FIRST(S') = {e, ε}  FOLLOW(S') = {$, e}

FIRST(E) = {b}  FOLLOW(E) = {t}

Parsing table:

| non-terminal \ terminal | a | b | e | i | t | $ |
|---|---|---|---|---|---|---|
| S | $S \rightarrow a$ | | | $S \rightarrow iEtSS'$ | | |
| S' | | | $S' \rightarrow eS$<br>$S' \rightarrow \epsilon$ | | | $S' \rightarrow \epsilon$ |
| E | | $E \rightarrow b$ | | | | |

Since, there is more than one production in a cell, thus the given grammar is not LL(1) grammar.

Scanned with CamScanner

implicitly stack ফাঁকা রাখার জন্য বলে recursive predictive parsing



hamile. explicitly stack বানায়ে বলে non-recursive predictive parsing.

## Non-recursive Predictive Parsing:

A non-recursive predictive parser can be built by maintaining a stack explicitly rather than implicitly via recursive calls. It is also called table-driven parser. Figure below shows the components of table driven parser.

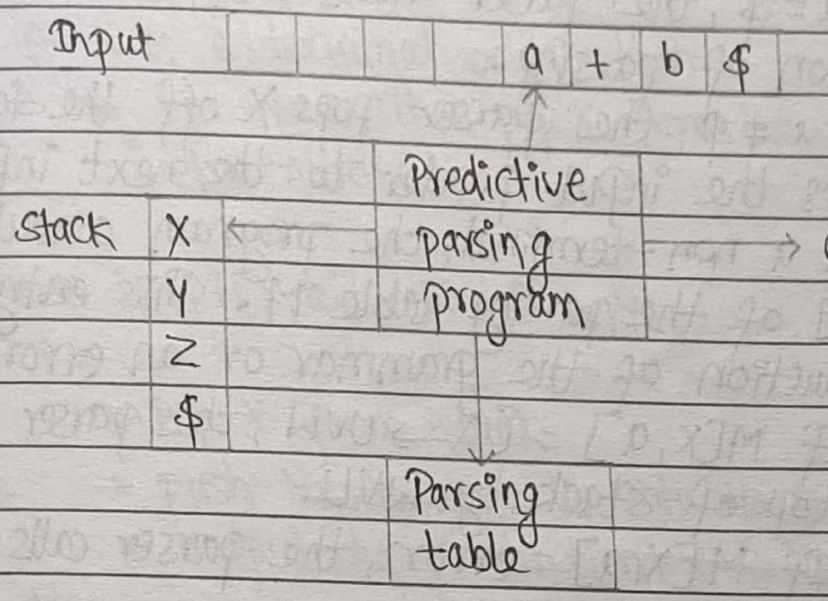



Fig:- Model of a table-driven predictive parser.

i) Input buffer :- The input buffer contains the string to be parsed, followed by the end marker $.

ii) Stack :- It contains a sequence of grammar symbols preceeded by $ to indicate the bottom of the stack. Initially, the stack contains the start symbol on top of $.

iii) Parsing table :- It is a two dimensional array M[A,a], where A is a non-terminal and 'a' is a terminal.

Scanned with CamScanner

iv) Predictive Parsing program:- The parser is controlled by a program that considers X, the symbol on top of stack, and a, the current input symbol. These two symbols determines the parser action. There are three possibilities:

i) If $X = a = \$$, the parser halts and announces successful completion of parsing.
ii) If $X = a \neq \$$, the parser pops X off the stack and advances the input pointer to the next input symbol.
iii) If X is a non-terminal, the program consults entry M[A, a] of the parsing table M. This entry will be an X-production of the grammar or an error entry.
   a) If $M[X, a] = \{X \rightarrow UVW$, the parser replace X on top of stack by WVU.
   b) If M[X, a] = error, the parser calls an error recovery routine.

## Algorithm for predictive parsing (table-driven parsing):

Input: A string w and a parsing table M for grammar G.
Output: If w is in L(G), a leftmost derivation of w; otherwise, an error.

Method: Initially, the parser has \$S on the stack with S, the start symbol of G on top, and w\$ in input buffer. The program that utilizes the predictive parsing table M to produce a parse for the input is as follows:

Scanned with CamScanner

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
   i) Let X be the top stack symbol and 'a' be the symbol pointed by ip.
   ii) If X is a terminal or $ then
   {
      * if X=a, then pop X from the stack and advance ip.
        else error();
   }
   else if M[X,a] = X → $Y_1, Y_2, \ldots, Y_k$ then
   {
      * pop X from stack
      * push $Y_k, Y_{k-1}, \ldots Y_1$ onto stack (with $Y_1$ on top)
      * output the production $X \to Y_1, Y_2, \ldots, Y_k$
   else
      error();
4. until X = $

Example: Consider the following grammar and trace out predictive parsing algorithm for w = abba.

$S \to aBa$

$B \to bB \mid \epsilon$

Sol:

input :- abba$

Parsing table:

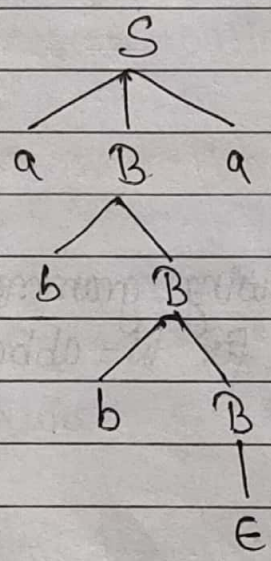
| | a | b | $ |
|---|---|---|---|
| S | $S \to aBa$ | | |
| B | $B \to \epsilon$ | $B \to bB$ | |

Scanned with CamScanner

| Matched | stack | input | Output |
|---|---|---|---|
| | $S | abba $ | S→aBa |
| a | $aBa | abba $ | |
| a | $aB | bba $ | B→bB |
| ab | $aBb | bba $ | |
| ab | $aB | ba $ | B→bB |
| abb | $aBb | ba $ | |
| abb | $aB | a $ | B→ϵ |
| abb | $a | a $ | |
| abba | $ | $ | accept, successful completion |

Outputs: S→aBa , B→bB , B→bB , B→ϵ

Derivation (leftmost): S ⇒ aBa ⇒ abBa ⇒ abbBa ⇒ abba




Example 2: Trace out the table-driven parsing algorithm for input w = id+id+id for the given grammar.

E → TE'  T → FT'  F → id | (E)

E' → +TE' | ϵ  T' → *FT' | ϵ

## Bottom-up parsing:

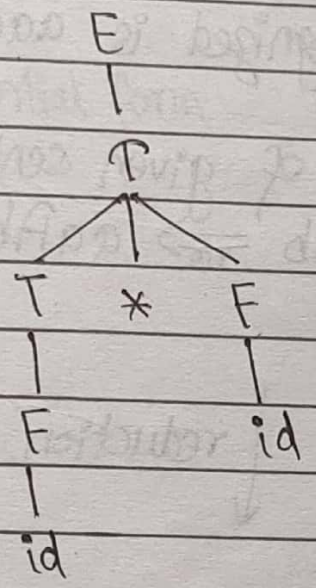
Bottom-up parsing attempts to construct a parse tree for an input string starting from leaves (the bottom) and working up towards the root (the top). A general type of bottom-up parser is a shift-reduce parser.

Example:-

Let us consider the mathematical expression grammar

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

Then, the bottom-up parse of the token stream id*id is as follows:




[Fig:- A bottom-up parse for id*id].

## Reduction:-

The process of replacing a substring by a non-terminal in bottom-up parsing is called reduction. It is reverse process of production. Example:- $S \rightarrow aA$

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

Scanned with CamScanner

## Shift-Reduce parsing :-

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

a string $\xrightarrow[\text{reduce to}]{}$ the starting symbol

The reduction trace out the rightmost derivation in reverse.

Example: Consider the grammar.

$S \rightarrow aABb$

$a \rightarrow aA \mid a$

$B \rightarrow bB \mid b$

The sentence to be recognized is aaabb.

The rightmost derivation of given sentence is:

$S \underset{rm}{\Rightarrow} aABb \underset{rm}{\Rightarrow} aAbb \underset{rm}{\Rightarrow} aaAbb \underset{rm}{\Rightarrow} aaabb$

input string : aaabb

aaAbb

aAbb

aABb

S

reduction

## Handles :- (Handle Pruning)

- A handle of a string is a substring that matches the right side of a production, and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a rightmost derivation.

Scanned with CamScanner

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

Example:-

Consider the grammar.

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

and input string $id+id*id$

The rightmost derivation of $id+id*id$

$E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*id \Rightarrow E+F*id$

$\Rightarrow E+id*id \Rightarrow T+id*id \Rightarrow F+id*id \Rightarrow id+id*id.$

| Rightmost sentential form | Reduction Production | Handle |
|---|---|---|
| $\underline{id}+id*id$ | $F \rightarrow id$ | id |
| $\underline{E}+id*id$ | $T \rightarrow F$ | F |
| $\underline{T}+id*id$ | $E \rightarrow T$ | T |
| $E+\underline{id}*id$ | $F \rightarrow id$ | id |
| $E+\underline{E}*id$ | $T \rightarrow F$ | F |
| $E+T*\underline{id}$ | $F \rightarrow id$ | id |
| $E+\underline{T*F}$ | $T \rightarrow T*F$ | T*F |
| $\underline{E+T}$ | $E \rightarrow E+T$ | E+T |
| $E$ | | |

Scanned with CamScanner

## Stack implementation of Shift-Reduce Parser:

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

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

## Actions in Shift-reduce parser:

There are 4 actions in shift-reduce parser.

i) Shift:- Shift the next input symbol onto the top of stack.
ii) Reduce :- Replaces the handle on the top of the stack by the non-terminal.
iii) Accept:- Successful completion of parsing
iv) Error:- Parser discovers a syntax error and calls an error recovery routine.

Scanned with CamScanner

Example: Consider the expression grammar.

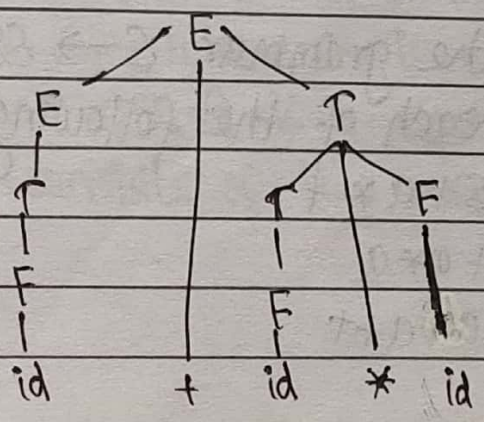
$$E \to E+T \mid T$$
$$T \to T*F \mid F$$
$$F \to (E) \mid id$$

and input string id+id*id. Trace out the shift-reduce parser.

Sol:

| Stack | Input | Action |
|---|---|---|
| $ | id+id*id$ | shift |
| $id | +id*id$ | reduced by F→id |
| $F | +id*id$ | reduced by T→F |
| $T | +id*id$ | reduced by E→T |
| $E | +id*id$ | shift |
| $E+ | id*id$ | shift |
| $E+id | *id$ | reduced by F→id |
| $E+F | *id$ | reduced by T→F |
| $E+T | *id$ | shift (or reduce?) |
| $E+T* | id$ | shift |
| $E+T*id | $ | reduced by F→id |
| $E+T*F | $ | reduced by T→T*F |
| $E+T | $ | reduced by E→E+T |
| $E | $ | accept. |

Parse tree:




Scanned with CamScanner

## Conflicts in Shift-Reduce Parsing:

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

i) Shift/reduce conflict:

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

Example:

$A \rightarrow ab \mid abcd$

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.

ii) Reduce/reduce conflict:

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

Example:

$A \rightarrow bc$

$B \rightarrow abc$

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

Exercise: For the grammar $S \rightarrow SS+ \mid SS* \mid a$ indicate the handle in each of the following right sentential form.

i) SSS+a*+

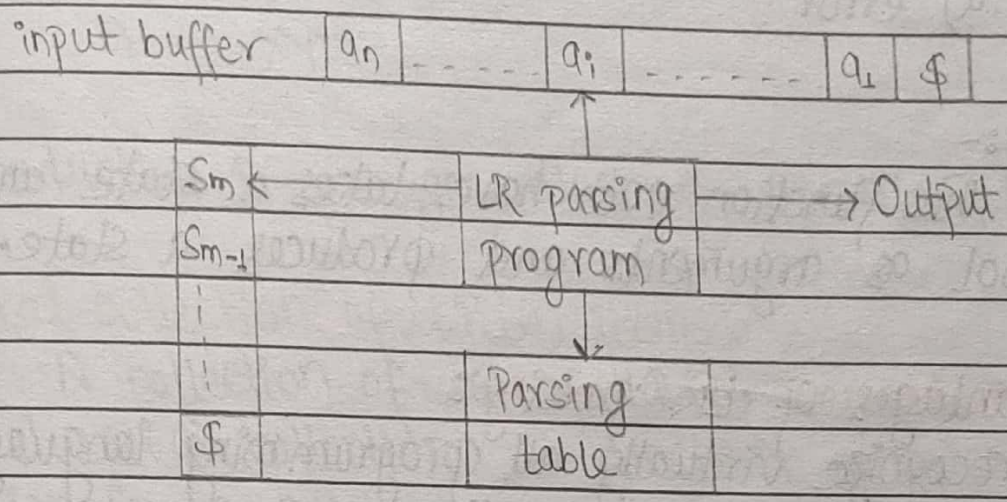
ii) SS+a*a

iii) aaa*a++

Scanned with CamScanner

## LR Parsers :-

LR parsing is the most general non-backtracking, efficient and most powerful shift-reduce parsing. In LR(k) parsing, the 'L' is for left-to-right scanning of the input, the 'R' for constructing rightmost derivation in reverse, and the 'k' for the number of input symbols of lookahead that are used in making parsing decisions. The general structure of LR parsing is shown in figure below.




[ Fig:- General structure of LR parser ]

- The program is same as for all LR parsers.
- The parsing program needs characters from an input buffer one at a time.
- The program uses a stack to store a string of the form $S_0X_1S_1X_2S_2X_3 \ldots X_mS_m$, where $S_m$ is on top. Each $X_i$ is a grammar symbol and each $S_i$ is a state.
- The parsing table consists of two parts: action and goto functions.

Scanned with CamScanner

i) action :-

The parsing program determines $S_m$, the state currently on top of stack, and $a_i$, the current input symbol. It then consult action $[S_m, a_i]$ in the action table which can have one of four values:

a) shift s, where s is a state
b) reduce by a grammar production $A \rightarrow \beta$
c) accept
d) error

ii) Goto :-

The function goto have takes a state and grammar symbol as arguments and produces a state.

## Advantages of LR parsing:

i) It recognize virtually all programming language constructs for which CFG can be written.
ii) It is an efficient non-backtracking shift-reduce parsing method.
iii) A grammar that can be parsed using LR method is a proper subset of a grammar that can be parsed with predictive or LL method.
iv) It detects a syntactic error as soon as possible.

## Drawbacks of LR method:

It is too much of work to construct a LR parser by hand for a programming language grammar. A specialized tool, called a LR parser generator, is needed. Example: YACC.

Scanned with CamScanner

## LR(0) item :-

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

Example :- The production $A \rightarrow XYZ$ yields the 4 items, which are:

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

The production $A \rightarrow \epsilon$ generates only one item, $A \rightarrow \cdot$.

## Canonical collection of LR(0) items :

A collection of set 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 function.

## 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 product $S' \rightarrow S$.

Example :- Consider the grammar.

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

Scanned with CamScanner

→ Its augmented grammar is :

$$E' \rightarrow E$$
$$E \rightarrow E+T \mid T$$
$$T \rightarrow T*F \mid F$$
$$F \rightarrow (E) \mid id.$$

## Closure Operation :-

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

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

Example :- Consider a grammar.

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

Sol$^n$:

Its augmented grammar is :

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

Scanned with CamScanner

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

$E' \rightarrow \cdot E$
$E \rightarrow \cdot E+T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T*F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot id$

## The goto operation:-

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

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

Example:-

Let,

$I_0$: $E \rightarrow \cdot E$
$E \rightarrow \cdot E+T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T*F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot id$

goto $(I_0, E)$

$I_1$: $E' \rightarrow E \cdot$
$E \rightarrow E \cdot +T$

goto $(I_0, T)$

$I_2$: $E \rightarrow T \cdot$
$T \rightarrow T \cdot *F$, etc.

Scanned with CamScanner

## Construction of canonical LR(0) collection:

Algorithm:

1. Augment the grammar by adding production $S' \rightarrow S$
2. $C = \{closure(\{S' \rightarrow \cdot S\})\}$
3. repeat the following until no more set of LR(0) items can be added to C.
   for each I in C and each grammar symbol X.
   if goto(I, X) is not empty and not in C
   add goto (I, X) to C.

Example 1: Consider the following grammar.

$$S \rightarrow AA$$
$$A \rightarrow aA \mid b$$

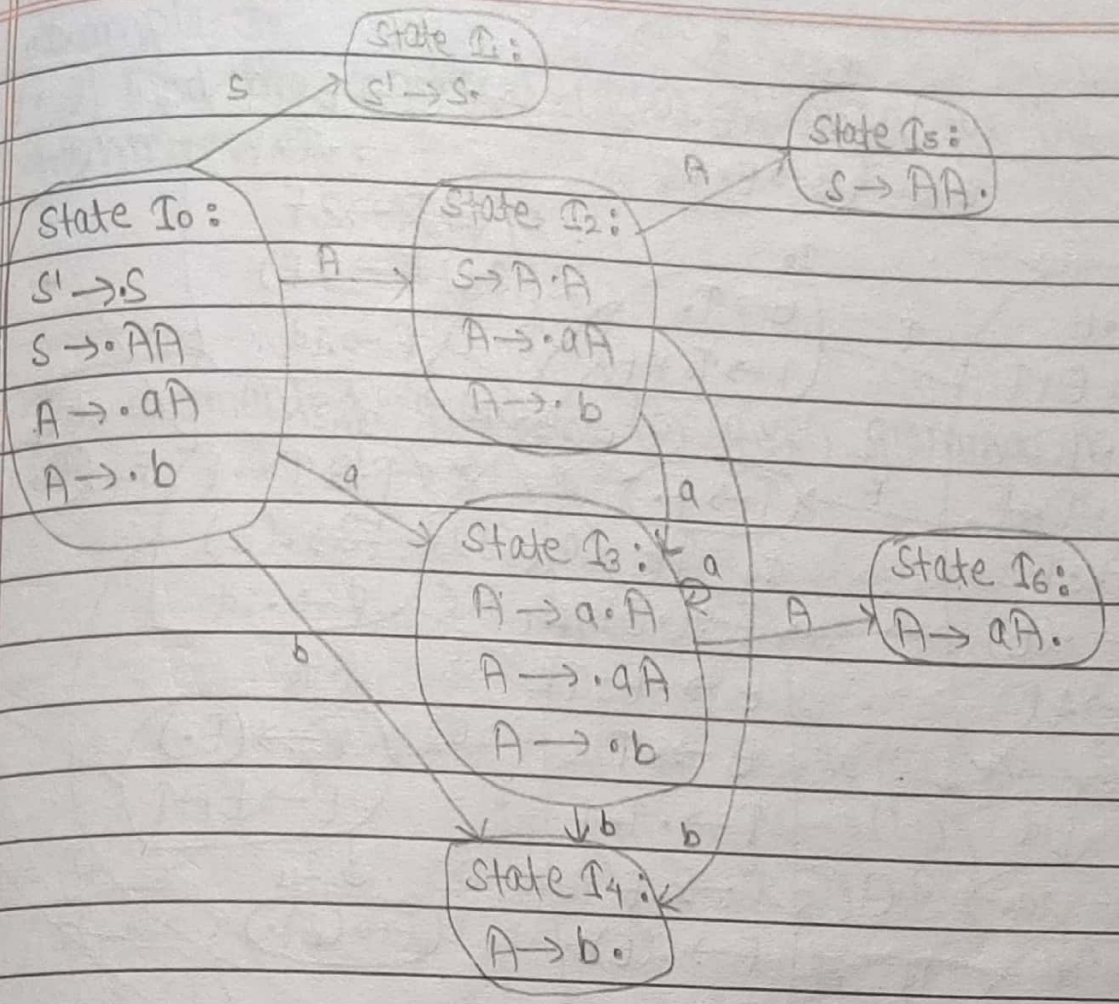
Find the canonical LR(0) collection for the grammar.

Soln:

The augmented grammar of given grammar is:

$$S' \rightarrow S$$
$$S \rightarrow AA$$
$$A \rightarrow aA \mid b$$

Scanned with CamScanner




Example: 2

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

$E' \rightarrow E$

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Scanned with CamScanner

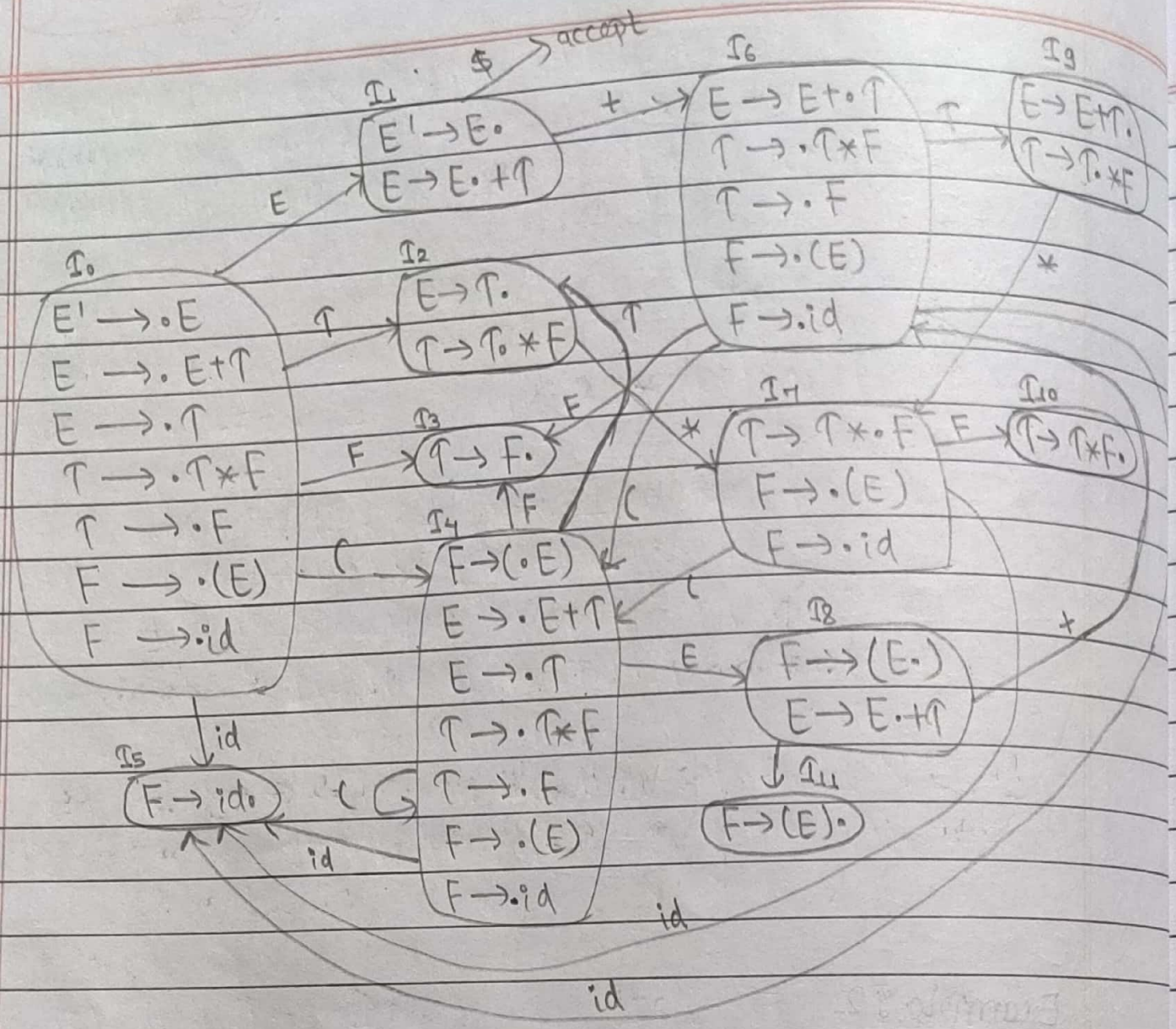



Fig:- automaton for expression grammar

Example 3:

Find the canonical LR(0) collection for the following grammar.

$S \rightarrow (L) \mid a$

$L \rightarrow L,S \mid S$

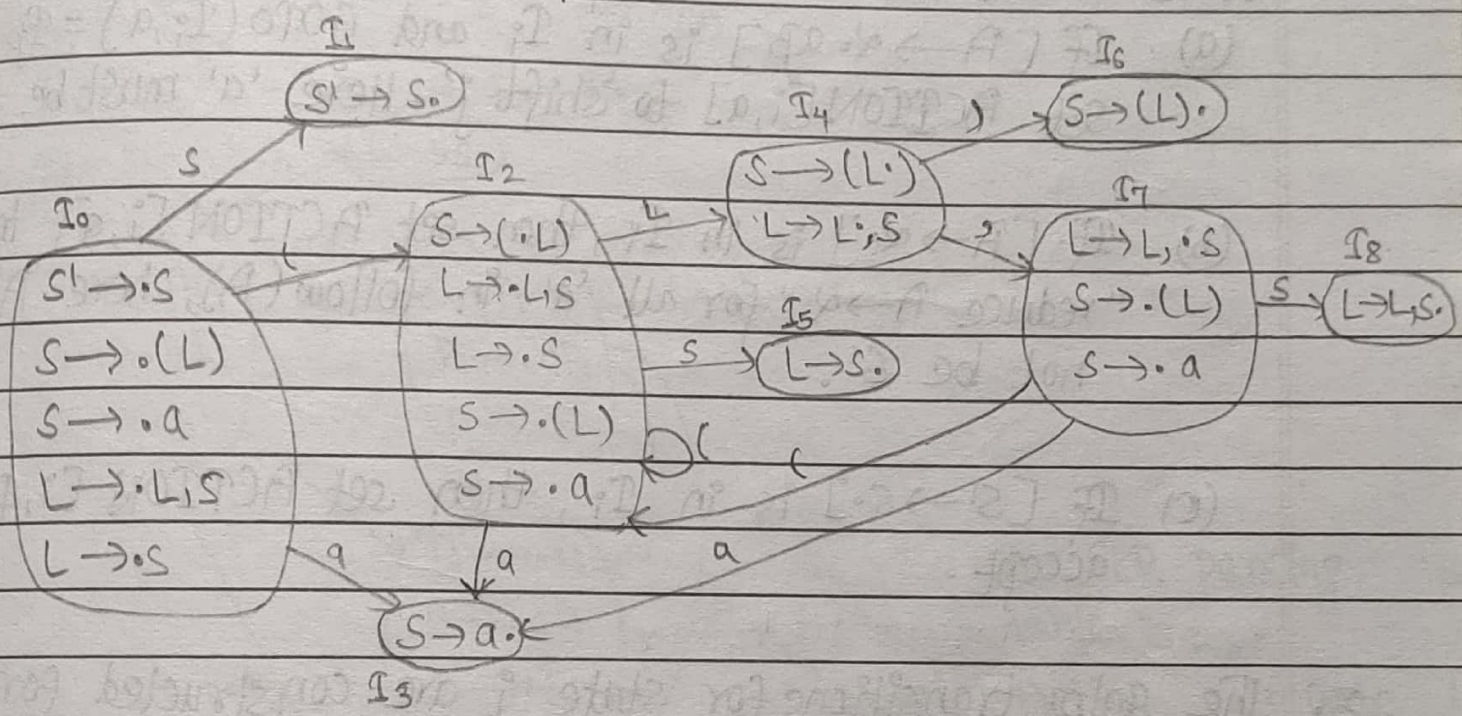
Sol:

The augmented grammar for given grammar is:

$S' \rightarrow S$

$S \rightarrow (L) \mid a$

$L \rightarrow L,S \mid S$



## Constructing SLR Parsing Table:

input: augmented grammar $G'$

output: The SLR-parsing table functions ACTION & GOTO for $G'$.

method:

1. Construct $C = \{I_0, I_1, ---, I_n\}$, the collection of sets of LR(0) items for $G'$.
2. State $i$ is constructed from $I_i$. The parsing actions for state $i$ are determined as follows:
   (a) If $[A \rightarrow \alpha . a\beta]$ is in $I_i$ and $GOTO(I_i, a) = I_j$, then set ACTION$[i, a]$ to 'shift $j$'. Here 'a' must be terminal.

   (b) If $[A \rightarrow \alpha .]$ is in $I_i$, then set ACTION $[i, a]$ to "reduce $A \rightarrow \alpha$" for all 'a' in follow(A), here 'A' may not be $S'$.

   (c) If $[S' \rightarrow S .]$ is in $I_i$, then set ACTION $[i, \$]$ to accept.
3. The goto transitions for state $i$ are constructed for all non-terminals A using the rule: If $GOTO(I_i, A) = I_j$, then $GOTO(i, A) = j$.
4. All entries not defined by rule (2) and (3) are made error.
5. The initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow .S]$.

Scanned with CamScanner

Example1:- Construct the SLR sets of items for the following grammar and compute the GOTO function for these sets of items. Show parsing table for this grammar.

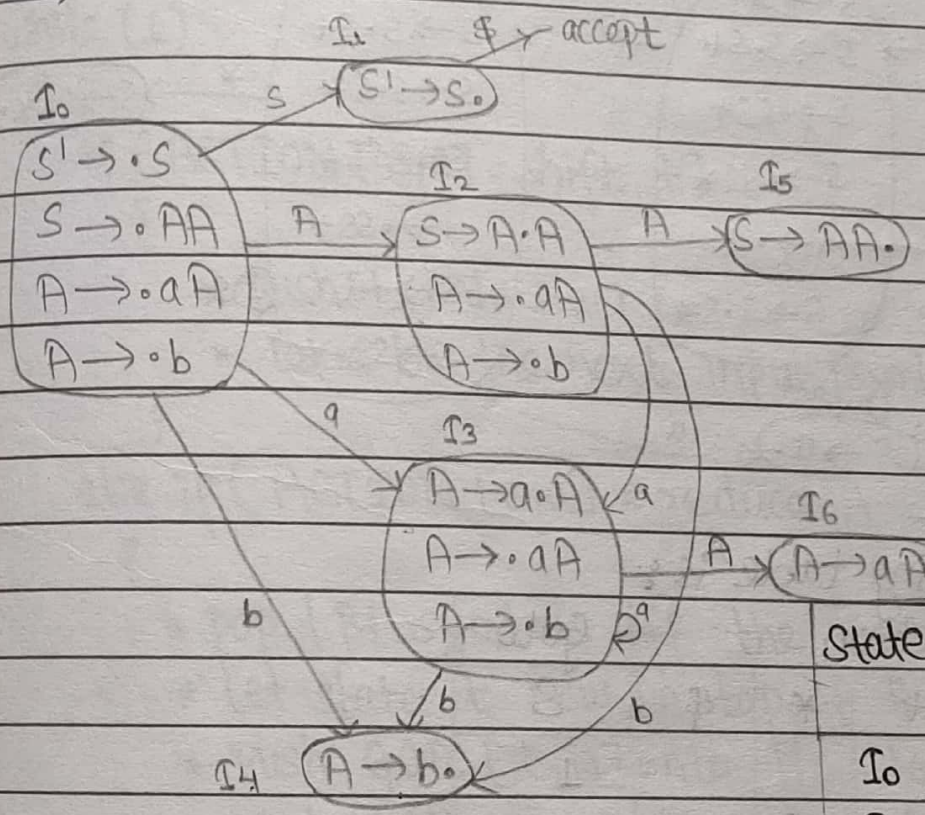
$$S \rightarrow AA$$
$$A \rightarrow aA \mid b$$

Sol:

Its augmented grammar

$$S' \rightarrow S$$
$$S \rightarrow AA \quad (1)$$
$$A \rightarrow aA \mid b$$
$$(2) \quad (3)$$

Then,




The SLR parsing table is:

| State | Action | | | Goto | |
|---|---|---|---|---|---|
| | a | b | $ | S | A |
| $I_0$ | $S_3$ | $S_4$ | | 1 | 2 |
| $I_1$ | | | accept | | |
| $I_2$ | $S_3$ | $S_4$ | | | 5 |
| $I_3$ | $S_3$ | $S_4$ | | | 6 |
| $I_4$ | $r_3$ | $r_3$ | $r_3$ | | |
| $I_5$ | $r_1$ | $r_1$ | $r_1$ | | |
| $I_6$ | $r_2$ | $r_2$ | $r_2$ | | |

Scanned with CamScanner

Example 2: Construct the SLR parsing table for the following grammar.

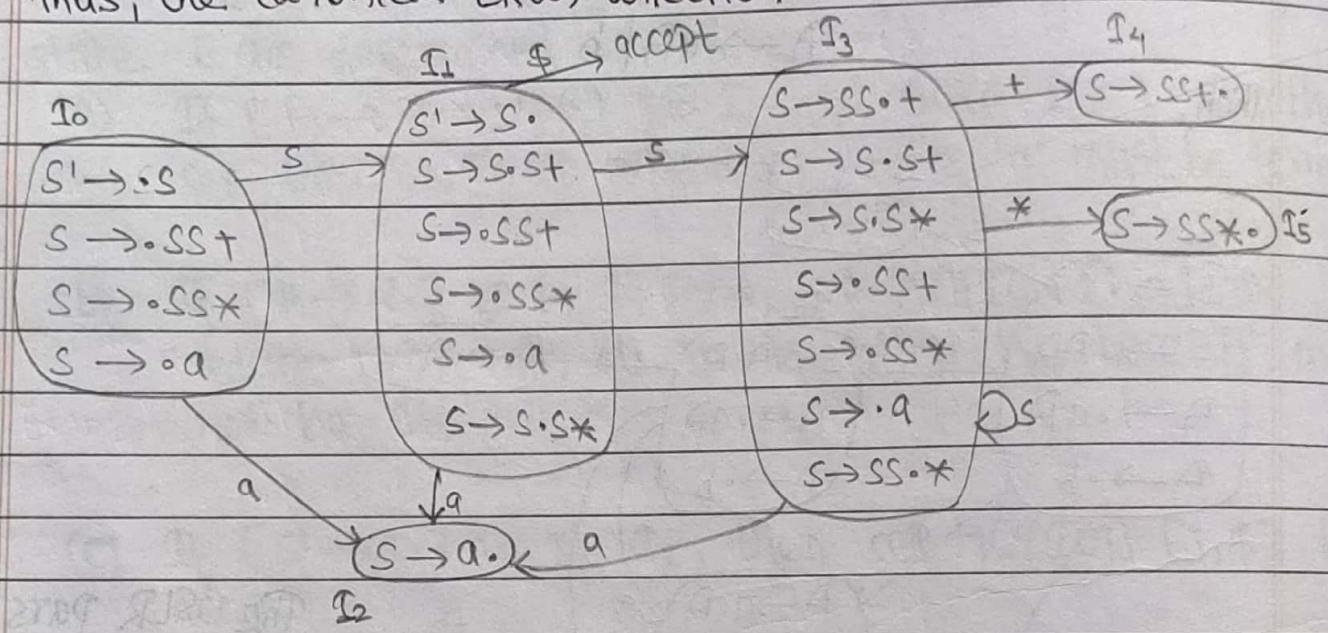
$$S \rightarrow SS+ \mid SS* \mid a$$

Sol$^n$:

Its augmented grammar is

$$S' \rightarrow S$$
$$S \rightarrow \underset{①}{SS+} \mid \underset{②}{SS*} \mid \underset{③}{a}$$

Thus, the canonical LR(0) collection is:




Now, SLR parsing table is:

| State | Action | | | | Goto |
|---|---|---|---|---|---|
| | a | + | * | $ | S |
| $I_0$ | $S_2$ | | | | 1 |
| $I_1$ | $S_2$ | | | accept | 3 |
| $I_2$ | $r_3$ | $r_3$ | $r_3$ | $r_3$ | |
| $I_3$ | $S_2$ | $S_4$ | $S_5$ | | 3 |
| $I_4$ | $r_1$ | $r_1$ | $r_1$ | $r_1$ | |
| $I_5$ | $r_2$ | $r_2$ | $r_2$ | $r_2$ | |

Scanned with CamScanner

## LR Parsing algorithm :-

input : An input string w and an LR parsing table with functions ACTION and GOTO for grammar G.

output : If w is in L(G), a bottom-up-parse for w, otherwise an error indication.

method:

1. Initially the parser has $S_0$ on stack, where $S_0$ is the initial state, and w$ in the input buffer. The parser then executes the following program.
2. Let 'a' be the first symbol of w$.
3. while (1)

```
{
  if (ACTION [s,a] = shift t)
   {
     * push t onto stack;
     * let a be the next input symbol;
   }
  else if (ACTION [s,a] = reduce A→β)
   {
     * pop |β| symbols off the stack
     * let state t now on top of the stack.
     * push GOTO [t,A] onto the stack.
     * output the production A→β.
   }
  else if (ACTION [s,a] = accept)
        break ;   |*parsing done*|
  else
        Call error recovery routine;
}
```

Scanned with CamScanner

Example1:- Show the actions of parsing table for the input aa*a+.

| Stack | Symbols | input | Action |
|---|---|---|---|
| 0 | | aa*a+$ | shift |
| 02 | a | a*a+$ | reduced by S→a |
| 01 | S | a*a+$ | shift |
| 012 | Sa | *a+$ | reduced by S→a |
| 013 | SS | *a+$ | shift |
| 0135 | SS* | a+$ | reduced by S→SS* |
| 01 | S | a+$ | shift |
| 012 | Sa | +$ | reduced by S→a |
| 013 | SS | +$ | shift |
| 0134 | SS+ | $ | reduced by S→SS+ |
| 01 | S | $ | accept |

Example 2: Show the actions of parsing table for the input aabb.

| Stack | Symbols | input | Action |
|---|---|---|---|
| 0 | | aabb$ | shift |
| 03 | a | abb$ | shift |
| 033 | aa | bb$ | shift |
| 0334 | aab | b$ | reduced by A→b |
| 0336 | aaA | b$ | reduced by A→aA |
| 036 | aA | b$ | reduced by A→aA |
| 02 | A | b$ | shift |
| 024 | Ab | $ | reduced by A→b |
| 025 | AA | $ | reduced by S→AA |
| 01 | S | $ | accept. |

Scanned with CamScanner

## Check whether given grammar is SLR(1) or not :

- Every SLR(1) grammar is unambiguous, but there are many unambiguous grammar that are not SLR(1).
- The SLR(1) grammar must not contain shift/reduce and reduce/reduce conflicts.

### i) Shift/reduce conflicts:

If in any state contains a completed item $[A \to \alpha.]$ and also any other non-completed item $[A \to \alpha.\beta]$, then first item puts reduce in parse table and second item puts the shift in parse table. This type of conflicts is called shift/reduce conflicts.

### ii) reduce/reduce conflicts:

If any state contains more than one final items then this situation is called reduce/reduce conflicts.

Example:

$A \to \alpha.$
$B \to \beta.$

Here, conflicts to choose which production to reduce.

Example 1:- Show that the following grammar is not SLR(1).

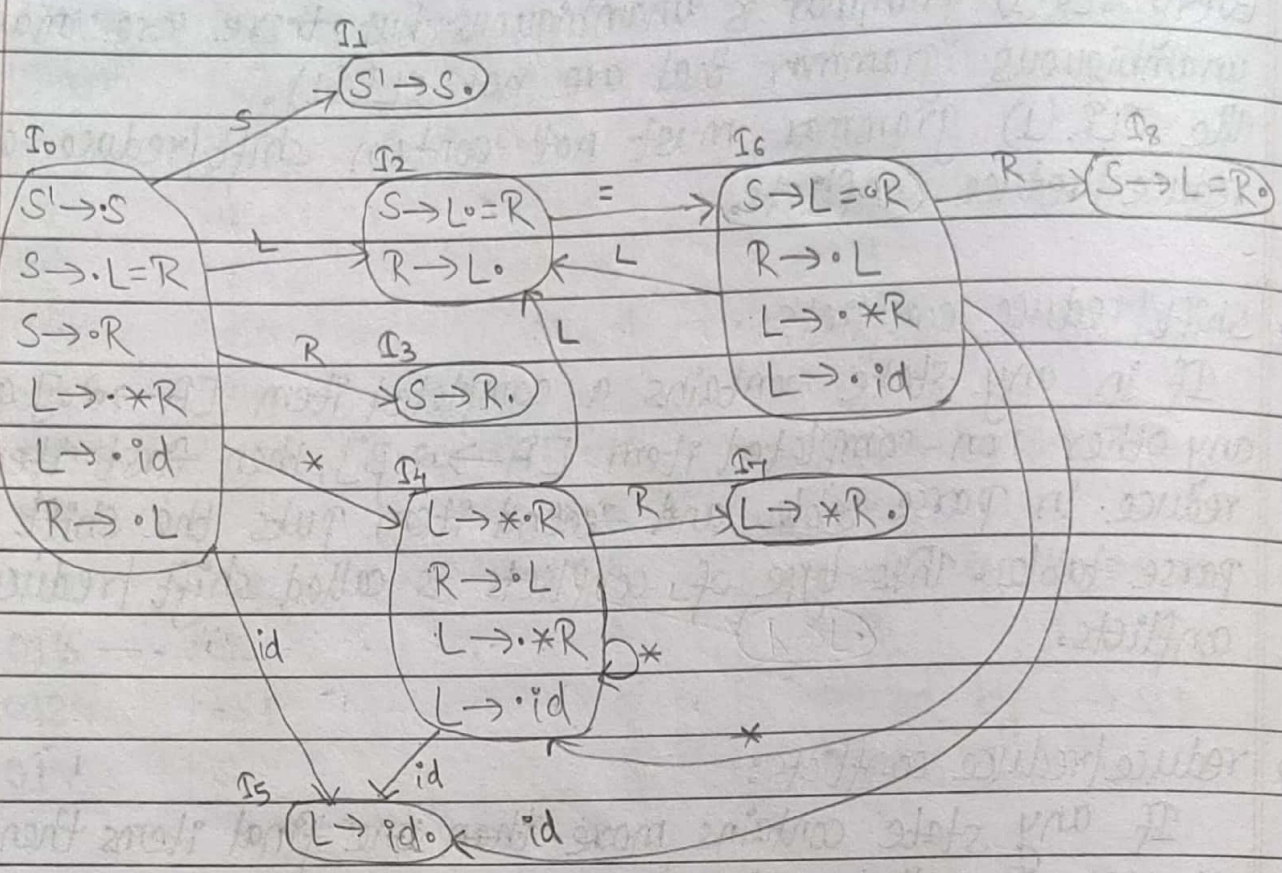
$S \to L=R \mid R$
$L \to *R \mid id$
$R \to L$

Sol: Its augmented grammar is:

$S' \to S$
$S \to L=R \mid R$
$L \to *R \mid id$

Scanned with CamScanner

The canonical collection of LR(0) item of the grammar is:




Here, in state $I_2$, the first item makes ACTION [2, =] be "shift 6". The second item makes ACTION [2, =] to reduce "R→L". Since follow(R) contains '=". Since there is both a shift and a reduce entry in ACTION [2, =], thus state 2 has shift/reduce conflict on input symbol '='. Therefore, the grammar is not SLR(1) grammar.

Example 2: Show that the following grammar

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

is LL(1) but not SLR(1).

Scanned with CamScanner

Sol: Here, first (AaAb) = {a} & first (BbBa) = {b}, these two sets are disjoint, thus the grammar is LL(1).

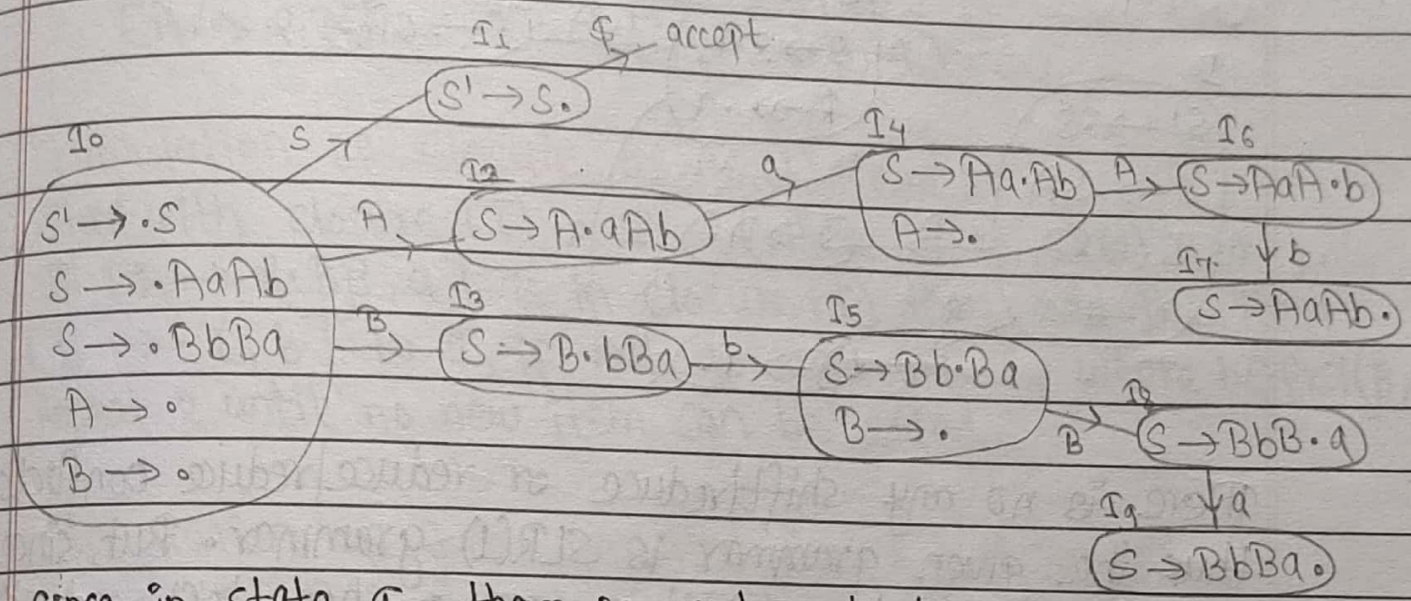
To check whether the given grammar is SLR(1) or not we have to construct canonical LR(0) items.

$S' \to S$

$S \to AaAb \mid BbBa$

$A \to \epsilon$

$B \to \epsilon$




Since, in state $I_0$, there is reduce/reduce conflicts between the production $A \to \cdot$ and $B \to \cdot$. Thus, the given grammar is not SLR(1) grammar.

Example 3: Show that the following grammar

$S \to SA \mid A$

$A \to a$

is SLR(1) but not LL(1).

Sol$^n$:

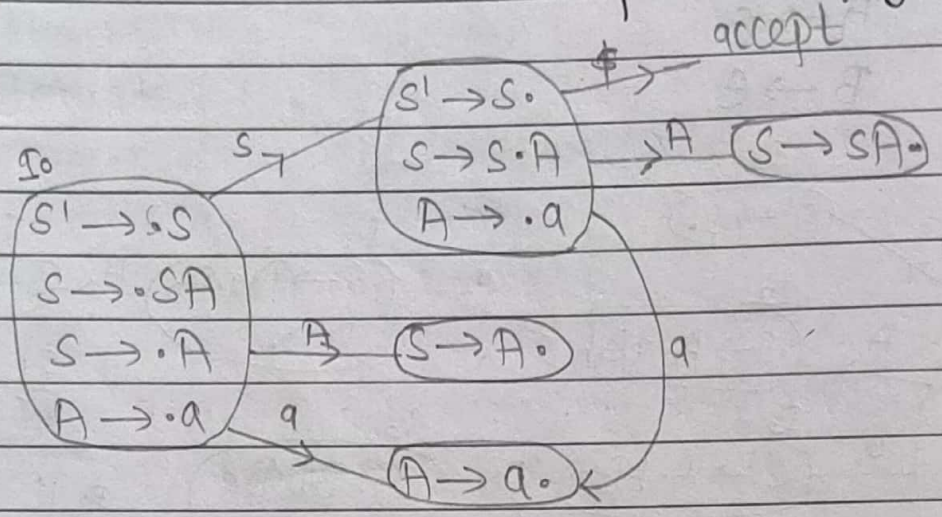
Its augmented grammar is:

$S' \rightarrow S$

$S \rightarrow SA \mid A$

$A \rightarrow a$

The canonical collection of items of given grammar is:




There is no any shift/reduce or reduce/reduce conflicts, thus the given grammar is SLR(1) grammar. But, since, $S \rightarrow SA$ production of the grammar has left recursion. Thus the grammar is not LL(1) grammar.

## Viable Prefixes:

The prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called viable prefixes.

Scanned with CamScanner

## LR(1) Grammar :-

SLR(1) 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) Items :-

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

Eg:

| LR(0) item | LR(1) item |
|---|---|
| $[A \rightarrow \alpha . \beta]$ | $[A \rightarrow \alpha . \beta, a]$ |

## Computation of Closure of LR(1) items :-

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

## Computation of Goto operation for LR(1) items :-

Let I is a set of LR(1) items and is a grammar symbol (terminal or non-terminal) then Goto [I, X] is computed as follows :-

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

Scanned with CamScanner

## Computation of Canonical LR(1) collection:

1. Augment the grammar with production $S' \to S$.
2. $C = \{ closure(\{S' \to \cdot S, \$\}) \}$ // the start state.
3. repeat
   for each $I \in C$ & each grammar symbol $X \in (N \cup T)$
   goto $(I, X) \neq \phi$ & goto $(I, X) \notin C$ then,
   add goto $(I, X)$ to $C$.

Example:

Construct the LR(1) collection for the grammar.

$S \to AA$

$A \to aA \mid b$.

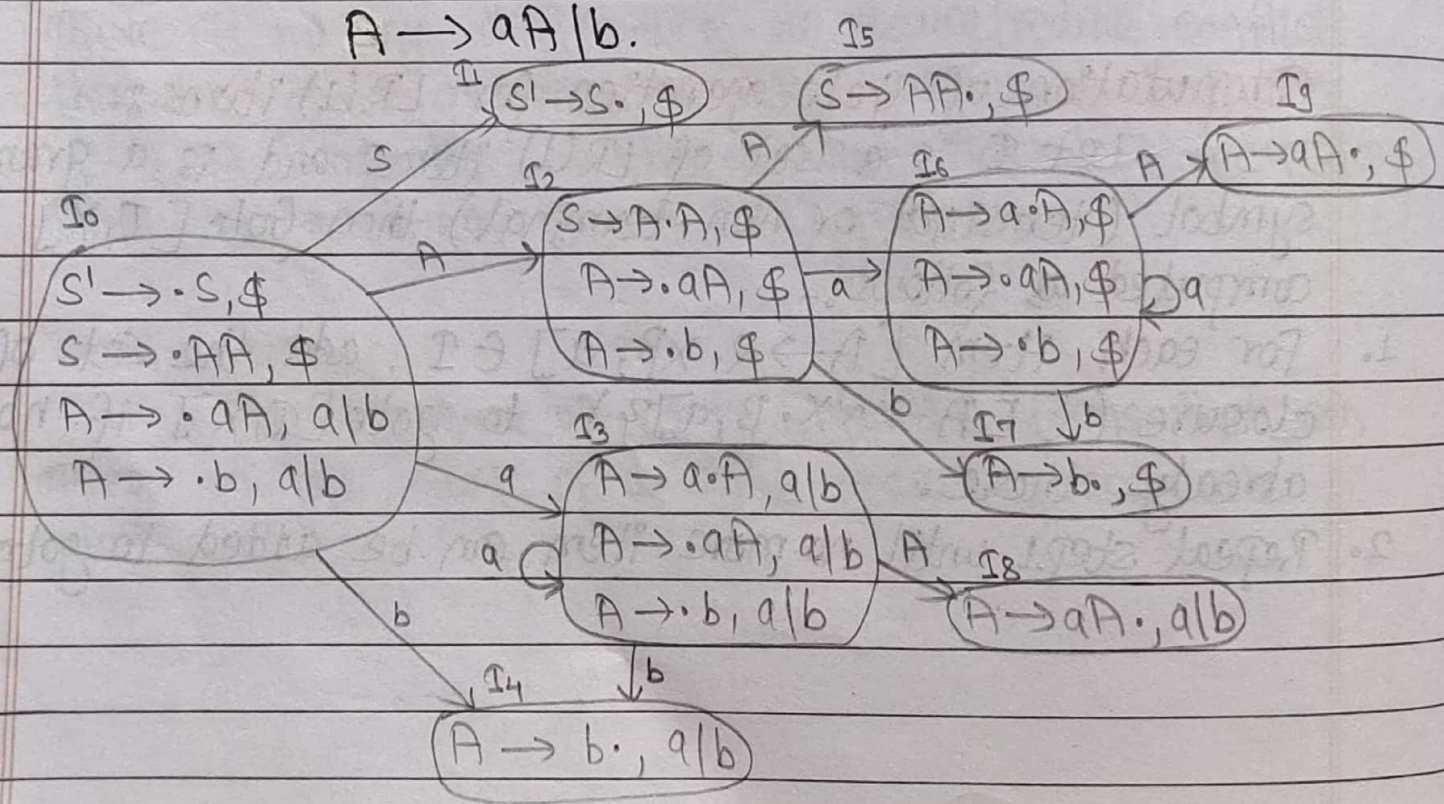
Sol:

The augmented grammar of above grammar is;

$S' \to S$

$S \to AA$

$A \to aA \mid b$.




Scanned with CamScanner

## Canonical LR

This is a more complex form of LR parsers. It uses LR(1) items that are augmented by the set of look ahead symbols that can follow the use of the underlying production. Reductions are only choosen when there is a valid item with dot at the right end and the current lookahead symbol is one of those allowed for this item. The CLR parser can avoid some of the parsing action conflicts that are present in SLR parser. But often has many more states than the SLR parser for the same grammar.

## CLR(1) parsing table :-

input: An augmented grammar G'.

output: The canonical LR parsing table functions ACTION & GOTO of G'.

Method:

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(1) items for G'.
2. State i of the parser is constructed from $I_i$. The parsing action for state 'i' is determined as follows:-
   a) If $[A \to \alpha . a\beta, b]$ is in $I_i$ and $GOTO(I_i, a) = I_j$ then set ACTION [i, a] to "shift j". Here 'a' must be terminal.
   b) If $[A \to \alpha ., a]$ is in $I_i$, $A \neq S'$, then set ACTION [i, a] to "reduce $A \to \alpha$".
   c) If $[S' \to S ., \$]$ is in $I_i$, then set ACTION [i, \$] to "accept".

Scanned with CamScanner

3. The goto transitions for state i are constructed for all non-terminals 'A' using the rule: If GOTO[$I_i$, A] = $I_j$ then GOTO[i, A] = j.
4. All entries not defined by rule (2) and (3) are made error.
5. The initial state of the parser is the one constructed from the set of items containing [S' → .S, $].

Example: Construct CLR(1) parsing table for the following grammar.

$S \rightarrow AA$

$A \rightarrow aA \mid b$

Sol:

| State | ACTION | | | GOTO | |
|---|---|---|---|---|---|
| | a | b | $ | S | A |
| $I_0$ | $S_3$ | $S_4$ | | 1 | 2 |
| $I_1$ | | | accept | | |
| $I_2$ | $S_6$ | $S_7$ | | | 5 |
| $I_3$ | $S_3$ | $S_4$ | | | 8 |
| $I_4$ | $r_3$ | $r_3$ | | | |
| $I_5$ | | | $r_1$ | | |
| $I_6$ | $S_6$ | $S_7$ | | | 9 |
| $I_7$ | | | $r_3$ | | |
| $I_8$ | $r_2$ | $r_2$ | | | |
| $I_9$ | | | $r_2$ | | |

[Fig:- CLR(1) parsing table]

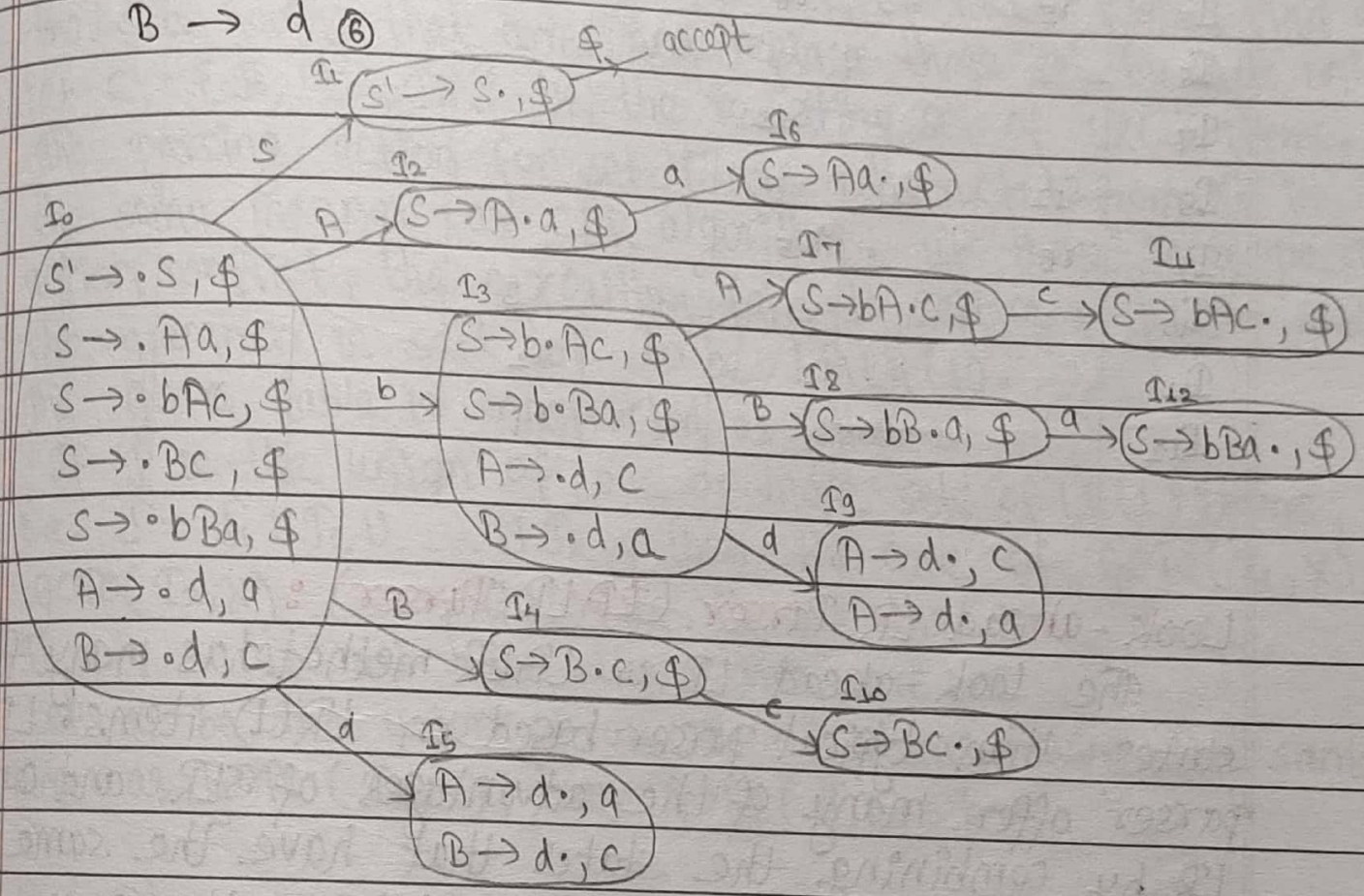
Scanned with CamScanner

2. $S \to Aa \mid bAc \mid Bc \mid bBa$
$A \to d$
$B \to d$

Sol:

Its augmented grammar is;

$S' \to S$ ①
$S \to Aa \mid bAc \mid Bc \mid bBa$ ② ③ ④
$A \to d$ ⑤
$B \to d$ ⑥



| State | ACTION | | | | | GOTO | | |
|---|---|---|---|---|---|---|---|---|
| | a | b | c | d | $ | S | A | B |
| $I_0$ | | $S_3$ | | $S_5$ | | 1 | 2 | 4 |
| $I_1$ | | | | | accept | | | |
| $I_2$ | $S_6$ | | | | | | | |
| $I_3$ | | | | $S_9$ | | | 7 | 8 |
| $I_4$ | | | $S_{10}$ | | | | | |
| $I_5$ | $r_5$ | | $r_6$ | | | | | |
| $I_6$ | | | | | | | | |
| $I_7$ | | | $S_{11}$ | | | | | |
| $I_8$ | $S_{12}$ | | | | | | | |
| $I_9$ | $r_6$ | | $r_5$ | | | | | |
| $I_{10}$ | | | | | $r_3$ | | | |
| $I_{11}$ | | | | | $r_2$ | | | |
| $I_{12}$ | | | | | $r_4$ | | | |

## Look-ahead LR Parser (LALR Parser):

The look-ahead LR or LALR method has many fewer states than typical parser based on LR(1) items. LALR parser offer many of the advantages of SLR and canonical LR by combining the states that have the same kernels. Thus, the number of states is the same as that of the SLR parser. But some parsing action conflict present in the SLR parser may be removed in the LALR parser. LALR parser is intermediate between the SLR and CLR parser.

Scanned with CamScanner

## Construction of LALR parsing table:

input : An augmented grammar $G'$.
output : The LALR parsing table functions ACTION & GOTO for $G'$.

Method:

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of set of LR(1) items.
2. For each core present among the sets of LR(1) items, find all the sets having that core, and replace these sets by their union.
3. Let $C' = \{J_0, \ldots, J_n\}$ be the resulting sets of LR(1) items. The parsing action for set 'i' are constructed from $J_i$ in the same manner as CLR algorithm. If there is a parsing action conflict, the algorithm fails to replace a parser and the grammar is said not to be LALR(1).
4. The GOTO table is constructed as follows:
   If $J$ is the union of one or more sets of LR(1) items, i.e. $J = I_1 \cup I_2 \cup \ldots \cup I_k$, then the core of $GOTO(I_1, X)$, $GOTO(I_2, X)$ ....., $GOTO(I_k, X)$ are the same since, $I_1, I_2, \ldots I_k$ all have the same core.
   Let $K$ be the union of all sets of items having the same core as $GOTO(I_1, X)$, then $GOTO(J, X) = K$.

Example:-

Consider the following grammar and construct the LALR(1) parsing table.

$S \rightarrow AA$

$A \rightarrow aA \mid b$.

Scanned with CamScanner

Sol:

The augmented grammar is;

$S' \rightarrow S$

$S \rightarrow AA$

$A \rightarrow aA \mid b$

Now,

There are pairs of sets of items that can be merged.

i) $I_3$ and $I_6$ are replaced by their union

$I_3 I_6$ . $A \rightarrow a.A, a|b|\$$

$A \rightarrow .aA, a|b|\$$

$A \rightarrow .b, a|b|\$$

ii) $I_4$ and $I_7$ are represented by their union.

$I_4 I_7$ . $A \rightarrow b., a|b|\$$

iii) $I_8$ and $I_9$ are represented by their union

$I_8 I_9$ $A \rightarrow aA., a|b|\$$

Then, the LALR(1) ACTION & GOTO function is;

| State | ACTION | | | GOTO | |
|---|---|---|---|---|---|
| | a | b | $ | S | A |
| 0 | $S_{36}$ | $S_{47}$ | | 1 | 2 |
| 1 | | | accept | | |
| 2 | $S_{36}$ | $S_{47}$ | | | 5 |
| 36 | $S_{36}$ | $S_{47}$ | | | 89 |
| 47 | $r_3$ | $r_3$ | $r_3$ | | |
| 5 | | | $r_1$ | | |
| 89 | $r_2$ | $r_2$ | $r_2$ | | |

Scanned with CamScanner

## Kernel items:

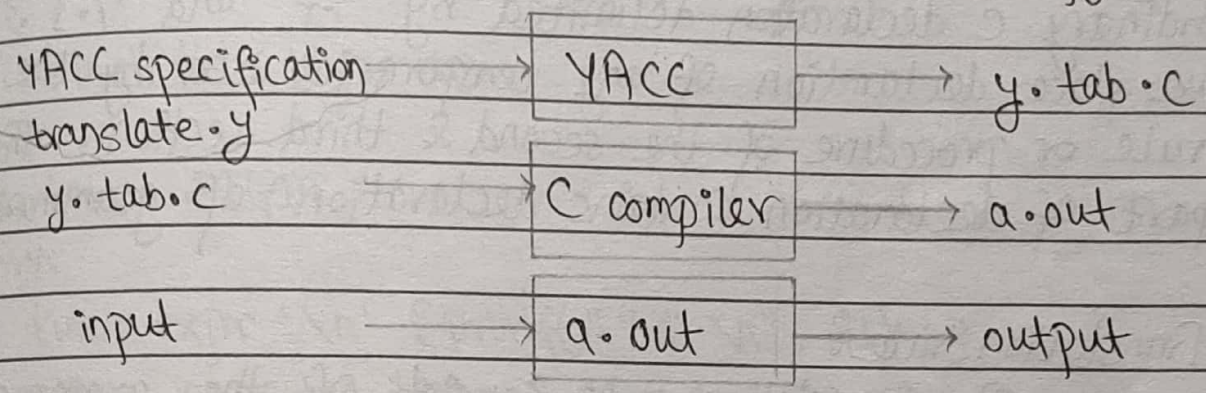
- The initial item $S' \rightarrow \cdot S$ & all items whose dot are not at the left end.

## Non-kernel items:

All items with their dots at the left end except for this $S' \rightarrow \cdot S$.

## The parser generator YACC:

YACC stands for Yet Another Compiler Compiler. It is a popular parser generator tool. A translator can be constructed using YACC in the manner illustrated as the figure below:




- First file, say translate.y containing a YACC specification of the translator is prepared.
- Then, it is transformed into C program called y.tab.c using YACC compiler.
- After compiling the C program y.tab.c, we obtain the desired object program a.out that performs the translation specified by the original YACC program.

Scanned with CamScanner

A YACC source program has three parts:

declaration
%%
translation rule
%%
Supporting C routine

Declaration:
There are two sections of declaration part of YACC program. Both are optional. In the first section, we put ordinary C declaration delimited by %{ and %}. Here, we place declaration of any temporaries used by the translation rule or procedure of the second & third section. The second part of declaration contains declaration of grammar token.

Translation rule:
The translation rule consists of the grammar production and associated semantic action. A set of production that we have been writing.

<head> → <body1> | <body2> ---- <body n> would be written in YACC as;

```
<head> : <body1>    { <semantic Action1> }
       |<body 2>    { <semantic Action 2> }
       ........     ..........
       |<body n>    { <semantic Action n> }
       ;
```

Supporting C routine:

The third part of YACC specification consists of supporting C routine.

Example;

Construct a simple desk calculator that reads an arithmetic operation, evaluate it and then prints its numeric value by YACC.

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

```
%{
   #include <ctype.h>
%}
%token DIGIT
%%
  line: expr '\n' {printf("%d \n", $1);}
  expr: expr '+' term { $$ = $1 + $3 ;}
  \term
  ;
  factor: '(' expr ')' { $$ = $2;}
  / DIGIT
  ;
%%
  yylex()
  {
    int c;
    c = getchar();
```

```
if ( isdigit(c))
  {
    yyval = c;
    return DIGIT;
  }
return c;
}.
```

Scanned with CamScanner

## 2.3) Syntax Directed Translation

To translate a programming language construct, a compiler needs to keep track of many quantities to the grammar symbol. There are two notation for associating semantic rules with production which are:

i) Syntax Directed Definition
ii) Syntax Directed Translation Scheme.

### Syntax Directed Definition:

A Syntax Directed Definition (SDD) is a CFG with attribute and rules. Attributes are associated with grammar symbol and rules are associated with production. If 'X' is a symbol and 'a' is one of its attribute, then we write x.a to denote the value of 'a' at a particular parse tree node labelled 'X'.

For example:

The syntax directed definition for simple desk calculator:

| | Production | Semantic Rules |
|---|---|---|
| i) | $L \rightarrow E$ return | $L.val = E.val$ |
| | $E \rightarrow E_1 + T$ | $E.val = E.val + T.val$ |
| | $E \rightarrow T$ | $E.val = T.val$ |
| | $T \rightarrow T_1 * F$ | $T.val = T_1.val * F.val$ |
| | $T \rightarrow F$ | $T.val = F.val$ |
| | $F \rightarrow (E)$ | $F.val = E.val$ |
| | $F \rightarrow digit$ | $F.val = digit.lexval$. |

Scanned with CamScanner

## Annotated Parse Tree :

A tree constructing for a given input string in which each node showing the values of attribute is called annotated Parse Tree.

Example,

Lets take an example for above grammar,

Now,

The annotated parse tree for the input string 5+3*4 is ;

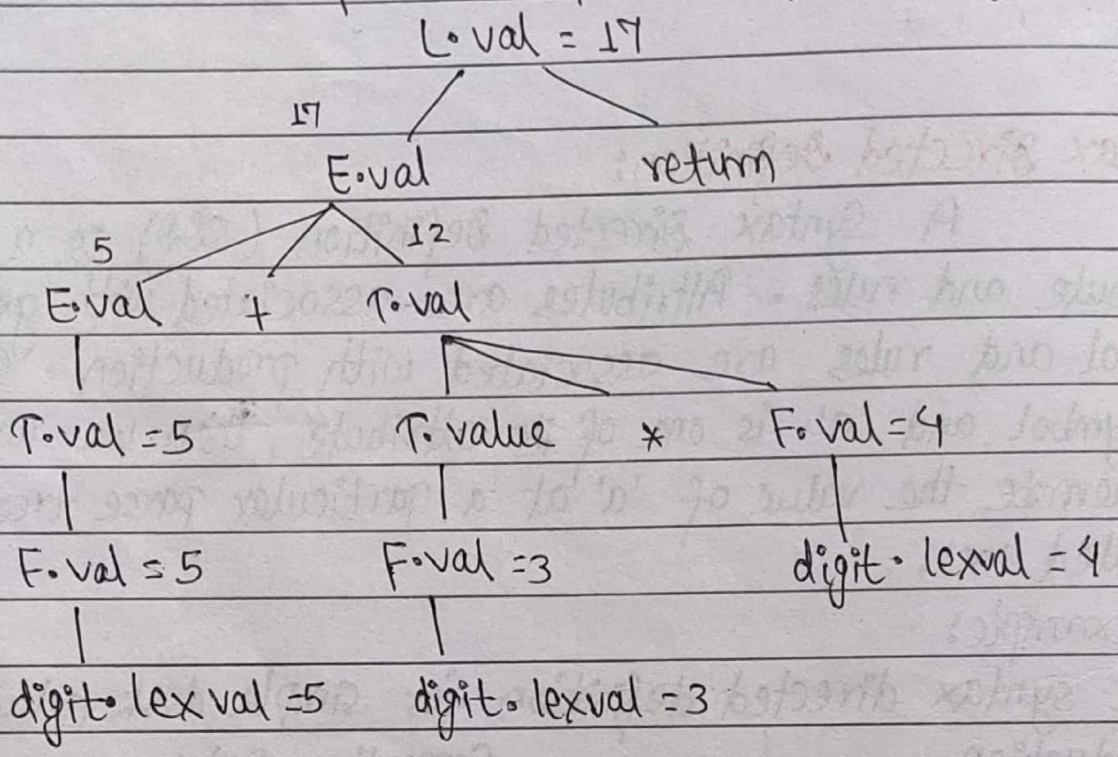



Fig: Annotated Parse tree for 5+3*4.

Scanned with CamScanner

## Inherited and Synthesized attribute :

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

Example:

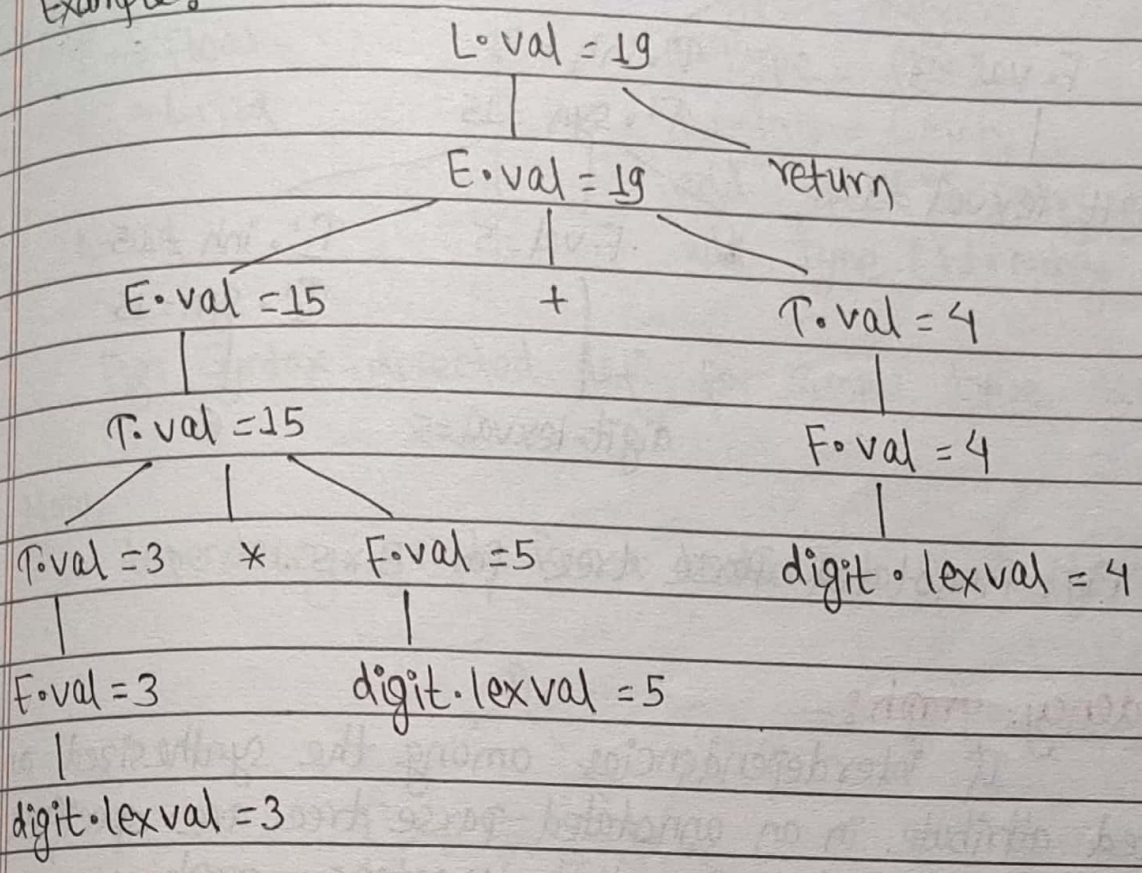



Fig: Annotated Parse tree for 3*5+4

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

Example:

| | Production | Semantic Rules |
|---|---|---|
| i) | $T \rightarrow FT'$ | $T'.inh = F.value$ $\quad T.val = T'.syn$ |
| ii) | $T' \rightarrow *FT_1'$ | $T_1'.inh = T'.inh * F.val$ $\quad T'.syn = T_1'.syn$ |
| iii) | $T' \rightarrow \epsilon$ | $T'.syn = T'.inh$ |
| iv) | $F \rightarrow digit$ | $F.val = digit.lexval$ |

Scanned with CamScanner

Now,

The annotated parse tree for 3*5,

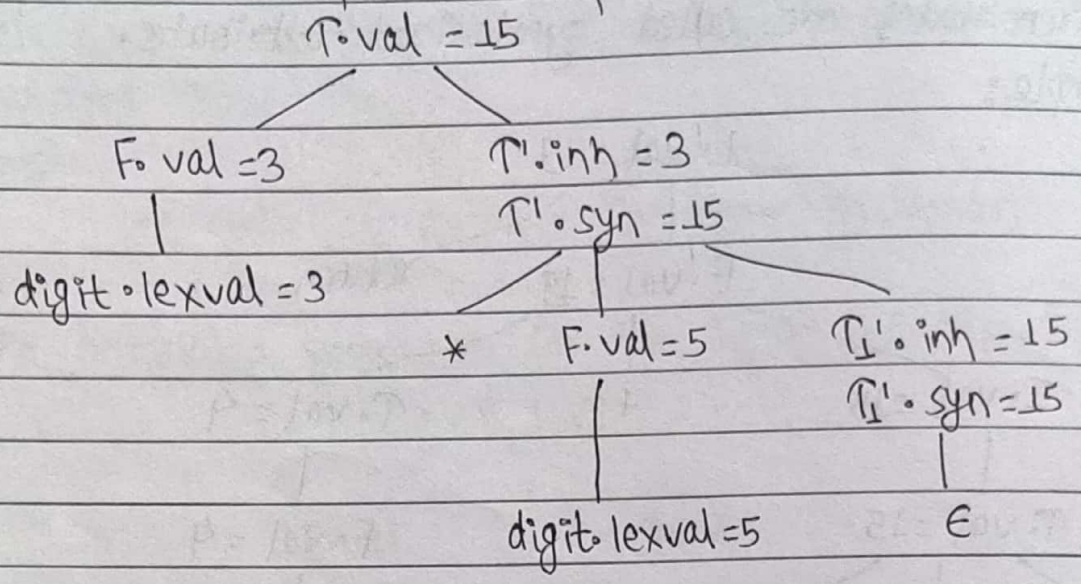



Fig: Annotated Parse tree for 3*5.

## Dependency graph:-

If interdependencies among the synthesized and inherited attribute 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 directed graph that contains attribute as node and dependencies across attribute as edges.

Scanned with CamScanner

For example;

| | Production | Semantic Rules |
|---|---|---|
| | $D \rightarrow TL$ | $L.inh = T.type$ |
| 1. | $T \rightarrow int$ | $T.type = integer$ |
| 2. | $T \rightarrow float$ | $T.type = float$ |
| 3. | $L \rightarrow L_1, id$ | $L_1.inh = L.inh$ |
| 4. | | add Type (id.entry, L.inh) |
| 5. | $L \rightarrow id$ | add Type (id.entry, L.inh) |

Fig: Syntax-directed def$^n$ for simple type declaration.

Now,
The dependency graph for a declaration float $id_1, id_2, id_3$.

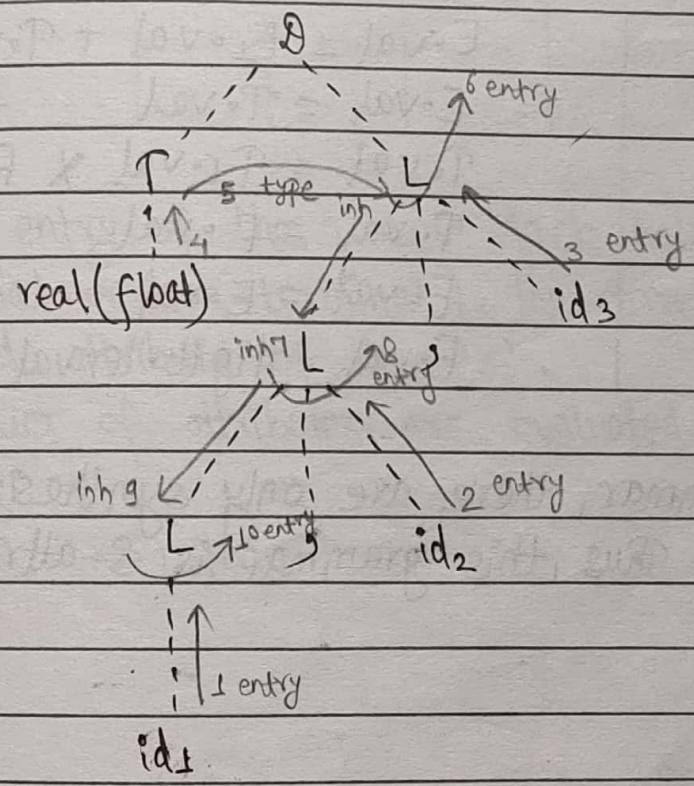



Fig: Dependency graph for a declaration float $id_1, id_2, id_3$.

Scanned with CamScanner

## S-attributed Definition:-

Syntax directed definition that uses synthesized attribute exclusively is called S-attributed definition or S-attributed grammar.

A parse tree for an S-attributed definition is annotated by evaluating the semantic rules for the attribute at each node in bottom of manner.

YACC or Bison only supports S-attributed definition.

Example; The syntax directed definition for a simple desk calculator.

| Production | Semantic Rules |
|---|---|
| $L \rightarrow E$ return | $L.val = E.val$ |
| $E \rightarrow E_1 + T$ | $E.val = E_1.val + T.val$ |
| $E \rightarrow T$ | $E.val = T.val$ |
| $T \rightarrow T_1 * F$ | $T.val = T_1.val \times F.val$ |
| $T \rightarrow F$ | $T.val = F.val$ |
| $F \rightarrow (E)$ | $F.val = E.val$ |
| $F \rightarrow digit$ | $F.val = digit.lexval$ |

In above grammar, there are only synthesized attribute val & lexval. Thus, this grammar is S-attributed grammar.

Scanned with CamScanner

## Bottom-up evaluation of S-attributed definition:

An S-attributed grammar can be translated bottom-up & when the grammar is being parsed using an L-R parser. The LR parser stack can be extended to hold not only grammar symbol but also its semantic attribute as well.

Hence,

ACTION [S, a] = shift 'S' becomes ACTION [S, a] = shift a. $$, S

A → XYZ  A.a = f(X.x, Y.y, Z.z) where all attributes are synthesized.

| | State | Value | | | |
|---|---|---|---|---|---|
| Top → | Z | Z.z | | | |
| | Y | Y.y | | | |
| | X | X.x | A | A.a | ← Top |
| | --- | ---- | --- | --- | |
| | --- | ---- | --- | --- | |

When an entry of the parser stack holds a grammar symbol X (terminal or non-terminal), the corresponding entry hold synthesized attribute of the 'X'.

The values of attributes are evaluated during reduction.

Example:

3*5+4n.

3*5+4n

| Stack | Value | Input | Action | Synthesized Rule |
|---|---|---|---|---|
| $ | - | 3*5+4n $ | Shift | |
| $3 | 3 | *5+4n $ | Reduce F→digit | $$ = $1 |
| $F | 3 | *5+4n $ | Reduce T→F | $$ = $1 |
| $T | 3 | *5+4n $ | Shift | |
| $T* | 3 | 5+4n $ | Shift | |
| $T*5 | 3,5 | +4n $ | Reduce F→digit | $$ = $1 |
| $T*F | 3,5 | +4n $ | Reduce T→T*F | $$ = $1 * $2 |
| $T | 15 | +4n $ | Reduce E→T | $$ = $1 |
| $E | 15 | +4n $ | Shift | |
| $E+ | 15 | 4n $ | Shift | |
| $E+4 | 15,4 | n $ | Reduce F→digit | $$ = $1 |
| $E+F | 15,4 | n $ | Reduce T→F | $$ = $1 |
| $E+T | 15,4 | n $ | Reduce E→E+T | $$ = $1 + $2 |
| $E | 19 | n $ | Shift | |
| $En | 19,n | $ | Reduce L→En | $$ = $1 (Print $1) |
| $L | 19,n | $ | accept | |

E → E+T|T
T → T*F|F
F → (E)|digit.

## L-attributed definition :

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

Mathematically,

a syntax directed definition is L-attributed if each inherited attribute of $X_j$, $1 \le j \le n$ on right side of $A \to X_1 X_2, \ldots, X_n$ and it depends,

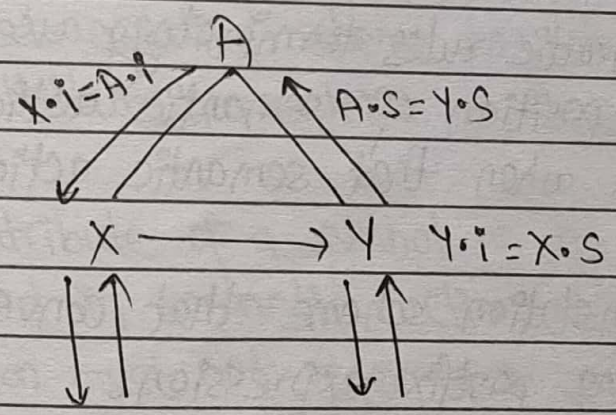
1. The attribute of the symbol $X_1, X_2, \ldots, X_{j-1}$ to the left of $X_j$ &
2. The inherited attribute of A.

$$A \to XY$$

Every S-attributed definition is L-attributed, the restriction only applied to the inherited attributes.

Example;

$$A \to XY$$



## Top-Down Translation: (Predictive Parsing)

L-attributed definition can be evaluated in top-down fashion with translation scheme.

The algorithm for elimination of left recursion is extended to evaluate action and attribute. Attributes in L-attributed definition implemented in translation scheme are fast as argument to procedure synthesized (or return) inherited.

## Syntax Directed Translation Scheme:

A translation scheme is a context free grammar in which attributes are associated with grammar symbol and semantic action are inserted within the right side of production and are enclosed within {}.

Example:

$$A \rightarrow \{\ldots\} X \{\ldots\} \{\ldots\}$$

semantic actions

In translation scheme, we use semantic action terminology instead of semantic rules terminology used in syntax directed definition. The position of semantic action on the right side indicates when that semantic action will be evaluated.

For example;

A simple translation scheme that converts infix expression to corresponding postfix expression.

$E \rightarrow TR$

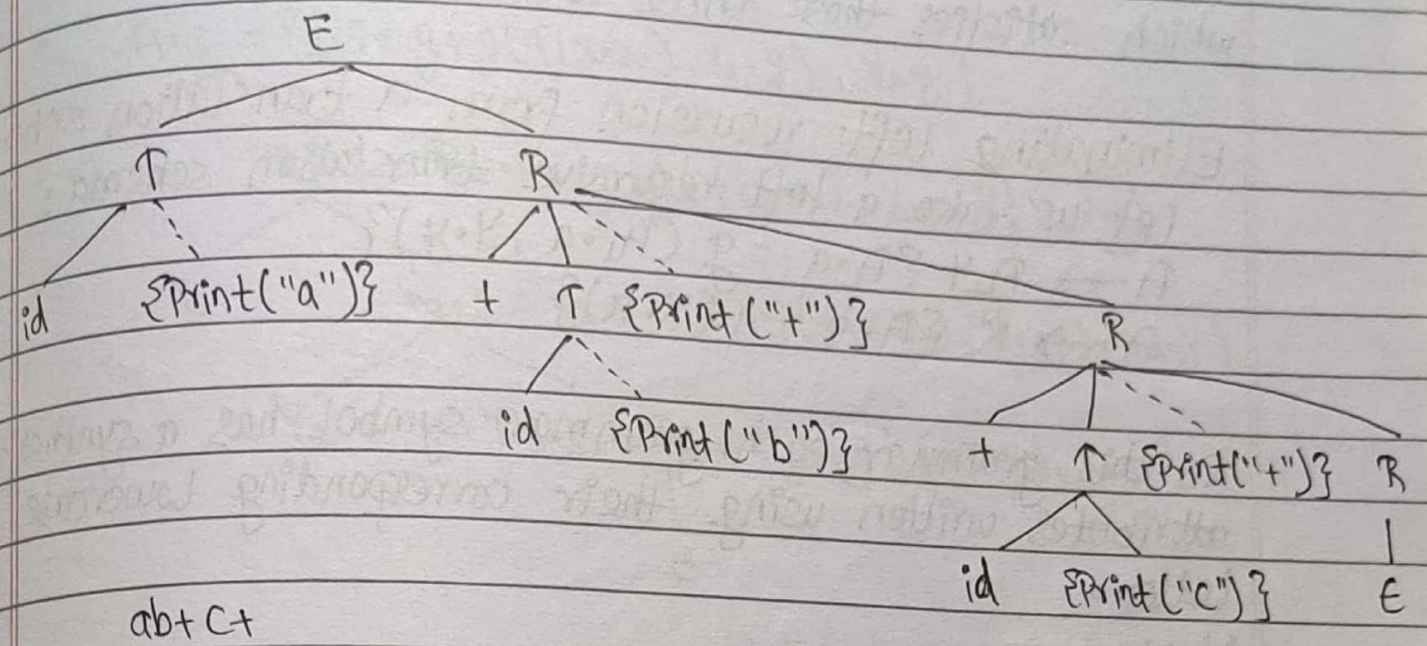
$R \rightarrow +T \{Print("+")\} R$

$R \rightarrow \epsilon$

$T \rightarrow id \{Print(id.name)\}$

Scanned with CamScanner

Here,
a+b+c (infix) & ab+c+ (Postfix expression)




The depth first traversal of the parse tree (executing the semantic action in that order) will produce postfix expression of the infix expression.

## Translation Scheme Requirement:

If a translation scheme has contain both synthesized and inherited attribute, we have to observe the following rules:

i) An inherited attribute of a symbol on the right side of a production must be computed in semantic action before that symbol.
ii) A semantic action must not refer to a synthesized attribute of a symbol to the right of that semantic action.
iii) A synthesized attribute of the non-terminal on the left can only be computed after all attributes it references have been computed.

Scanned with CamScanner

With an L-attributed syntax directed definition, it always possible to construct a corresponding translation scheme which satisfies these three conditions.

## Eliminating left recursion from a translation scheme:

Let us take a left recursive translation schema:

$A \rightarrow A_1 Y \{A.a = g(A_1.a, Y.y)\}$

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

In this grammar, each grammar symbol has a synthesized attributes written using their corresponding lowercase letter.

Now,

Eliminating left recursion as,

$A \rightarrow XR$

$R \rightarrow YR_1$

$R \rightarrow \epsilon$

Now,

Taking the semantic action for each symbol as follows:

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

$R \rightarrow Y\{R_1.inh = g(R.inh, Y.y)\}\ R_1\ \{R.syn = R_1.syn\}$

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

Example,

$A.a = g(g(f(X.x), Y_1.y), Y_2.y)$

$A.a = g(f(X.x), Y.y)$

$y_2$

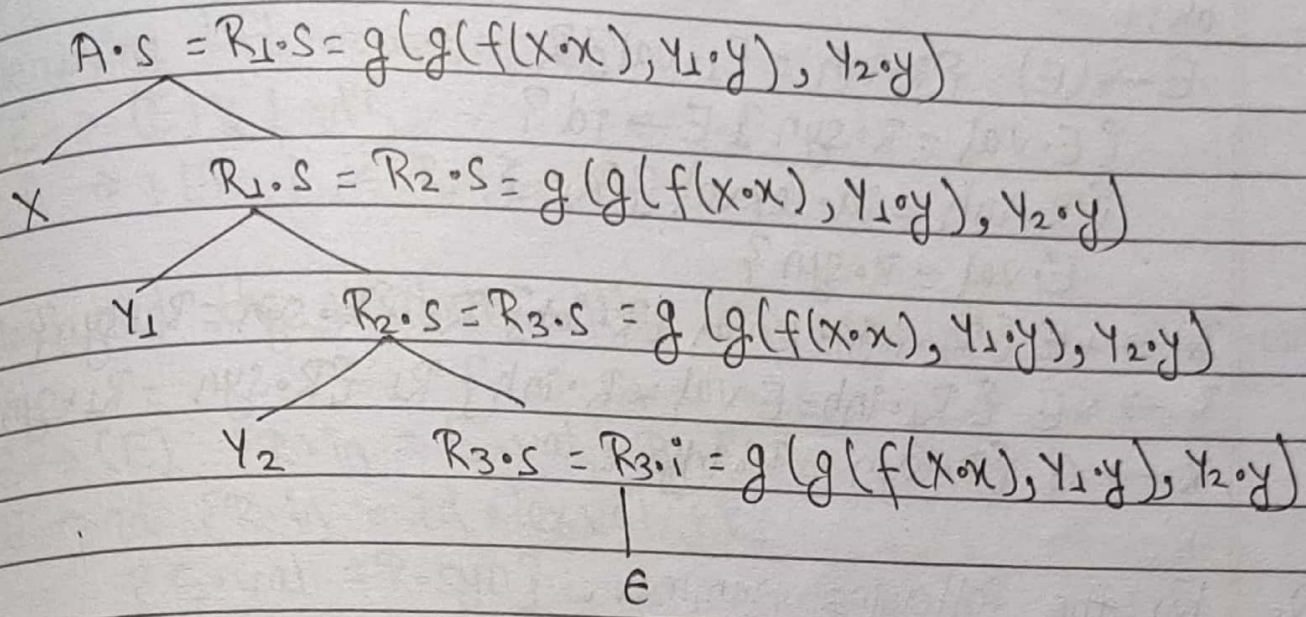
$A.a = f(X.x)$

$y_1$

$X$

Values are computed using left recursive grammar.

When the values are computed non-left recursive grammar for string XYY.




Q. For the following grammar,

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

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

Sol:

First part:

$E \rightarrow E+E \quad \{E.val = E_1.val + E_2.val\}$

$E \rightarrow E*E \quad \{E.val = E_1.val * E_2.val\}$

$E \rightarrow (E) \quad \{E.val = E_1.val\}$

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

Second Part:

Remove left recursion as,

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

$R \rightarrow +ER_1 \mid *ER_1 \mid \epsilon$

Scanned with CamScanner

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

$E \rightarrow (E)$ $\{R.inh = E_1.val\}$ $R$
$\{E.val = R.syn\}$ $E \rightarrow id$ $\{$
$\{R.inh = id.lexval\}$ $R_1$ $\{$
$E.val = R.syn\}$
$R \rightarrow +E$ $\{R_1.inh = E.val + R.inh\}$ $R_1$ $\{R.syn = R_1.syn\}$
$R \rightarrow *E$ $\{R_1.inh = E.val * R.inh\}$ $R_1$ $\{R.syn = R_1.syn\}$
$R \rightarrow \epsilon$ $\{R.syn = R.inh\}$

Q. For the following grammar.
$E \rightarrow E+E \mid E-E \mid T$
$T \rightarrow (E) \mid num$
Annotate the grammar with SDD using synthesized attribute. Remove the left recursion from the grammar and rewrites the attributes correspondingly.

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 annotate parse tree for the expression $2*(3+5)$ using the modified grammar.
Trace the path in which semantic attributes are evaluation.

Sol:

$E \rightarrow E+E$ $\{E.val = E_1.val + E_2.val\}$
$E \rightarrow E*E$ $\{E.val = E_1.val$ $E_2.val\}$

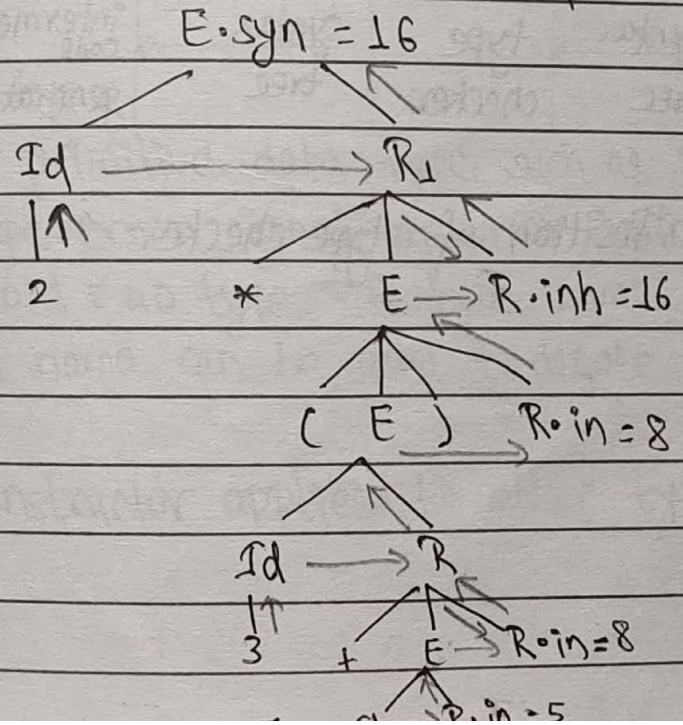
Scanned with CamScanner

$E \to (E)$ $\{E.val = E_1.val\}$
$E \to id$ $\{E.val = id.lexval\}$

Removing left recursion as,
$E \to (E)R \mid idR$
$R \to +ER_1 \mid *ER_1 \mid \epsilon$

Now, add the attribute within this non-left recursive grammar as,
$E \to (E)$ $\{R.in = E_1.val\}$ $R$ $\{E.val = R.syn\}$
$E \to id$ $\{R.in = id.lexval\}$ $R_1$
$\{E.val = R.syn\}$
$R \to +E$ $\{R_1.in = E.val + R.in\}$ $R_1$ $\{R.syn = R_1.syn\}$
$R \to *E$ $\{R_1.in = E.val * R.in\}$ $R_1$ $\{$
$R.syn = R_1.syn\}$
$R \to \epsilon$ $\{R.syn = R.in\}$

Second Part, Annotate parse tree for $2*(3+5)$ is,



## 2.4) Type Checking

- A type checker verifies that the type of a construct matches that expected by its context. For example, the built-in arithmetic operator mod in Pascal requires integer operands, so a type checker must verify that the operands of mod have type integer.
- Type checking is the process of verifying that each operation execution in a program respects the type system of the language.
- This generally means that all operands in any expression are of appropriate types and number.
- Much of what we do in the semantic analysis phase is type checking.

### Position of Type Checker

| | | | | | | |
|---|---|---|---|---|---|---|
| token stream → | parser | syntax tree → | type checker | syntax tree → | intermediate code generator | intermediate representation → |

Fig: Position of type checker

Scanned with CamScanner

## Designing a Type Checker:

When designing a type checker for a compiler, here's the process:

1. Identify the types that are available in the language
2. Identify the language constructs that have types associated with them.
3. Identify the semantic rules for the language.

- If a problem is found, eg: one tries to add a char to a double in C, we encounter a type error.
- A language is considered strongly-typed if each and every type error is detected during compilation.
- Type checking can be done compilation, during execution, or divided across both.

## Type Expression:

The type of a language construct is denoted by a type expression. A type expression can be:

- A basic type
  - a primitive data type such as integer, real, char, boolean,....
  - type-error signal - an error during type checking
  - void: no type - A type name
  - a name can be used to denote a type expression.

- A type constructor applies to other type expressions.

Scanned with CamScanner

- arrays: If T is a type expression, then array (I, T) is a type expression where I denotes index range.
  Ex: array (0...99, int)

- products: If $T_1$ and $T_2$ are type expressions, then their cartesian product $T_1 \times T_2$ is a type expression.
  Ex: int x int.

- pointers: If T is a type expression, then pointer (T) is a type expression.
  Ex: pointer(int)

- functions: We may treat functions in a programming language as mapping from a domain type D to a range type R. So, the type of a function can be denoted by the type expression $D \rightarrow R$ where D are R type expressions.
  Ex: int → int represents the type of a function which takes an int value as parameter, and its return type is also int.

## Type System:

- A type system defines a set of types and rules to assign types to programming language constructs.
- Informal type system rules, for example "if both operands of addition are of type integer, then the result is of type integer"
- A type checker implements type system
- A sound type system eliminates run-time type checking for type errors.

Scanned with CamScanner

# Static versus Dynamic Type Checking

## Static Type Checking:

- Static type checking is done at compile-time. The information the type checker needs is obtained via declarations and stored in a master symbol table.
- Typical examples of static checking are:
  - Type checks
  - Flow-of-control checks
  - Uniqueness checks
  - Name-related checks

## Dynamic Type Checking:

- Dynamic type checking is implemented by including type information for each data location at run time.
- For example, a variable of type double would contain both the actual double value and some kind of tag indicating "double type".
- The execution of any operation begins by first checking these type tags. The operation is performed only if everything checks out. Otherwise, a type error occurs and usually halts execution.
- A programming language is strongly-typed, if every program its compiler accepts will execute without type errors.
- In practice, some of type checking operations are done at run-time (so, most of the programming languages are not strongly-typed).
- Ex: int x[100]; x[i] most of the compilers cannot guarantee that i will be between 0 and 99.

Scanned with CamScanner

## Type Conversion and Coercion :

### Type Conversion :

- A language may have operators for converting a value of one type to a value of another type, eg: an integer to a floating point number. Some-times these operators are explicit in the program and hence easy to check.
- However, many languages allow implicit conversion of integers to floats, such that, for example, 3+3.12 is well-typed with the implicit assumption that the integer 3 is converted to a float before the addition.

### Coercion :

- Type conversions which happen implicitly is called coercion.
- Implicit type conversions are carried out by the compiler recognizing a type incompatibility and running a type conversion routine (for example, something like inttoreal(int)) that takes a value of the original type and returns a value of the required type.

### Where type checking is needed?

- Assignments: When a variable is given a value by an assignment, it must be verified that the type of the value is the same as the declared type of the variable.

- Overloading: The same name is used for several different operations over several different types.

Scanned with CamScanner

- Polymorphism / Generic types: Some languages allow a function to be poly-morphic or generic, that is, to be defined over a large class of similar types, eg: over all arrays no matter what the types of the elements are.

- Data structures: A data structure may define a value with several components (eg: a struct, tuple or record), or a value that may be of different types at different times (eg: a union, variant or sum). To type-check such structures, the type checker must be able to represent their types.

- The type checker may need a data structure that describes complex types.

## A Simple Type Checking System

Example Simple Language

$P \rightarrow D;\ S$ Pointer to T

$D \rightarrow D;\ D \mid id : T$

$T \rightarrow$ boolean | char | integer | array [num] of T | ↑T

$S \rightarrow id = E \mid$ if E then S | while E do S | S; S

$E \rightarrow$ true | false | literal | num | id | E and E | E+E | E[E] | E↑

- This grammar generate the language with declarations followed by statements like

```
key : integer;
key = 199;
if true then key = 100;
```

Example Simple Translation scheme

$D \rightarrow id : T$ {addtype (id.entry, T.type)}

$T \rightarrow boolean$ { T.type = boolean }

$T \rightarrow char$ { T.type = char }

$T \rightarrow integer$ { T.type = integer }

$T \rightarrow array [num] of T_1$ { T.type = array(L.num.val, $T_1$.type) }

$T \rightarrow \uparrow T_1$ { T.type = pointer (T) }

Example Type Checking of Expressions

$E \rightarrow id$ {E.type = lookup (id.entry) }

$E \rightarrow charliteral$ {E.type = char }

$E \rightarrow intliteral$ { E.type = int }

$E \rightarrow E_1 + E_2$ { E.type = ($E_1$.type == $E_2$.type )? $E_1$.type : type_error

$E \rightarrow E_1 [E_2]$ { E. type = ($E_2$.type == int and $E_1$.type == array (s,t)) ? t : type_error }

$E \rightarrow E_1 \uparrow$ { E.type = ( $E_1$.type == pointer(t) ) ? t : type_error }

Example Type Checking of Statements

- In some languages statements have a type associated with them, while some other languages don't assign types to statements. In the latter case, statements are given a type void to distinguish a type safe statement with one which has a type error.

$S \rightarrow id = E$ {S.type = (id.type == E.type)? void : type_error}

Note: the type of id is determined by: id.type = lookup (id.entry)

$S \rightarrow$ if E then $S_1$ { S.type = ( E.type == boolean)? $S_1$.type : type_error }

$S \rightarrow$ while E do $S_1$ { S.type = ( E.type == boolean)? $S_1$.type : type_error }

$S \rightarrow S_1 ; S_2$ { S.type = ( $S_1$.type == void and $S_2$.type == void)? void : type_error }

Scanned with CamScanner

## Example Type Checking of Functions

- A function to an argument can be captured by production:

$T \rightarrow T \rightarrow T$ — Function type declaration

$E \rightarrow E(E)$ — Function call

Example:-

```
v : integer;
odd : integer → boolean;
if odd(3) then
    v := 1;
```

$T \rightarrow T_1 \rightarrow T_2$ $\{T.type = function(T_1.type, T_2.type)\}$

$E \rightarrow E_1(E_2)$ $\{E.type = (E_1.type == function(s,t) \text{ and } E_2.type == s)?$
$t : type_error\}$

## Structural Equivalence of Type Expressions:

- How to know that two type expressions are equal? As long as type expressions are built from basic types (no type names), we may use structural equivalence between two type expressions.

### Structural Equivalence Algorithm:

```
sequiv(s,t) : boolean;
  if (s and t are same basic types) then return true
else if (s = array(s1,s2) and t = array(t1,t2)) then return (sequiv(s1,t1)
        and sequiv(s2,t2))
else if (s = s1 × s2 and t = t1 × t2) then return (sequiv(s1,t1) and sequiv(s2,t2))
elseif (s = pointer(s1) and t = pointer(t1)) then return (sequiv(s1,t1))
elseif (s = s1 → s2 and t = t1 → t2) then return (sequiv(s1,t1) and sequiv(s2,t2))
else return false
```

Scanned with CamScanner

## Syntax-Directed Definitions for Type Checking in Yacc

```
%{
 enum Types {Tint, Tfloat, Tpointer, Tarray, ----};
 typedef struct Type
  {
    enum Types type;
    struct Type *child; // at most one type parameter
  } Type;
%}
%union
```

Scanned with CamScanner

# UNIT-4

## Code Generation and Optimization

### Introduction

- The final phase of a compiler is code generator
- It receives an intermediate representation (IR) with supplementary information in symbol table
- Produces a semantically equivalent target program
- Code generator main tasks:
  - Instruction selection
  - Register allocation and assignment
  - Instruction ordering

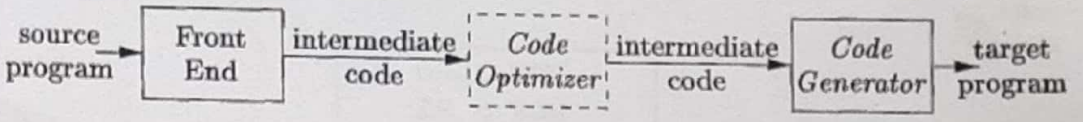



Fig: position of Code Generator

### Code Generation Design Issue

The most important criterion is that it produces correct code

1. *Input to the code generator*
   The input to the code generator is intermediate representation together with the information in the symbol table. What type of input postfix, three-address, dag or tree?

2. *The target program*
   Which one is the output of code generator: Absolute machine code (executable code), Relocatable machine code (object files for linker), Assembly language (facilitates debugging), Byte code forms for interpreters (e.g. JVM)

3. *Instruction Selection*
   The code generator must map the IR program into a code sequence that can be executed by the target machine. The complexity of performing this mapping is determined by a factors such as
   - the level of the IR
   - the nature of the instruction-set architecture
   - the desired quality of the generated code.

4. *Register Allocation*



Scanned with CamScanner

A key problem in code generation is deciding what values to hold in what registers. The use of registers is often subdivided into two sub problems:

- Register allocation, during which we select the set of variables that will reside in registers at each point in the program.
- Register assignment, during which we pick the specific register that a variable will reside in.

5. *Evaluation Order*
   The order in which computations are performed can affect the efficiency of the target code.

## Basic Blocks

A *basic block* is a sequence of consecutive instructions in which flow of control enters by one entry point.

```
     MOV 1,R0
     MOV n,R1
     JMP L2
L1:  MUL 2,R0
     SUB 1,R1
L2:  JMPNZ R1,L1
```

```
     MOV 1,R0
     MOV n,R1
     JMP L2
```

```
L1:  MUL 2,R0
     SUB 1,R1
```

```
L2:  JMPNZ R1,L1
```

## Basic Block Construction Algorithm

*Input*: A sequence of three-address statements
*Output*: A list of basic blocks with each three-address statement in exactly one block

1. Determine the set of *leaders*, the first statements if basic blocks
   a. The first statement is the leader
   b. Any statement that is the target of a conditional or goto is a leader
   c. Any statement that immediately follows conditional or goto is a leader
2. For each leader, its basic block consist of the leader and all statements up to but not including the next leader or the end of the program

## Flow Graphs

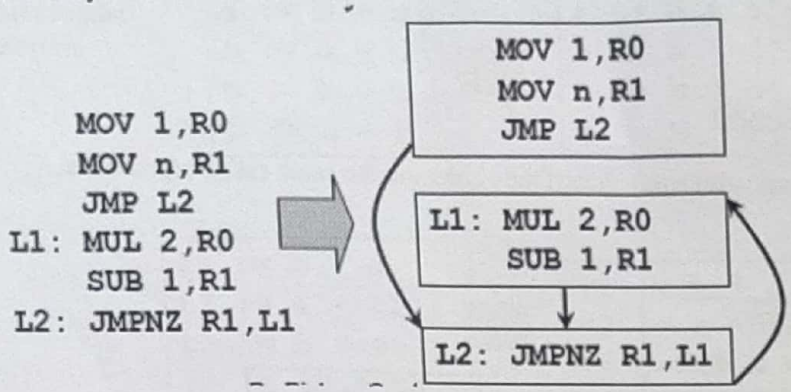
A *flow graph* is a graphical depiction of a sequence of instructions with control flow edges.

A flow graph can be defined at the intermediate code level or target code level.



Scanned with CamScanner

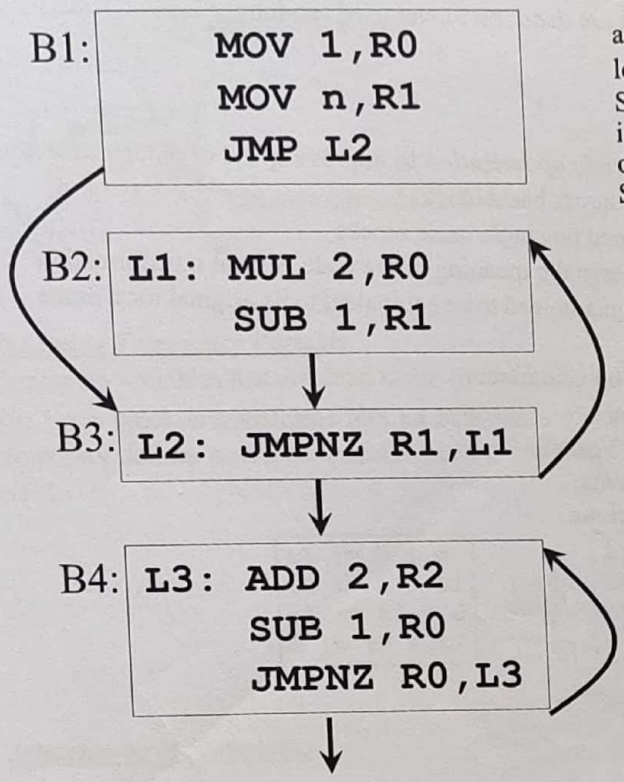
The nodes of flow graphs are the basic blocks and flow-of-control to immediately follow node connected by directed arrow.




## Loops

A *loop* is a collection of basic blocks, such that

- All blocks in the collection are *strongly connected*




- The collection has a unique *entry*, and the only way to reach a block in the loop is through the entry

Strongly connected components: there is path of length of one or more from one node to another to make a cycle. Such as {B2,B3}, {B4}

Entries: B3, B4

*A loop that consist no other loop is called inner loop*



Scanned with CamScanner

## Equivalence of Basic Blocks

Two basic blocks are (semantically) *equivalent* if they compute the same set of expressions

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

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

Transformation

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

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

Blocks are equivalent, assuming **t1** and **t2** are *dead*: no longer used (no longer *live*)

## Optimization of Basic Blocks

A *code-improving transformation* is a ***code optimization*** to improve speed or reduce code size
*Global transformations* are performed across basic blocks
*Local transformations* are only performed on single basic blocks
Transformations must be safe and preserve the meaning of the code A local transformation is safe if the transformed basic block is guaranteed to be equivalent to its original form Some local transformation are:

- Common-Subexpression Elimination
- Dead Code Elimination
- Renaming Temporary Variables
- Interchange of Statements
- Algebraic Transformations

***Common Subexpression Elimination***

Remove redundant computations



Scanned with CamScanner

Look at 2nd and 4th: compute same expression

```
a := b + c
b := a - d
c := b + c
d := a - d
```

```
a := b + c
b := a - d
c := b + c
d := b
```

Look at 1st and 3rd : b is redefine in 2nd therefore different in 3rd, not the same expression

```
t1 := b * c
t2 := a - t1
t3 := b * c
t4 := t2 + t3
```

```
t1 := b * c
t2 := a - t1
t4 := t2 + t1
```

***Dead Code Elimination***

Remove unused statements

```
b := a + 1
a := b + c
...
```

```
b := a + 1
...
```

Assuming a is *dead* (not used)

```
goto L2
b := x + y
...
```

Remove unreachable code

***Renaming Temporary Variable***

Temporary variables that are dead at the end of a block can be safely renamed

The basic block is transforms into an equivalent block in which each statement that defines a temporary defines a new temporary. Such a basic block is called *normal-form block* or *simple block*.

```
t1 := b + c
t2 := a - t1
t1 := t1 * d
d := t2 + t1
```

```
t1 := b + c
t2 := a - t1
t3 := t1 * d
d := t2 + t3
```

Normal-form block

***Interchange of Statements***

Prepared by: RC

Scanned with CamScanner

Independent statements can be reordered without effecting the value of block to make its optimal use.

```
t1 := b + c
t2 := a - t1
t3 := t1 * d
d := t2 + t3
```

⇒

```
t1 := b + c
t3 := t1 * d
t2 := a - t1
d := t2 + t3
```

Note that normal-form blocks permit all statement interchanges that are possible

***Algebraic Transformation***

Change arithmetic operations to transform blocks to algebraic equivalent forms
Simplify expression or replace expensive expressions by cheaper ones.

```
t1 := a - a
t2 := b + t1
t3 := t2 **2
```

⇒

```
t1 := 0
t2 := b
t3 := t2 * t2
```

In statement 3, usually require a function call

Transforms to simple and equivalent statement

## Next-Use Information

Next-use information is needed for dead-code elimination and register assignment (if the name in a register is no longer needed, then the register can be assigned to some other name)

All non-temporary variables and temporary that is used across the block are considered live.

If *i*: $x = \ldots$ and *j*: $y = x + z$ are two statements *i* & *j*, then *next-use* of $x$ at *i* is *j*.

Next-use is computed by a backward scan of a basic block and performing the following actions on statement

```
i: x := y op z
```

- Add liveness/next-use info on $x$, $y$, and $z$ to statement *i* (whatever in the symbol table)
- Before going up to the previous statement (scan up):
  - Set $x$ info to "not live" and "no next use"
  - Set $y$ and $z$ info to "live" and the next uses of $y$ and $z$ to *i*

## A Simple Code Generator

Generates target code for a sequence of three-address statements using next-use information
Uses new function *getreg* to assign registers to variables
Computed results are kept in registers as long as possible, which means:



Scanned with CamScanner

- Result is needed in another computation
- Register is kept up to a procedure call or end of block

Checks if operands to three-address code are available in registers

## Code Generator Algorithm

For each statement $x := y$ op $z$

1. Set location $L = getreg(y, z)$ // to store the result of $y$ *op* $z$
2. If $y \notin L$ then generate //L is address descriptor --wait!
   **MOV** $y'$,$L$ //to place copy of y in L
   where $y'$ denotes one of the locations where the value of $y$ is available (choose register if possible)
3. Generate instruction
   **OP** $z'$,$L$
   where $z'$ is one of the locations of $z$;
   Update register/address descriptor of $x$ to include $L$
4. If $y$ and/or $z$ has no next use and is stored in register, update register descriptors to remove $y$ and/or $z$

## Register and Address Descriptors

A *register descriptor* keeps track of what is currently stored in a register at a particular point in the code, e.g. a local variable, argument, global variable, etc.

**MOV a,R0** "**R0** contains **a**"

An *address descriptor* keeps track of the location where the current value of the name can be found at run time, e.g. a register, stack location, memory address, etc.

**MOV a,R0**
**MOV R0,R1** "**a** in **R0** and **R1**"

## The getreg Algorithm

To compute getreg(y,z)

1. If $y$ is stored in a register $R$ and $R$ only holds the value $y$, and $y$ has no next use, then return $R$; Update address descriptor: value $y$ no longer in $R$
2. Else, return a new empty register if available
3. Else, find an occupied register $R$;
   Store contents (register spill) by generating
   **MOV** $R$,$M$
   for every $M$ in address descriptor of $y$;
   Return register $R$
4. If not used in the block or no suitable register return a memory location



Scanned with CamScanner

## Code Generation Example

Statement: d :=(a-b) + (a - c) + (a - c)

| Statements | Code Generated | Register Descriptor | Address Descriptor |
|---|---|---|---|
| t := a - b | MOV a,R0<br>SUB b,R0 | Registers empty<br>R0 contains t | t in R0 |
| u := a - c | MOV a,R1<br>SUB c,R1 | R0 contains t<br>R1 contains u | t in R0<br>u in R1 |
| v := t + u | ADD R1,R0 | R0 contains v<br>R1 contains u | u in R1<br>v in R0 |
| d := v + u | ADD R1,R0<br>MOV R0,d | R0 contains d | d in R0<br>d in R0 and memory |

## Peephole Optimization

- Peephole represents sequence of instructions with at most one entry point
- Statement-by-statement code generation often produce redundant instructions that can be optimize to save time and space requirement of target program.
- Examines a short sequence of target instructions in a window (*peephole*) and replaces the instructions by a faster and/or shorter sequence whenever possible.
- Applied to intermediate code or target code
- Typical optimizations:
  - Redundant instruction elimination
  - Elimination of Unreachable Code
  - Flow-of-control optimizations
  - Algebraic simplifications
  - Use of machine idioms

### *Redundant Instruction Elimination*

Consider

**MOV R0,a**
**MOV a,R0**

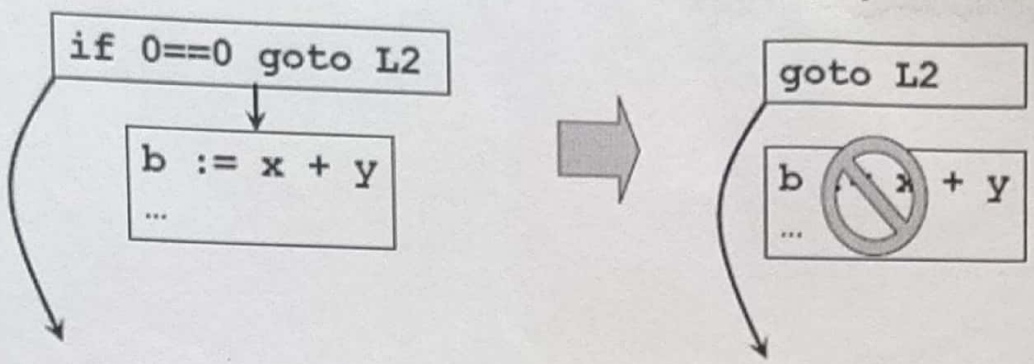
The second instruction can be deleted because first ensures value of a in R0, but only if it is not labeled with a target label

### *Eliminating Unreachable Code*



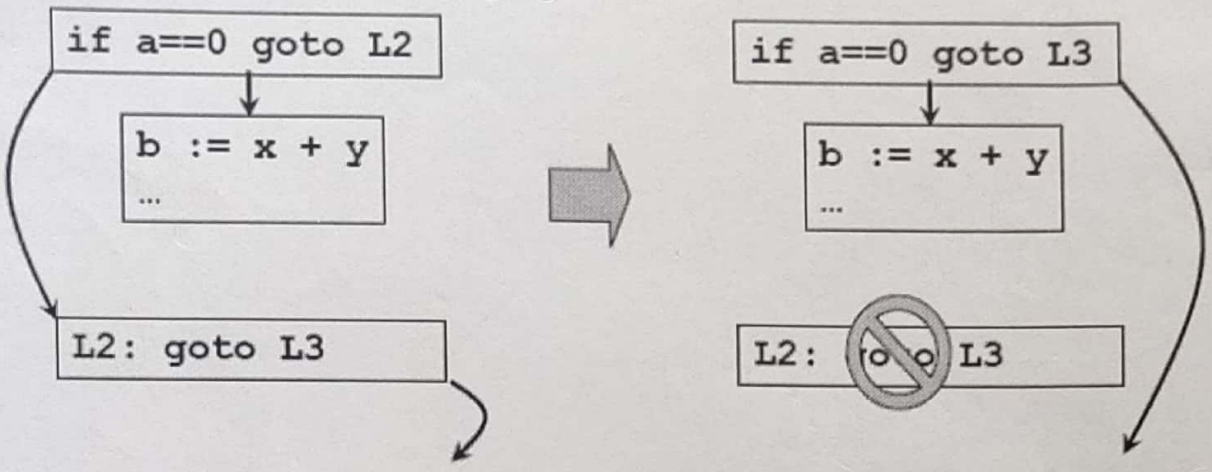
Scanned with CamScanner

An unlabeled instruction immediately following an unconditional jump can be removed

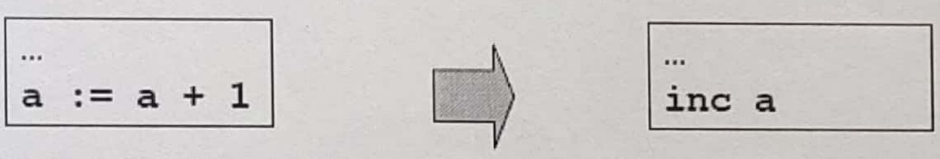



***Flow of Control Optimization (Branch Chaining)***

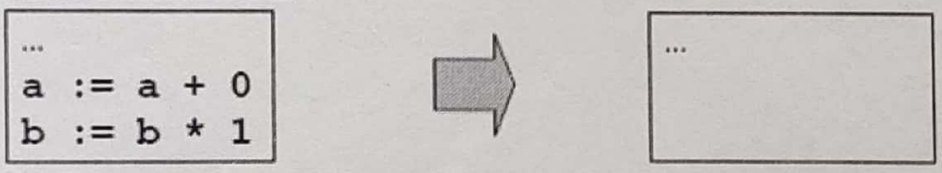
Shorten chain of branches by modifying target labels




***Utilize Machine Idioms***




***Algebraic Simplification***




Prepared by: RC

Scanned with CamScanner