SASURIE INSTITUTIONS
COIMBATORE | TIRUPPUR | VIJAYAMANGALAM

SASURIE COLLEGE OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COMPILER DESIGN NOTES

(E325A) COMPILER DESIGN

(Common to CSE, ECM)

Course Objectives:

Student will:

1. Understand the major concept areas of language translation and compiler design.
2. Enrich the knowledge in various phases of compiler ant its use, code optimization techniques, machine code generation, and use of symbol table.
3. Extend the knowledge of parser by parsing LL parser and LR parser.
4. Implement the concepts of semantic analysis using semantic rules.
5. Apply the knowledge of data flow analysis and object code generation.

UNIT – I:

Overview of Compilation:

Phases of Compilation – Lexical Analysis, Regular Grammar and regular expression for common programming language features, pass and Phases of translation, interpretation, bootstrapping, data structures in compilation – LEX lexical analyzer generator.

UNIT – II:

Top down Parsing: Context free grammars, Top down parsing – Backtracking, LL (1), recursive descent parsing, Predictive parsing, Preprocessing steps required for predictive parsing.

Bottom up parsing: Shift Reduce parsing, SLR,CLR and LALR parsing, Error recovery in parsing , handling ambiguous grammar, YACC –automatic parser generator.

UNIT – III:

Semantic analysis: Intermediate forms of source Programs – abstract syntax tree, polish notation and three address codes. Attributed grammars, Syntax directed translation, Conversion of popular Programming languages language Constructs into Intermediate code forms, Type checker.

UNIT – IV

Symbol Tables: Symbol table format, organization for block structures languages, hashing, tree structures representation of scope information. Block structures and non block structure storage allocation: static, Runtime stack and heap storage allocation, storage allocation for arrays, strings and records.

Code optimization: Consideration for Optimization, Scope of Optimization, local optimization, loop optimization, frequency reduction, folding, DAG representation.

UNIT – V:

Data flow analysis: Flow graph, data flow equation, global optimization, redundant sub expression elimination, Induction variable elements, Live variable analysis, Copy propagation.

Object code generation: Object code forms, machine dependent code optimization, register allocation and assignment generic code generation algorithms, DAG for register allocation.

TEXT BOOKS :

1. Principles of compiler design -A.V. Aho . J.D.Ullman; Pearson Education.
2. Modern Compiler Implementation in C- Andrew N. Appel, Cambridge University Press.

REFERENCES BOOKS:

1. lex &yacc – John R. Levine, Tony Mason, Doug Brown, O'reilly
2. Modern Compiler Design- Dick Grune, Henry E. Bal, Cariel T. H. Jacobs, Wiley dreamtech.
3. Engineering a Compiler-Cooper & Linda, Elsevier.

Course Outcomes:

At the end of course student will able to:

1. Design a Lexical Analyzer.
2. Compare different types of parsing techniques
3. Describe the concepts of Semantic analysis and type checking.
4. Derive a Intermediate code from source code.
5. Apply different code optimization and code generation techniques.

Compiler Design: Introduction

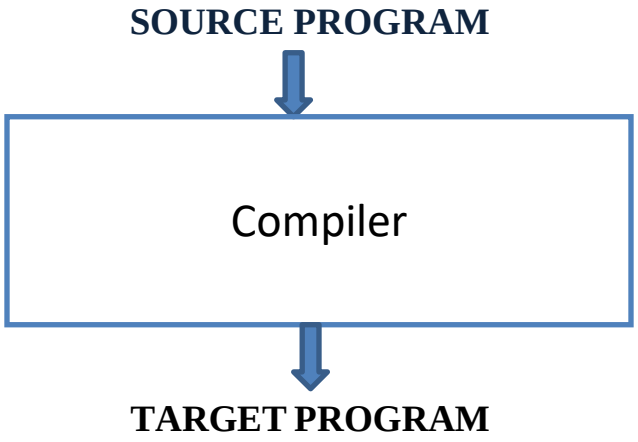
UNIT-1: Mind Map Syllabus

Overview of Compilation:

Phases of Compilation – Lexical Analysis, Regular Grammar and regular expression for common programming language features, pass and Phases of translation, interpretation, bootstrapping, data structures in compilation – LEX lexical analyzer generator.

Compiler:

- It is a software which converts a program written in high level language (Source Language) to low level language (Object/Target/Machine Language).
- The source code is translated to object code successfully if it is free of errors.
- The compiler specifies the errors at the end of compilation with line numbers when there are any errors in the source code.



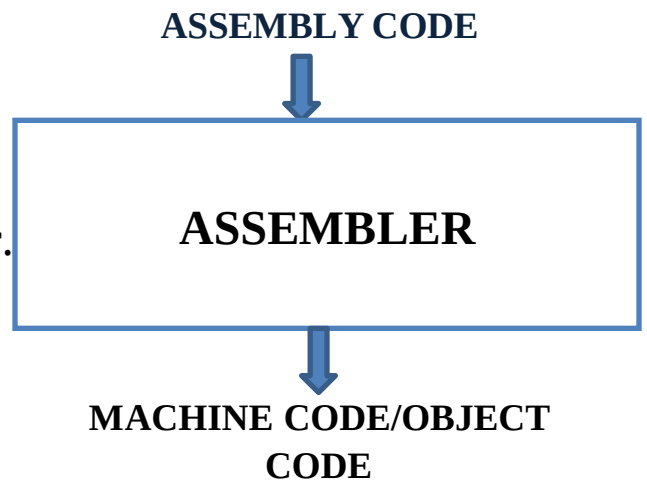
Introduction...

Interpreter :

- The translation of single statement of source program into machine code is done by language processor and executes it immediately before moving on to the next line is called an interpreter.
- If there is an error in the statement, the interpreter terminates its translating process at that statement and displays an error message.
- The interpreter moves on to the next line for execution only after removal of the error.
- An Interpreter directly executes instructions written in a programming or scripting language without previously converting them to an object code or machine code.

Assembler:

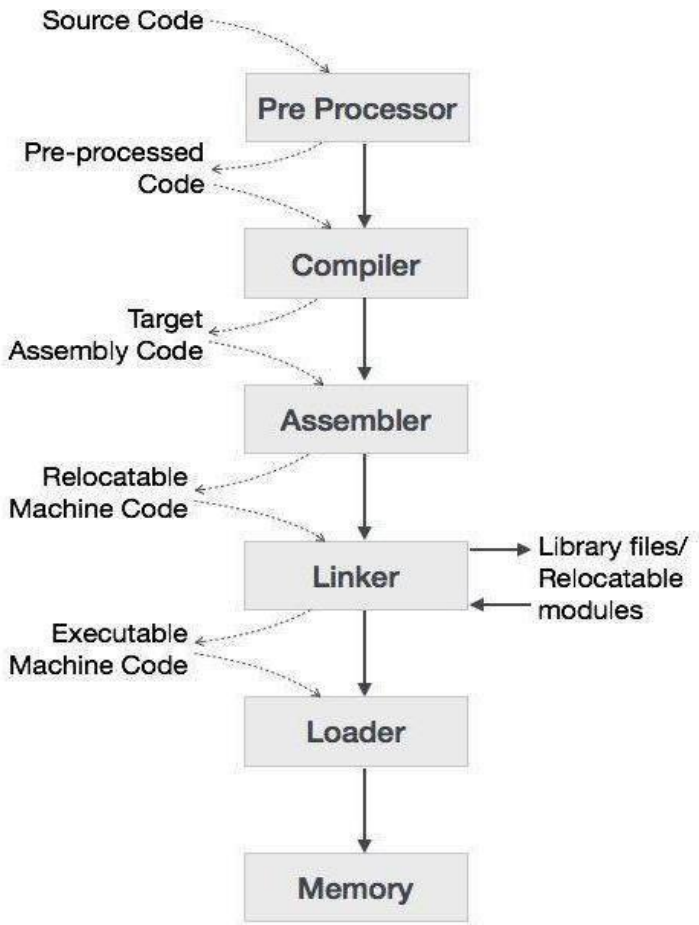
- The Assembler is used to translate the program written in Assembly language into machine code.
- The source program is a input of assembler that contains assembly language instructions.
- The output generated by assembler is the object code or machine code understandable by the computer.



Language Processing System

- we write programs in high-level language, which is easier for us to understand and remember. These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine. This is known as **Language Processing System.**
- **Steps Involved in language processing:**

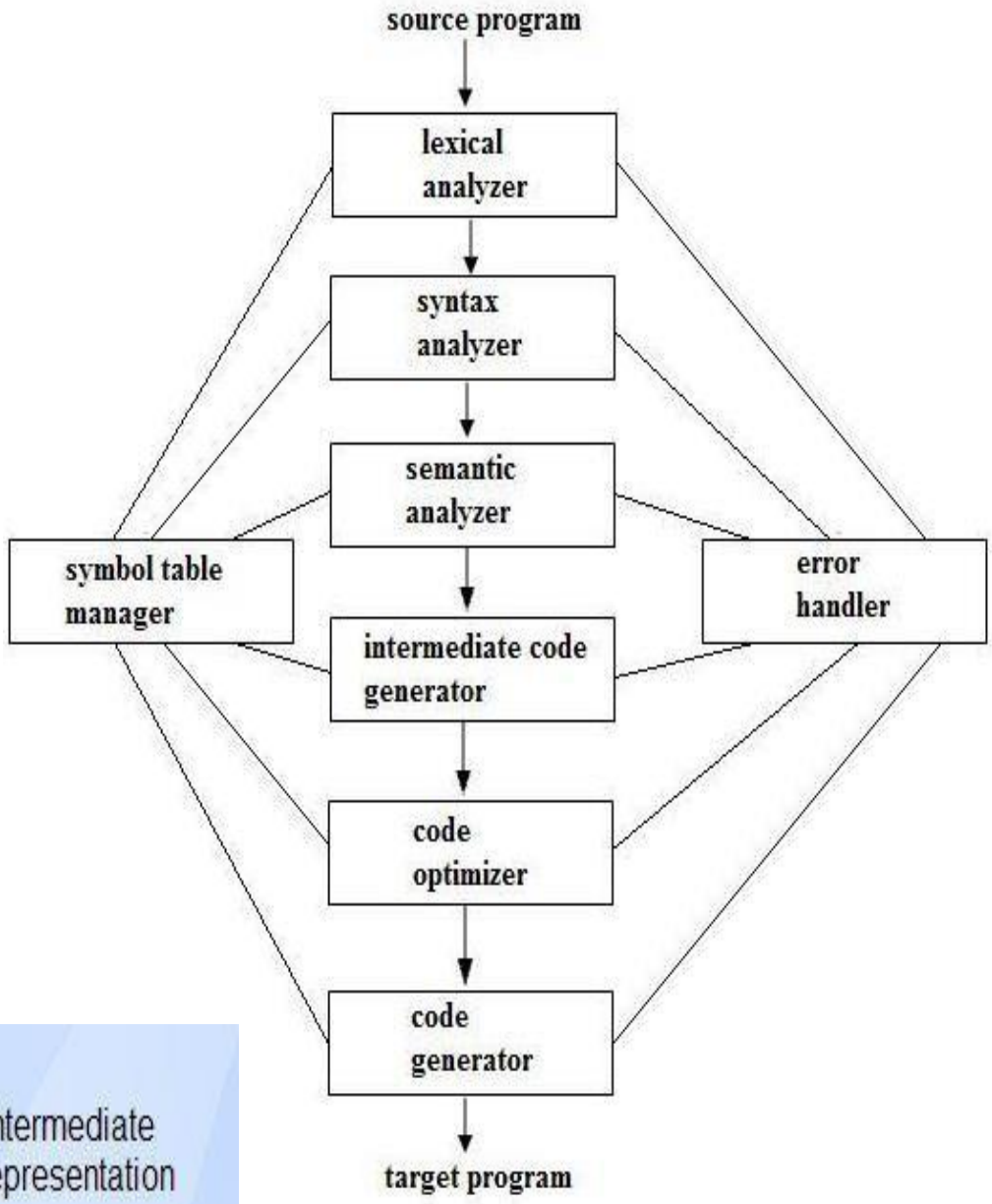
1. User writes a program in high-level language(Source code).
2. A preprocessor, generally considered as a part of compiler, is a tool that produces input for compilers. It deals with macro-processing, augmentation, file inclusion, language extension, etc.
3. The compiler, compiles the program and translates it to assembly program (low-level language).
4. An assembler then translates the assembly program into machine code (object).
5. A linker tool is used to link all the parts of the program together for execution (executable machine code).
6. A loader loads all of them into memory and then the program is executed.



Phases of Compilation

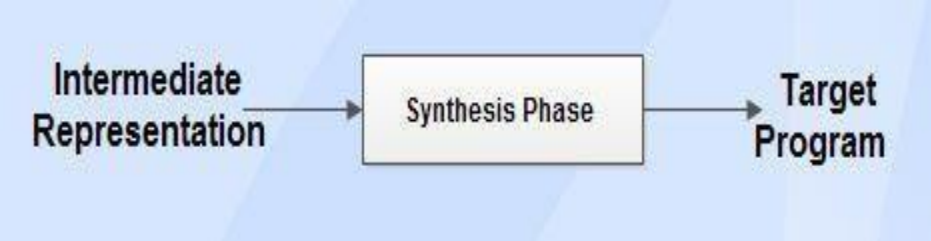
Compiler Phases:

- The compilation process contains the sequence of various phases. Each phase takes source program in one representation and produces output in another representation.
- Each phase takes input from its previous stage.
- There are two parts of Compilation:
  I. Analysis (Front End)
  II. Synthesis (Back End)
- The Analysis part breaks the source program into constituent pieces and creates an intermediate representation of source program.



Phases of Compilation...

- The Synthesis part Construct the desired target program from the Intermediate representation.



- **The different phases of compiler are:**
  1. Lexical analysis
  2. Syntax analysis
  3. Semantic analysis
  4. Intermediate code generator
  5. Code optimizer
  6. Code generator
- All of the above mentioned phases involve the following tasks:
  - Symbol table management.
  - Error handling.

Phases of Compilation...

Lexical Analysis:

- Lexical analysis is the first phase of compiler which is also termed as **scanning.**
- Source program is scanned to read the stream of characters and those characters are grouped to form a sequence called **lexemes** which produces token as output.
- **Token:** Token is a sequence of characters that represent lexical unit, which matches with the pattern, such as keywords, operators, identifiers etc.
- **Lexeme:** Lexeme is instance of a token i.e., group of characters forming a token.
  Example: Pi=3.14,here string Pi is a lexeme for the token "identifier"
- **Pattern:** Pattern describes the rule that the lexemes of a token takes. It is the structure that must be matched by strings.
- Once a token is generated the corresponding entry is made in the symbol table.
- Example **: c=a+b*5**

Lexemes	Tokens
c	Identifier id1
=	assignment symbol
a	Identifier id2
+	+ (addition symbol)
b	Identifier id3
*	* (multiplication symbol)
5	5 (number)

- Output of LA is **<id1>=< id2> +<id3 > * 5**

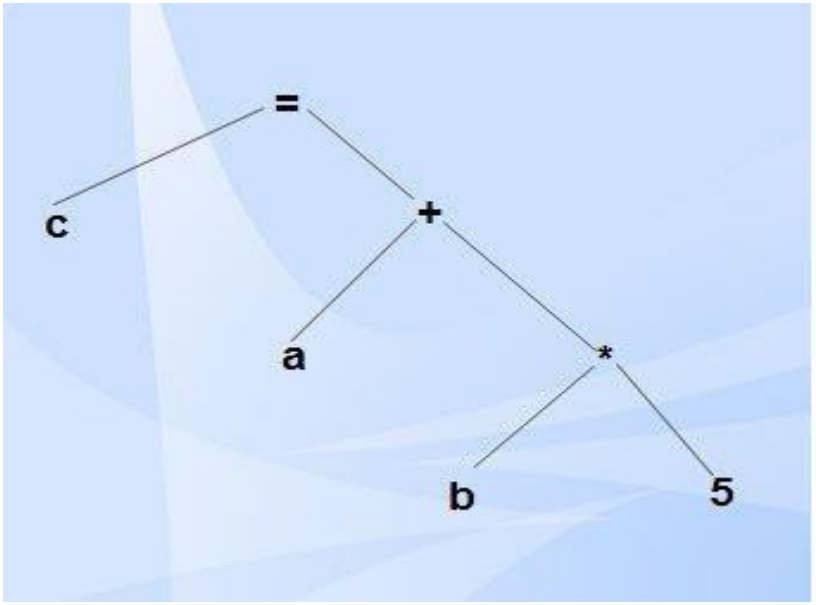
Phases of Compilation...

Syntax Analysis:

- Syntax analysis is the second phase of compiler which is also called as **parsing.**
- Parser converts the tokens produced by lexical analyzer into a tree like representation called **parse tree**.
- A parse tree describes the syntactic structure of the input.
- Syntax tree is a compressed representation of the parse tree in which the **operators** appear as **interior nodes** and the **operands** of the operator are the **children of the node for that operator**.

Input: Tokens

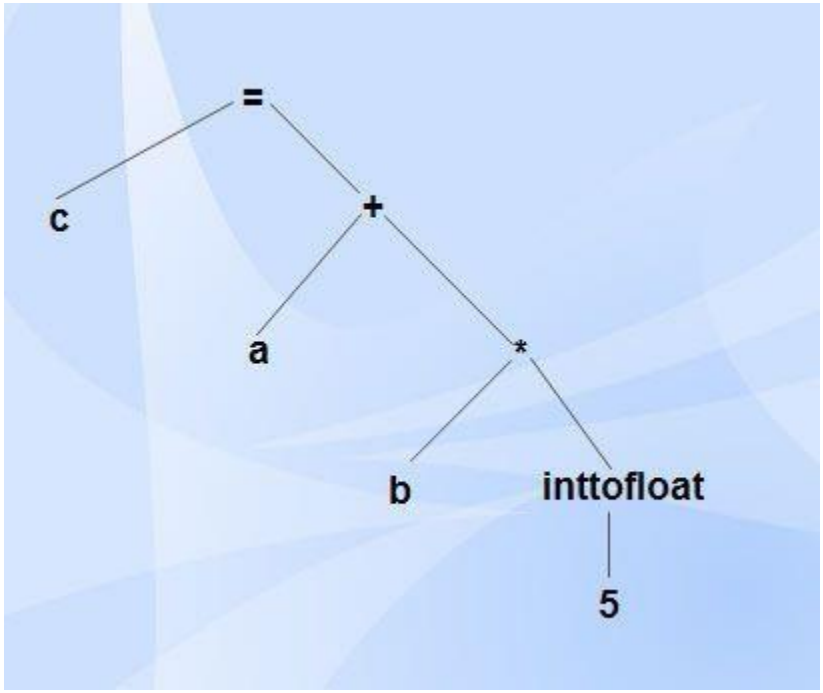
Output: Parse Tree



Phases of Compilation...

Semantic Analysis:

- Semantic analysis is the third phase of compiler.
- It checks for the semantic consistency.
- Type information is gathered and stored in symbol table or in syntax tree.
- Performs type checking.



Phases of Compilation...

Intermediate Code Generation:

Intermediate code generation produces intermediate representations for the source program which are of the following forms:

- Postfix notation
- Three address code
- Syntax tree

- Most commonly used form is the three address code.

t_1 = **inttofloat (5)**

$t_2 = id_3 * t_1$

$t_3 = id_2 + t_2$

$id_1 = t_3$

- **Three address code** is a type of intermediate **code** which is easy to generate and can be easily converted to machine **code**.
- It makes use of at most **three** addresses or operands and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by **compiler**.

Phases of Compilation...

Code Optimization:

- Code optimization phase gets the intermediate code as input and produces optimized intermediate code as output.
- It can be done by reducing the number of lines of code for a program.
- During the code optimization, the result of the program is not affected.

$$t_1 = id_3 * 5.0$$

$$id_1 = id_2 + t_1$$

Code Generation:

- Code generation is the final phase of a compiler.
- It gets input from code optimization phase and produces the target code /object code as result.
- Intermediate instructions are translated into a sequence of machine instructions or assembly code that perform the same task.

LDF R_2, id_3
MULF R_2, #5.0
LDF R_1, id_2
ADDF R_1, R_2
STF id_1, R_1

Phases of Compilation...

Symbol Table Management:

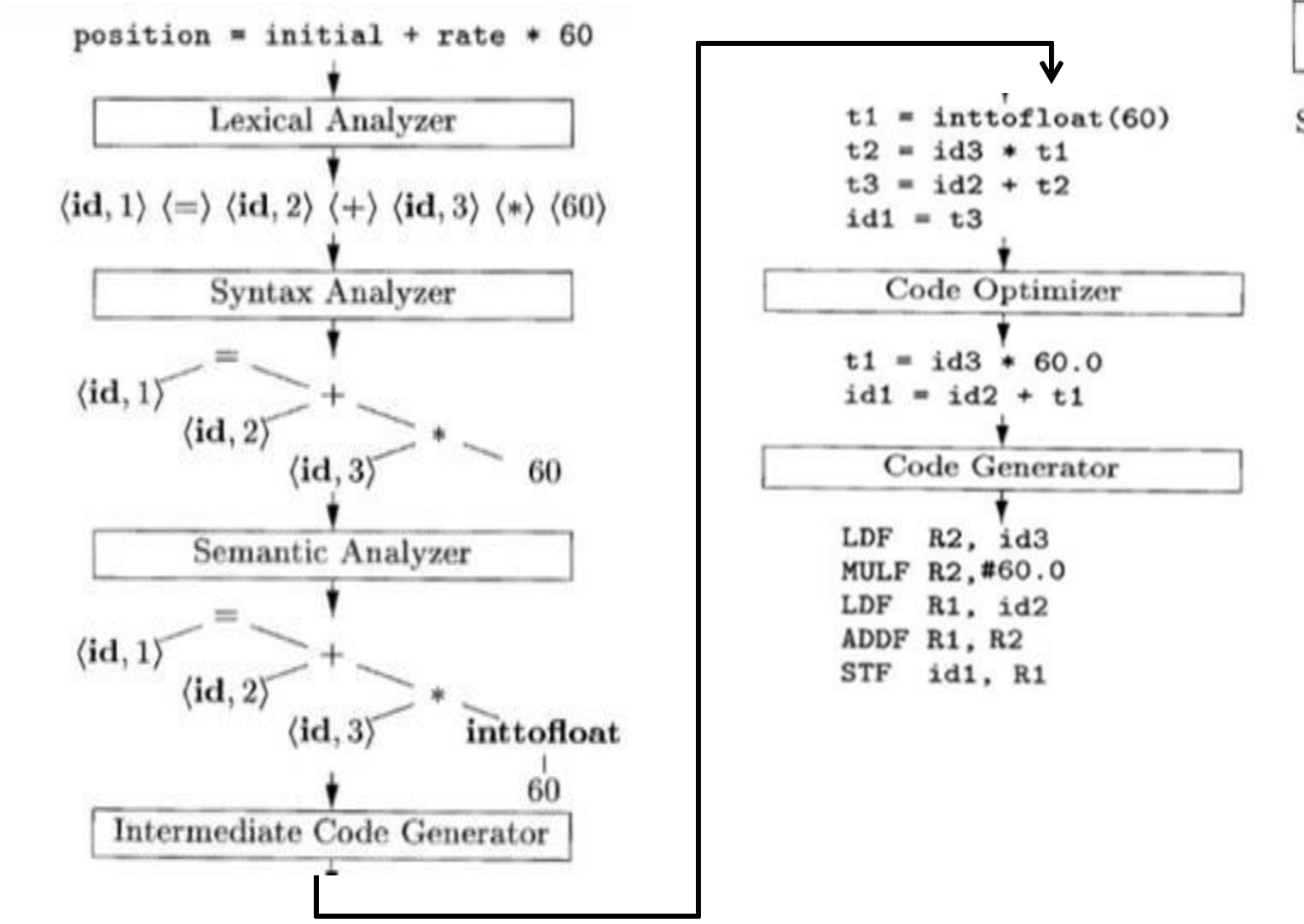
- Symbol table is used to store all the information about identifiers used in the program.
- It is a data structure containing a record for each identifier, with fields for the attributes of the identifier.
- It allows finding the record for each identifier quickly and to store or retrieve data from that record.
- Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

Error Handling:

- Each phase can encounter errors. After detecting an error, a phase must handle the error so that compilation can proceed.
- In lexical analysis, errors occur in separation of tokens.
- In syntax analysis, errors occur during construction of syntax tree.
- In semantic analysis, errors may occur at the following cases:
  - (i) When the compiler detects constructs that have right syntactic structure but no meaning
  - (ii) During type conversion.

Phases of Compilation

Example 1: Write the output for all the phases of compiler.

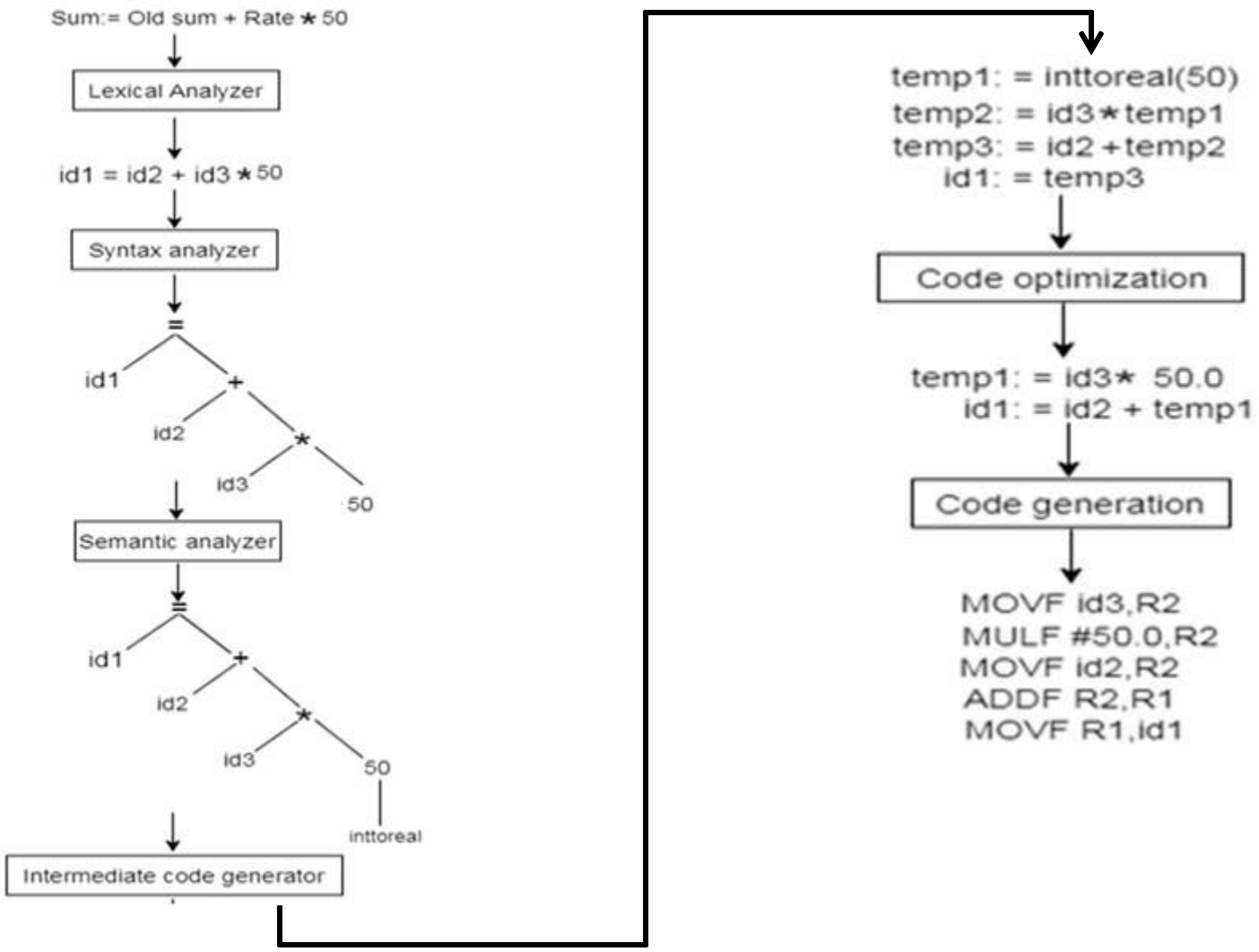


1	position	...
2	initial	...
3	rate	...

SYMBOL TABLE

Phases of Compilation

Example 2:



Regular Expressions& Regular Grammars

- The lexical analyzer needs to scan and identify only a finite set of valid string/token/lexeme that belong to the language .
- It searches for the pattern defined by the language rules.
- Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols.
- The grammar defined by regular expressions is known as **regular grammar**. The language defined by regular grammar is known as **regular language**.
- **Representing valid tokens of a language in regular expression**

  If x is a regular expression, then:
  - x* means zero or more occurrence of x. i.e., it can generate { e, x, xx, xxx, xxxx, … }
  - x+ means one or more occurrence of x. i.e., it can generate { x, xx, xxx, xxxx … } or x.x*
  - x? means at most one occurrence of x i.e., it can generate either {x} or {e}.
  - [a-z] is all lower-case alphabets of English language.
  - [A-Z] is all upper-case alphabets of English language.
  - [0-9] is all natural digits used in mathematics.
- **Representing occurrence of symbols using regular expressions**
  - letter = [a – z] or [A – Z]
  - digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 or [0-9]
  - sign = [+ | -]

Regular Expressions& Regular Grammars...

- **Representing language tokens using regular expressions**

  Decimal = $(sign)^{?}(digit)^{+}$

  Identifier = (letter)(letter | digit)*

- The only problem left with the lexical analyzer is how to verify the **validity of a regular expression** used in specifying the patterns of keywords of a language. A well-accepted solution is to use **finite automata for verification**.

Finite automata:

- Finite Automata(FA) is the simplest machine to recognize patterns.
- The finite automata or finite state machine is an abstract machine which have five elements or tuple .
- It has a set of states and rules for moving from one state to another but it depends upon the applied input symbol. Basically it is an abstract model of digital computer.
- A **Finite Automata** is a 5-tuple Machine **M={ Q, Σ, q, F, δ }** :

  Q : Finite set of states.

  Σ : set of Input Symbols.

  q : Initial state.

  F : set of Final States.

  δ : Transition Function.

Finite Automata...

FA is characterized into two types:

1) Deterministic Finite Automata (DFA)
2) Nondeterministic Finite Automata (NFA)

Deterministic Finite Automata (DFA) :

- DFA refers to Deterministic Finite Automaton.
- A Finite Automata(FA) is said to be deterministic, if corresponding to an input symbol, there is single resultant state i.e. there is only one transition.
- A deterministic finite automata is set of five tuples and represented as:

$$\mathbf{M = (Q, \Sigma, \mathit{qo}, F, \delta)}$$

Where,

Q – Non Empty finite set of states

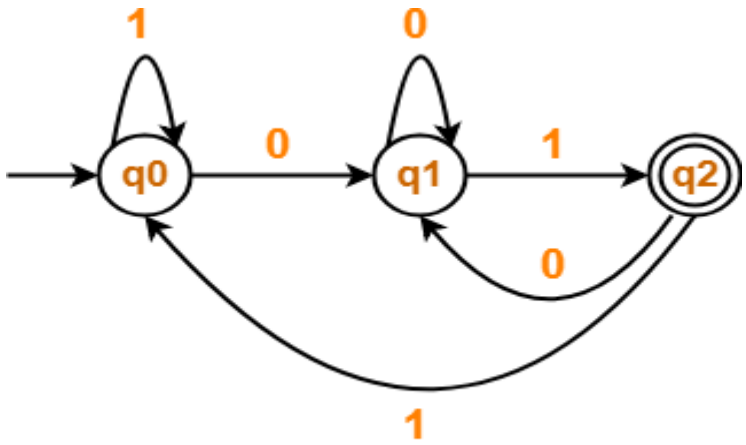
Σ – Non Empty finite set of input symbols

qo – Start/Initial state

F – set of final states

δ – Transition function

$\boldsymbol{\delta : Q \times \Sigma \rightarrow Q}$



DFA

Finite Automata...

Non -Deterministic Finite Automata (NFA) :

- NFA refers to Nondeterministic Finite Automaton.
- A Finite Automata(FA) is said to be non deterministic, if there is more than one possible transition from one state on the same input symbol.
- A non deterministic finite automata is also set of five tuples and represented as:

$$\mathbf{M = (Q, \Sigma, \mathit{qo}, F, \delta)}$$

Where,

Q – Non Empty finite set of states

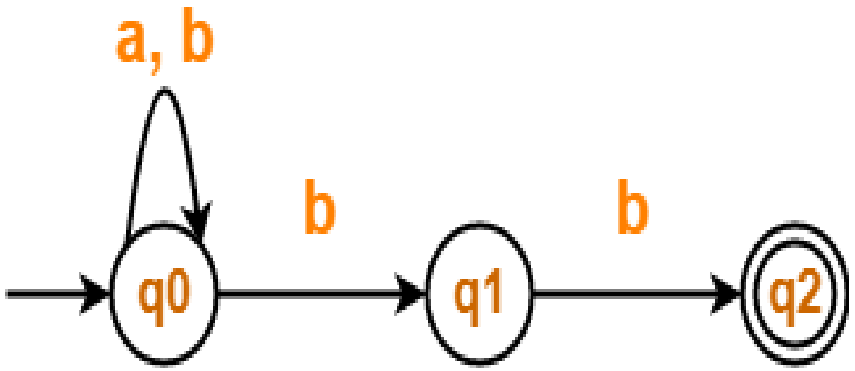
Σ – Non Empty finite set of input symbols

qo – Start/Initial state

F – set of final states

δ – Transition function

$$\delta : Q \times \Sigma \rightarrow 2^{Q}$$



NFA

NFA to DFA Conversion

Let, M = (Q, ∑, δ, q0, F) is an NFA which accepts the language L(M). There should be equivalent DFA denoted by M' = (Q', ∑', q0', δ', F') such that L(M) = L(M').

Steps for converting NFA to DFA:

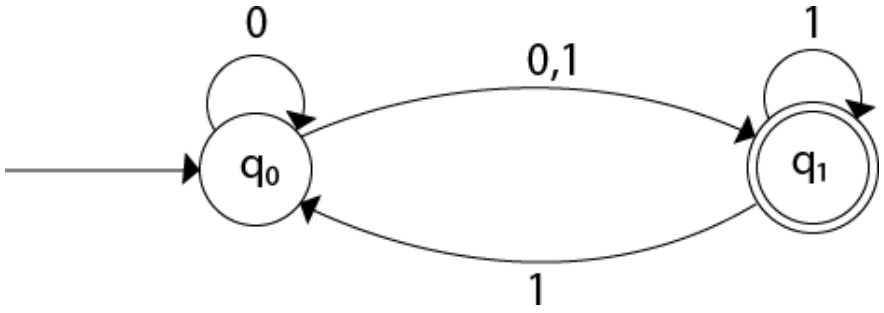
Step 1: Initially Q' = ϕ

Step 2: Add q0 of NFA to Q'. Then find the transitions from this start state.

Step 3: In Q', find the possible set of states for each input symbol. If this set of states is not in Q', then add it to Q'.

Step 4: In DFA, the final state will be all the states which contain F(final states of NFA)

Convert the NFA to DFA:



Transition table for given NFA is

State	0	1
→q0	{q0, q1}	{q1}
*q1	ϕ	{q0, q1}

NFA to DFA Conversion...

Now we will obtain δ' transition for state q0.

$$\delta'([q0], 0) = \{q0, q1\}$$
$$= [q0, q1] \quad \text{(new state generated)}$$
$$\delta'([q0], 1) = \{q1\} = [q1]$$

The δ' transition for state q1 is obtained as:

$$\delta'([q1], 0) = \phi$$
$$\delta'([q1], 1) = [q0, q1]$$

Now we will obtain δ' transition on [q0, q1].

$$\delta'([q0, q1], 0) = \delta(q0, 0) \cup \delta(q1, 0)$$
$$= \{q0, q1\} \cup \phi$$
$$= \{q0, q1\}$$
$$= [q0, q1]$$

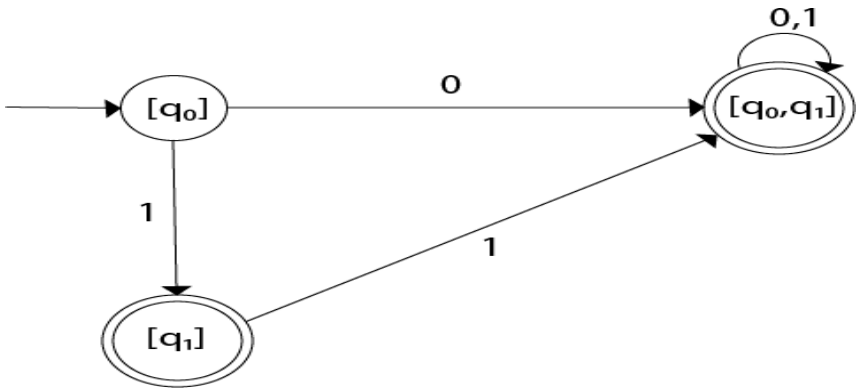
Similarly,

$$\delta'([q0, q1], 1) = \delta(q0, 1) \cup \delta(q1, 1)$$
$$= \{q1\} \cup \{q0, q1\}$$
$$= \{q0, q1\}$$
$$= [q0, q1]$$

DFA Transition table

State	0	1
→[q0]	[q0, q1]	[q1]
*[q1]	ϕ	[q0, q1]
*[q0, q1]	[q0, q1]	[q0, q1]

DFA :



As in the given NFA, q1 is a final state, then in DFA wherever, q1 exists that state becomes a final state. Hence in the DFA, final states are [q1] and [q0, q1]. Therefore set of final states F = {[q1], [q0, q1]}.

ε- NFA

ε- NFA:NFA with ε-Moves

It is a five tuple Machine and represented as:

$$\mathbf{M = (Q, \Sigma, \mathit{qo}, F, \delta)}$$

Where,

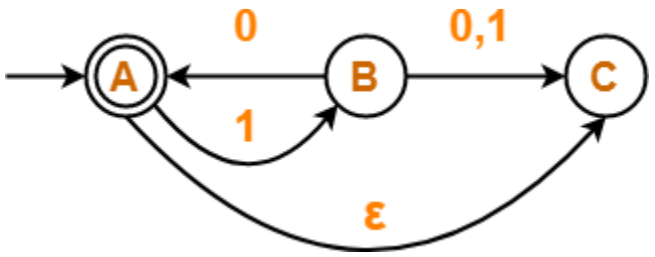
Q – Non Empty finite set of states

Σ – Non Empty finite set of input symbols

qo – Start/Initial state

F – set of final states

δ – Transition function



$$\delta : Q \times \Sigma \cup \{\varepsilon\} \rightarrow 2^{Q}$$

Epsilon Closure:

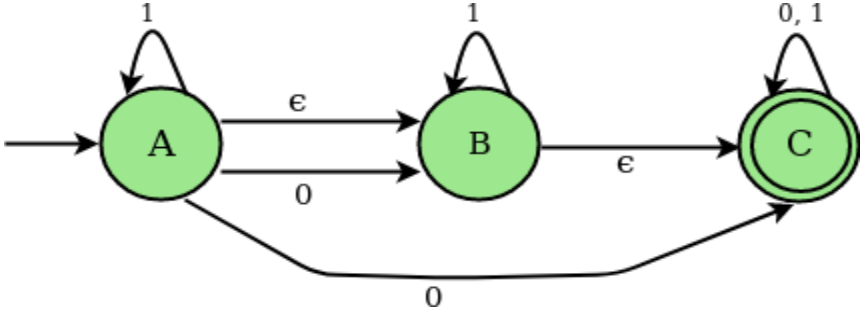
Epsilon closure for a given state X is a set of states which can be reached from the states X with only (null) or ε moves including the state X itself.

Example:

∈ **closure(A) :** {A, B, C}

∈ **closure(B) :** {B, C}

∈ **closure(C) :** {C}



ε- NFA to NFA without ε moves

Steps for converting ε- NFA to NFA without ε moves:

Step-1: Find the ε-closure of the states qi where qi ∈Q

Step-2: Find the Extended transition function as

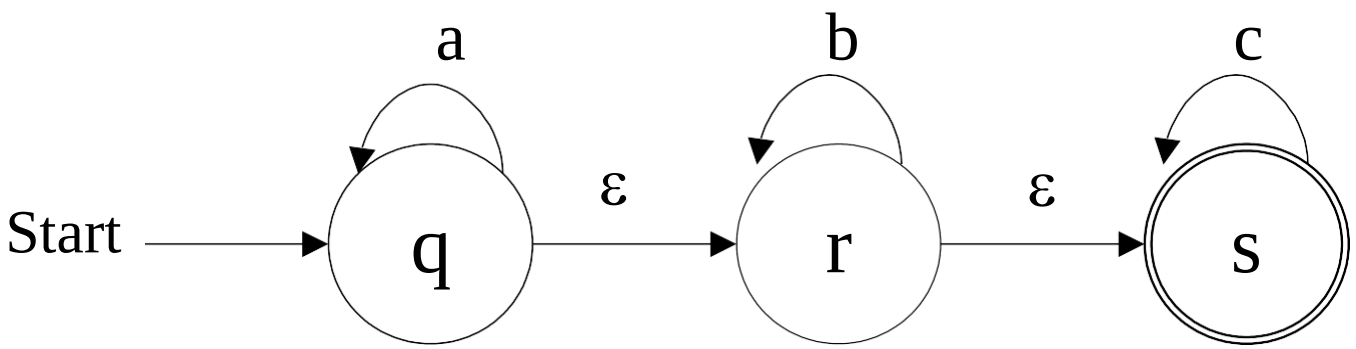
$$\hat{\delta}(q0, \varepsilon) = \varepsilon\text{-closure}(q0)$$
$$\hat{\delta}(q0, a) = \varepsilon\text{-closure}(\delta(\hat{\delta}(q0, \varepsilon), a))$$

repeat this for each input symbol.

Step-3 : Draw the transition table and diagram using resultant transitions.

Step-4: if the ε-closure of the state contains the final state of ε- NFA then make the state as final.

Problem: Convert ε- NFA to NFA



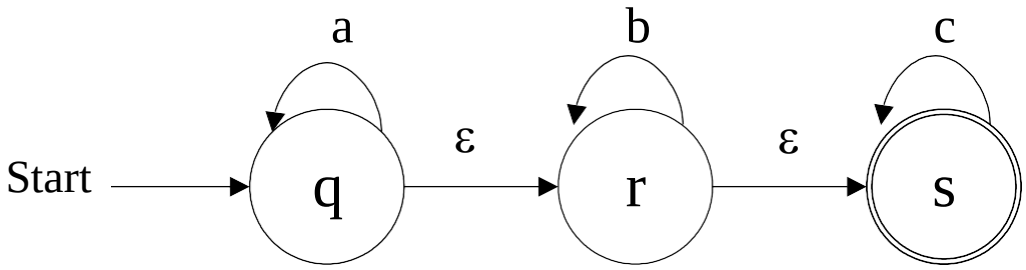
ε- NFA to NFA without ε moves...

Step 1: Find ε closures

ε closure(q)= {q,r,s}

ε closure(r)= {r,s}

ε closure(s)= {s}



Step 2: Find δ for all states

δ'(q,a)= ε closure (δ(δ'(q, ε),a))

= ε closure (δ(ε closure(q),a))

= ε closure(δ((q,r,s),a))

= ε closure (δ(q,a) U δ(r,a) U δ(s,a))

= ε closure (q U θ U θ)

= ε closure (q)

= {q,r,s}

δ'(q,b)= ε closure (δ(δ'(q, ε),b))

= ε closure (δ(ε closure(q),b))

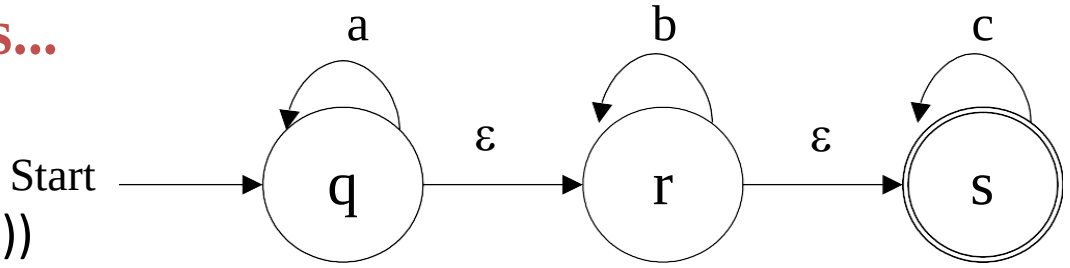
= ε closure(δ((q,r,s),b))

= ε closure (δ(q,b) U δ(r,b) U δ(s,b))

= ε closure (θ Ur U θ)

= ε closure (r) **= {r,s}**

ε- NFA to NFA without ε moves...



δ'(q,c)= ε closure (δ(δ'(q, ε),c))

= ε closure (δ(ε closure(q),c))

= ε closure(δ((q,r,s),c))

= ε closure (δ(q,c) U δ(r,c) U δ(s,c))

= ε closure (θ U θ U s)

= ε closure (s)

= {s}

δ'(r,a)= ε closure (δ(δ'(r, ε),a))

= ε closure (δ(ε closure(r),a))

= ε closure(δ((r,s),a))

= ε closure (δ(r,a) U δ(s,a))

= ε closure (θ U θ) **= θ**

δ'(r,b) = ε closure (δ(δ'(r, ε),b))

= ε closure (δ(ε closure(r),b))

= ε closure(δ((r,s),b))

= ε closure (δ(r,b) U δ(s,b))

= ε closure (r U θ) = ε closure (r) **={r,s}**

ε- NFA to NFA without ε moves...

δ’(r,c)= ε closure (δ(δ’(r, ε),c))

= ε closure (δ(ε closure(r),c))

= ε closure(δ((r,s),c))

= ε closure (δ(r,c) U δ(s,c))

= ε closure (θ U s)

= ε closure (s)

= {s}

δ’{s,a}= ε closure (δ(δ’(s, ε),a))

=ε closure (δ(s,a))

= ε closure (θ)

= θ

δ’{s,b}= ε closure (δ(δ’(s, ε),b))

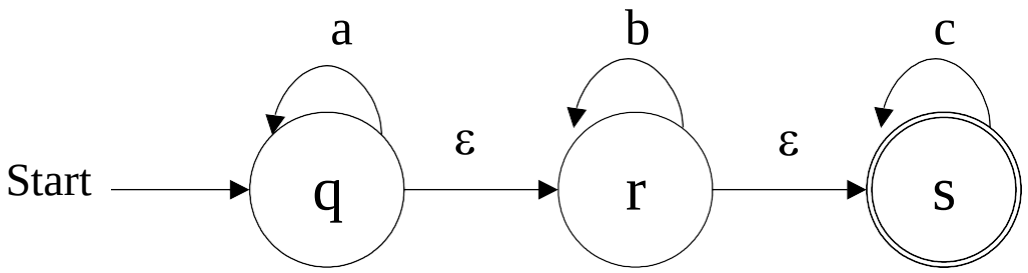
=ε closure (δ(s,b))

= ε closure (θ) **= θ**

δ’{s,c}=ε closure (δ(δ’(s, ε),c))

=ε closure (δ(s,c))

= ε closure (s) **= {s}**



ε- NFA to NFA without ε moves...

Step 3: Draw transition table and diagram for all new states

Let,

(q,r,s)= D

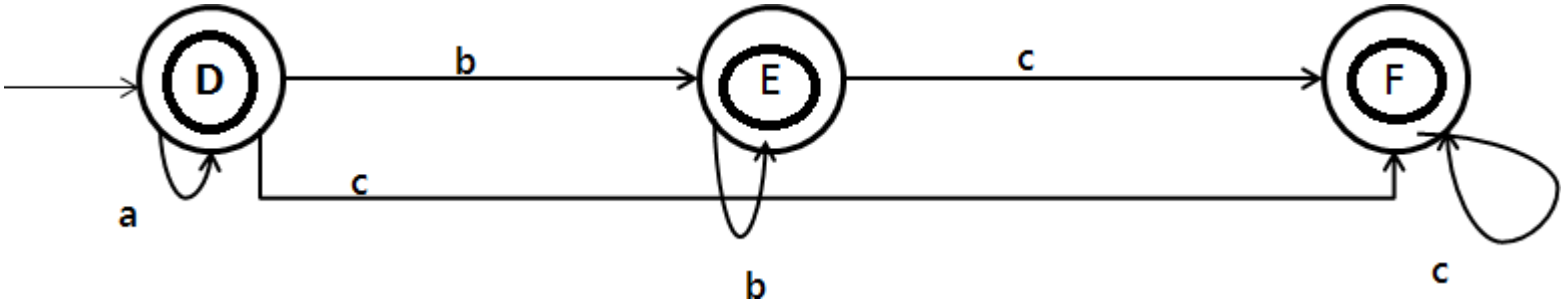
(r,s) =E

s =F

Step 4: Final states are D,E and F.

	a	b	c
->*D	D	E	F
*E	θ	E	F
*F	θ	θ	F

NFA without ε moves:



ε- NFA to DFA

Conversion of ε- NFA to DFA :

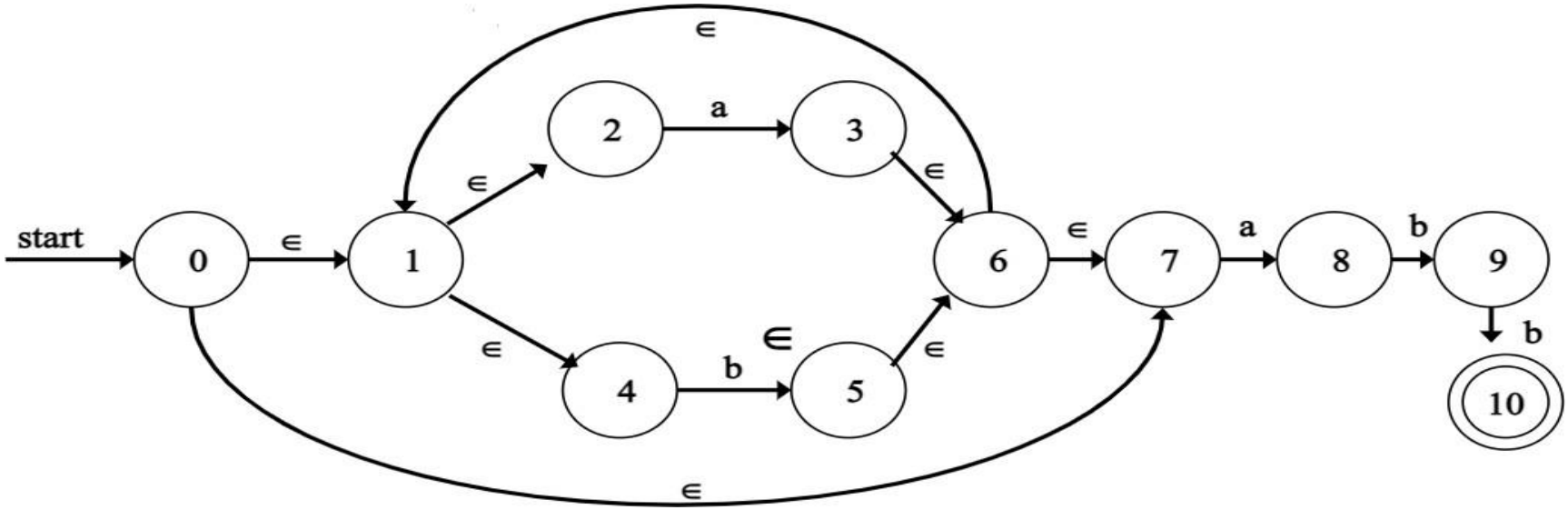
Let the DFA be D and its transition table be Dtrans, Dstates represents states of the DFA and N be the NFA.

1. Initially , ∈ closure(s) is the only state in Dstates and it is Unmarked.
2. While there is an Unmarked state T in Dstates do

```
begin
    Mark T
    for each input symbol "a" do
    begin
     U= ∈ closure(δ( T, a))
     if U is not in Dstates then
         add U as an Unmarked state to DState
     Dtrans[T , a]=U
     End
End
```

ε- NFA to DFA

Conversion of ε- NFA to DFA :



First we calculate: ∈-closure(0) (i.e., state 0)

∈-closure(0) = {0, 1, 2, 4, 7} (all states reachable from 0 on ∈-moves)

Let A={0, 1, 2, 4, 7} be a state of new DFA, D.

ε- NFA to DFA

Conversion of ε- NFA to DFA :

2nd , we calculate : a : ∈-closure (*move*(A,a)) and
b : ∈-closure (*move*(A,b))

a : ∈-closure (*move*(A,a)) = ∈-closure (*move*({0,1,2,4,7},a))}
adds {3,8} (since *move*(2,a)=3 and *move*(7,a)=8)

From this we have : ∈-closure({3,8}) = {1,2,3,4,6,7,8}
(since 3→6 →1 →4, 6 →7, and 1 →2 all by ∈-moves)

Let B={1,2,3,4,6,7,8} be a new state. Define Dtran[A,a] = B.

b : ∈-closure (*move*(A,b)) = ∈-closure (*move*({0,1,2,4,7},b))

adds {5} (since *move*(4,b)=5)

From this we have : ∈-closure({5}) = {1,2,4,5,6,7}
(since 5→6 →1 →4, 6 →7, and 1 →2 all by ∈-moves)

Let C={1,2,4,5,6,7} be a new state. Define Dtran[A,b] = C.

ε- NFA to DFA

Conversion of ε- NFA to DFA :

3rd , we calculate for state B on {a,b}

a : ∈-closure (*move*(B,a)) = ∈-closure (*move*({1,2,3,4,6,7,8},a))}
= {1,2,3,4,6,7,8} = B

Define Dtran[B,a] = B.

b : ∈-closure (*move*(B,b)) = ∈-closure (*move*({1,2,3,4,6,7,8},b))}
= {1,2,4,5,6,7,9} = D

Define Dtran[B,b] = D.

4th , we calculate for state C on {a,b}

a : ∈-closure (*move*(C,a)) = ∈-closure (*move*({1,2,4,5,6,7},a))}
= {1,2,3,4,6,7,8} = B

Define Dtran[C,a] = B.

b : ∈-closure (*move*(C,b)) = ∈-closure (*move*({1,2,4,5,6,7},b))}
= {1,2,4,5,6,7} = C

Define Dtran[C,b] = C.

ε- NFA to DFA

Conversion of ε- NFA to DFA :

5th , we calculate for state D on {a,b}

a : ∈-closure (*move*(D,a)) = ∈-closure (*move*({1,2,4,5,6,7,9},a))}
= {1,2,3,4,6,7,8} = B

Define Dtran[D,a] = B.

b : ∈-closure (*move*(D,b)) = ∈-closure (*move*({1,2,4,5,6,7,9},b))}
= {1,2,4,5,6,7,10} = E

Define Dtran[D,b] = E.

Finally, we calculate for state E on {a,b}

a : ∈-closure (*move*(E,a)) = ∈-closure (*move*({1,2,4,5,6,7,10},a))}
= {1,2,3,4,6,7,8} = B

Define Dtran[E,a] = B.

b : ∈-closure (*move*(E,b)) = ∈-closure (*move*({1,2,4,5,6,7,10},b))}
= {1,2,4,5,6,7} = C

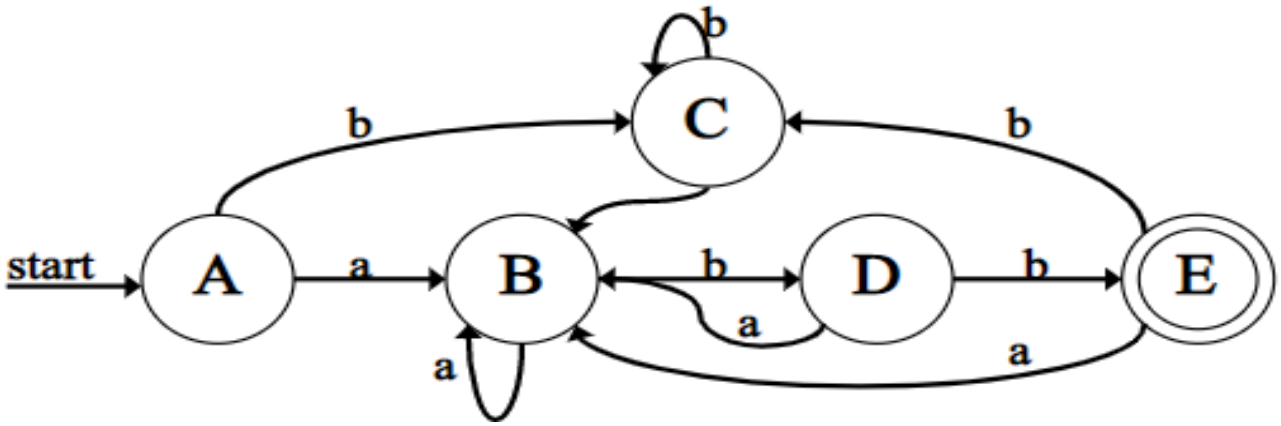
Define Dtran[E,b] = C.

ε- NFA to DFA

Conversion of ε- NFA to DFA :

This gives the transition table Dtran for the DFA of:

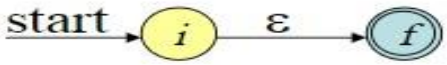
Dstates	Input Symbol	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C



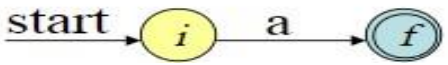
Converting RE to NFA

Conversion of RE to NFA (Thompson Construction)

- Empty string ε is a regular expression denoting { ε }

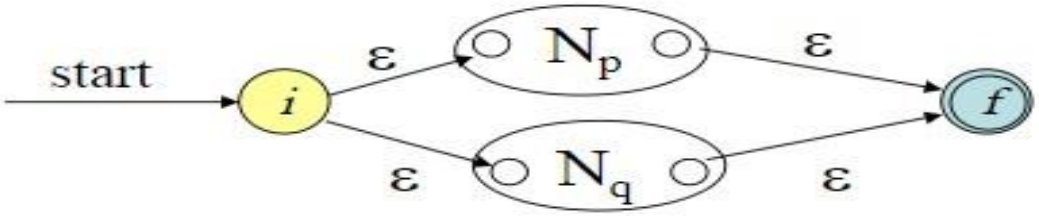


- *a* is a regular expression denoting {*a*} for any *a* in Σ

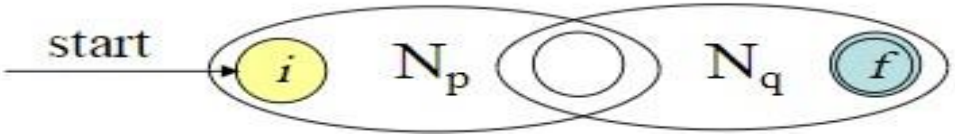


If P and Q are regular expressions with NFAs N_p, N_q:

P | Q (union)



PQ (concatenation)

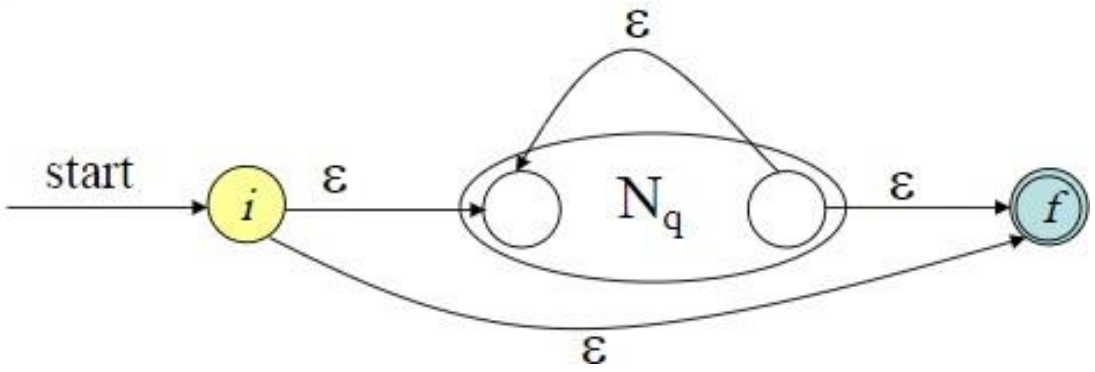


Converting RE to NFA

Conversion of RE to NFA (Thompson Construction)

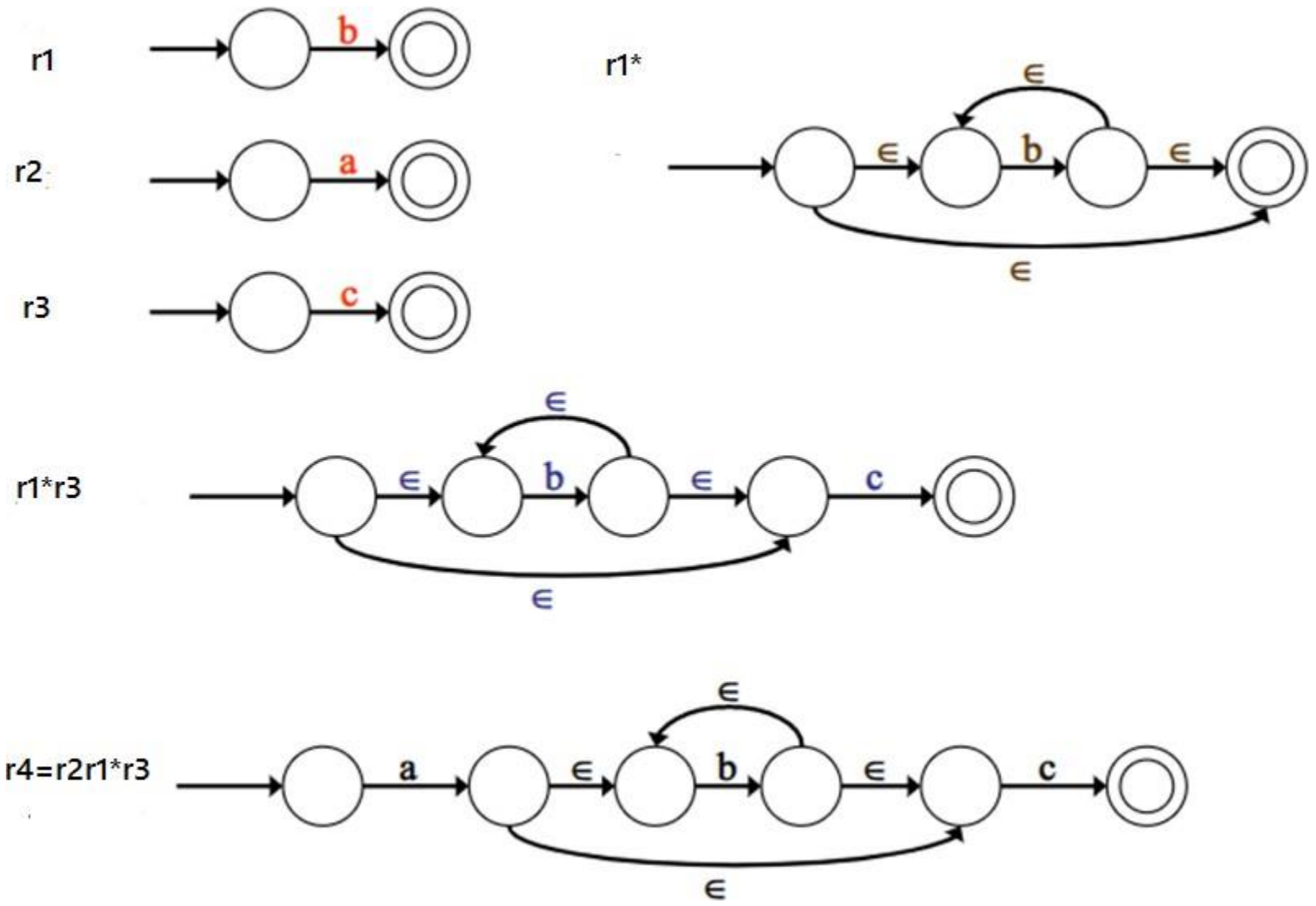
If Q is a regular expression with NFA N_q:

Q* (closure)



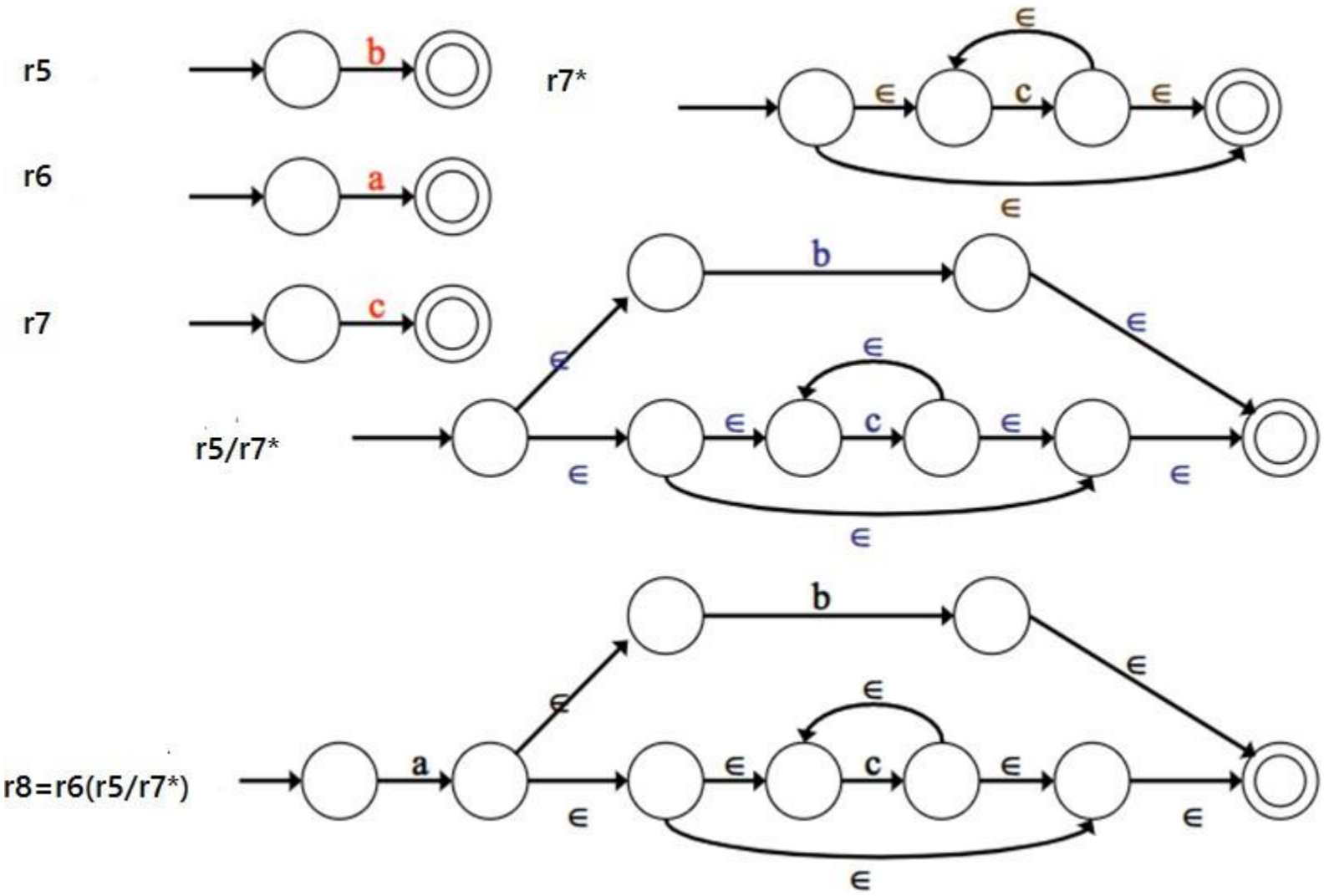
Converting RE to NFA

Problem: Convert the RE (ab*c)/ (a(b/c*)) to NFA



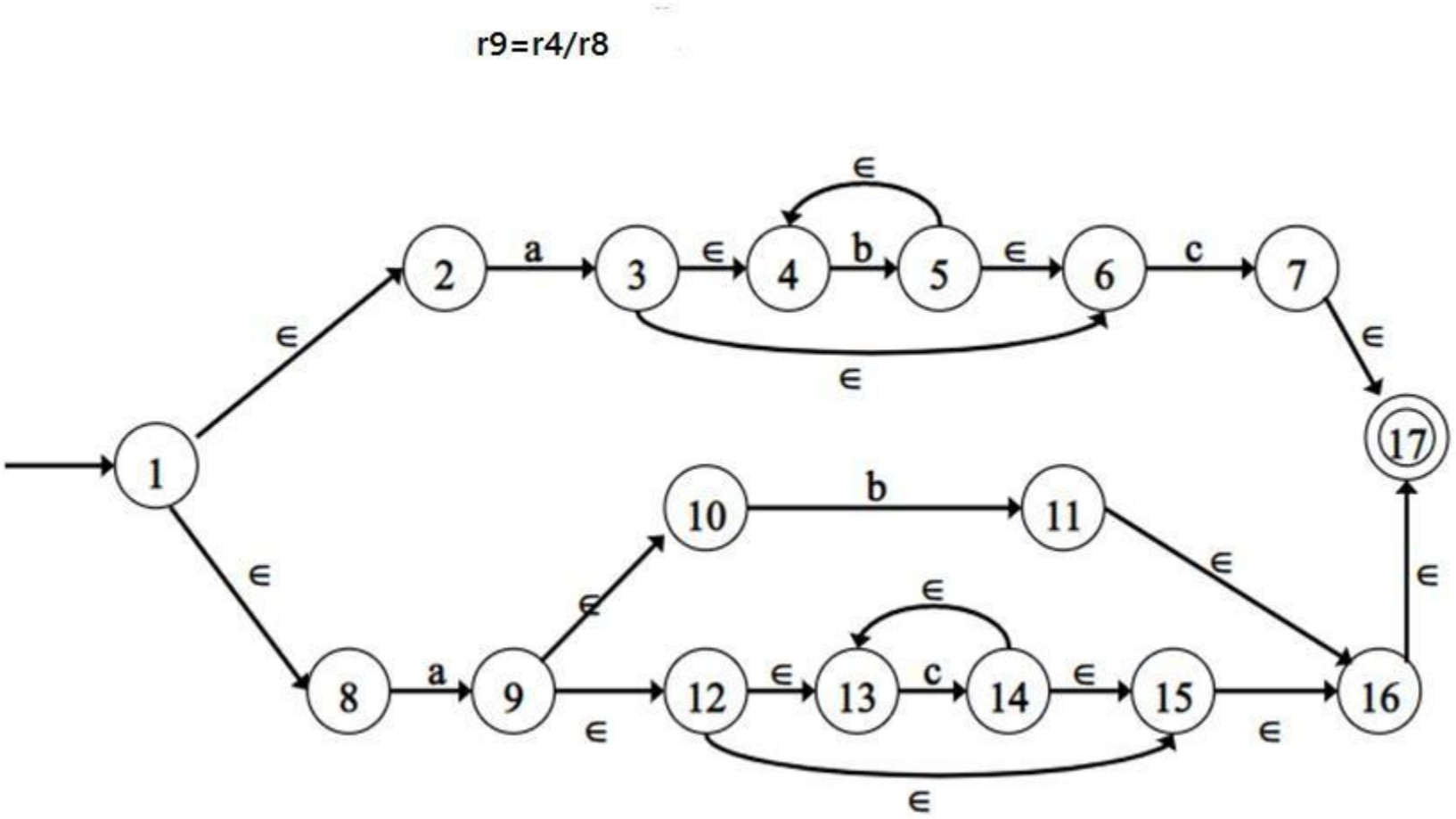
Converting RE to NFA

Convert the RE (ab*c)/ (a(b/c*)) to NFA



Converting RE to NFA

Convert the RE (ab*c)/ (a(b/c*)) to NFA

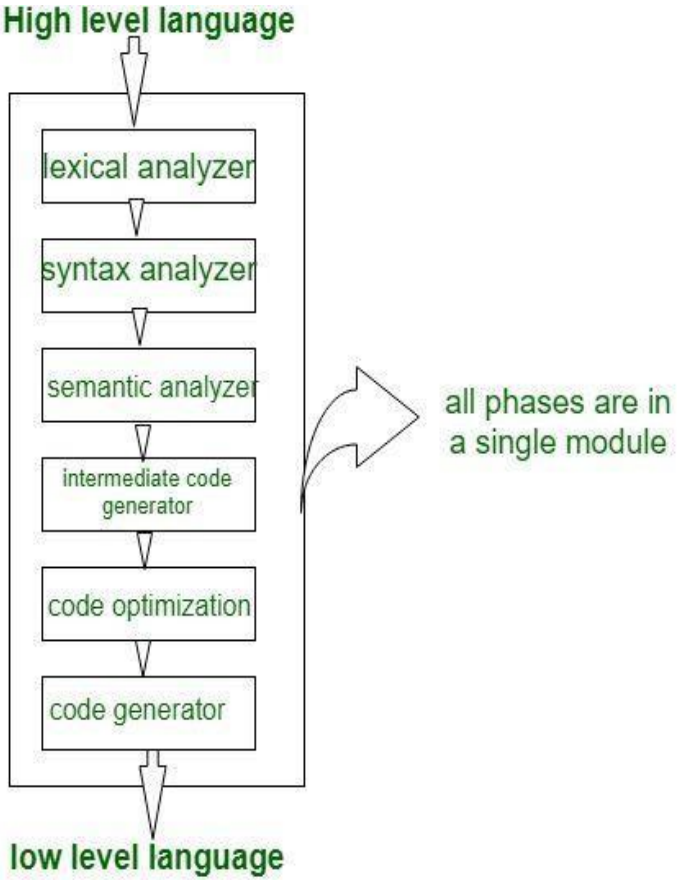


Pass and Phases of Translation

- A compiler can have many phases and passes.
- **Pass** : A pass refers to the traversal of a compiler through the entire program.
- **Phase** : A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage.
- Compiler pass are two types:
    1. Single Pass Compiler
    2. Two Pass Compiler *or* Multi Pass Compiler.

Single Pass Compiler(Narrow Compilers):

- If we combine or group all the phases of compiler design in a **single** module known as single pass compiler.
- A one pass/single pass compiler is that type of compiler that passes through the part of each compilation unit exactly once.
- Single pass compiler is faster and smaller than the multi pass compiler.
- As a disadvantage of single pass compiler is that it is less efficient in comparison with

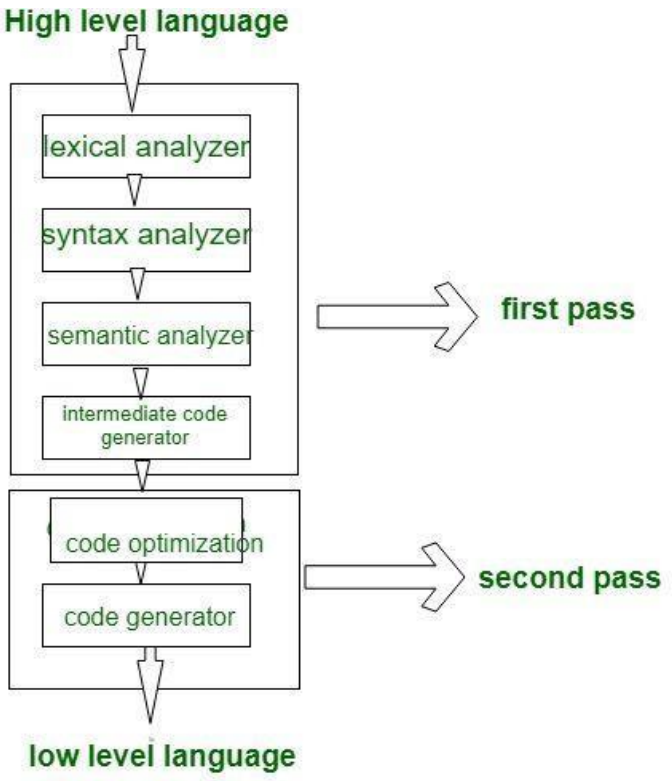


multipass compiler.

Pass and Phases of Translation...

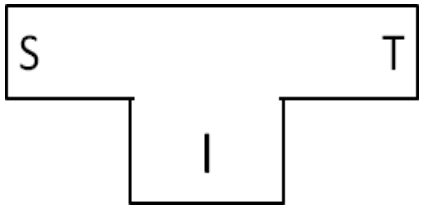
Multipass Compiler(Wide Compilers):

- A Two pass/multi-pass Compiler is a type of compiler that processes the source code of a program multiple times. In multipass Compiler we divide phases in two pass as:
- In first pass the included phases are as Lexical analyzer, syntax analyzer, semantic analyzer, intermediate code generator are work as front end
- First pass is platform independent because the output of first pass is as three address code which is useful for every system .
- In second Pass the included phases are as Code optimization and Code generator are work as back end and the synthesis part refers to taking input as three address code and convert them into Low level language/assembly language and second pass is platform dependent because final stage of a typical compiler converts the intermediate representation of program into an executable set of instructions which is dependent on the system.

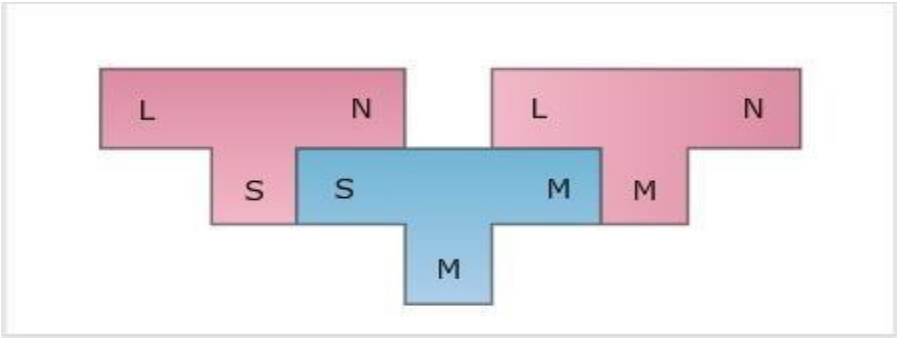


Bootstrapping

- Bootstrapping is widely used in the compilation development.
- It is a process in which simple language is used to translate more complicated program which in turn may handle for more complicated program. This complicated program can further handle even more complicated program and so on.
- It is used to produce a self-hosting compiler.
- Self-hosting compiler is a type of compiler that can compile its own source code . i.e. a compiler written in the source programming language that it intends to compile.
- A compiler can be characterized by three languages:
  1) Source Language
  2) Target Language
  3) Implementation Language



- The T- diagram shows a compiler ${}^{S}C_{I}^{T}$ for Source S, Target T, implemented in I.
- Cross Compiler is a compiler which runs on one machine and produces output for another machine.



Cross Compiler

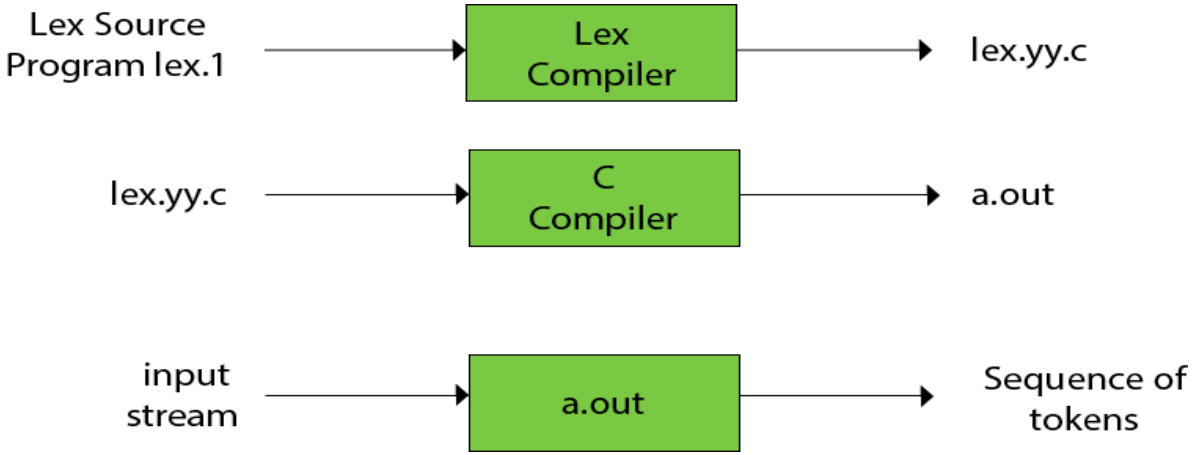
LEX

LEX:

- Lex is a program that generates lexical analyzer.
- It is a Unix utility.
- The lexical analyzer is a program that transforms an input stream into a sequence of tokens.
- Lex specifies tokens using Regular Expression.

The function of Lex is as follows:

1. Firstly lexical analyzer creates a program called lex specification file , lex.l in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.
2. Finally C compiler runs the lex.yy.c program and produces an object program a.out.
3. a.out is lexical analyzer that transforms an input stream into a sequence of tokens.



LEX...

The structure of LEX programs:

%{
Declarations
%}
%%
Rules
%%
Auxiliary Functions

Declaration Section:

- The declarations section consists of two parts, **auxiliary declarations** and **regular definitions**.
- **The auxiliary declarations** are copied as such by LEX to the output *lex.yy.c* file. This C code consists of instructions to the C compiler and are not processed by the LEX tool.
- The auxiliary declarations (which are optional) are written in C language and are enclosed within ' %{ ' and ' %} ' .
- It is generally used to declare functions, include header files, or define global variables and constants.
- LEX allows the use of short-hands and extensions to regular expressions for **the regular definitions.** A regular definition in LEX is of the form : **D R** where D is the symbol representing the regular expression R.

LEX...

Rules:

- Rules in a LEX program consists of two parts :
  1. The pattern to be matched
  2. The corresponding action to be executed
- Patterns are defined using the regular expressions and actions can be specified using C Code.
- The Rules can be given as

R1 {Action1}
R2 {Action2}
.
.
.
Rn {Action n}

Where Ri is RE and Action i is the action to be taken for corresponding RE.

Auxiliary Functions:

- All the required procedures are defined in this section.

Note: Function **yywrap** is called by lex when input is exhausted. When the end of the file is reached the return value of **yywrap**() is checked. If it is non-zero, scanning terminates and if it is 0 scanning continues with next input file.

LEX...

Lex Program for count tokens in source program:

```
%{
int n = 0 ;
%}

// rule section
%%

//count number of keywords
"while"|"if"|"else" {n++;printf("\t keywords : %s", yytext);}

// count number of keywords
"int"|"float" {n++;printf("\t keywords : %s", yytext);}

// count number of identifiers
[a-zA-Z_][a-zA-Z0-9_]* {n++;printf("\t identifier : %s", yytext);}

// count number of operators
"<="|"=="|"="|"++"|"-"|"*"|"+" {n++;printf("\t operator : %s", yytext);}

// count number of separators
[(){}|, ;]    {n++;printf("\t separator : %s", yytext);}

// count number of floats
[0-9]*"."[0-9]+ {n++;printf("\t float : %s", yytext);}
```

```
// count number of integers
[0-9]+ {n++;printf("\t integer : %s", yytext);}

.    ;
%%


int main()

{

    yylex();

    printf("\n total no. of token = %d\n", n);

}
```

Note: yylex() match the characters with the regular expression.

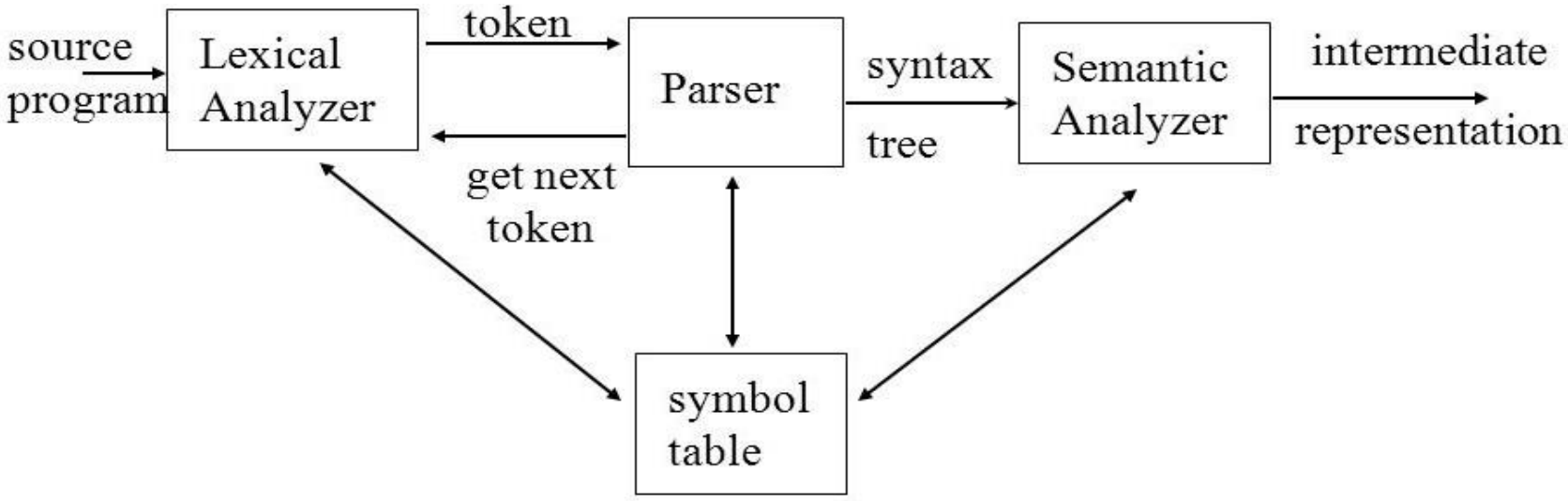
Compiler Design: Parsing

UNIT – II:

Top down Parsing: Context free grammars, Top down parsing – Backtracking, LL (1), recursive descent parsing, Predictive parsing, Pre-processing steps required for predictive parsing.

Bottom up parsing: Shift Reduce parsing, SLR,CLR and LALR parsing, Error recovery in parsing , handling ambiguous grammar, YACC –automatic parser generator.

Role of a Parser:



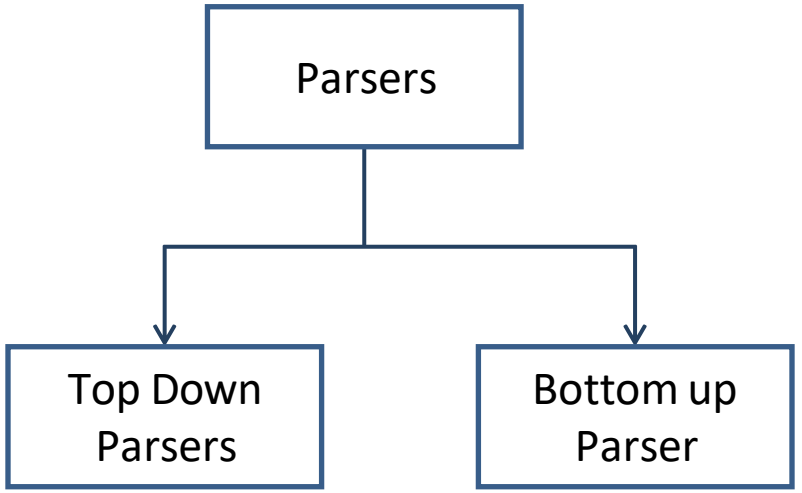
Parsing

- A syntax analyzer is also known as parser.
- A parser takes input in the form of a sequence of tokens from Lexical Analyzer and builds a data structure in the form of a parse tree.
- It verifies whether the string can be generated by the grammar for the source language.
- It also returns any syntax error for the source language.
- A parser for a grammar G is a program that takes input as a string s and produces an output either,
  - ✓ A parser tree for s, if s is a sentence of G or
  - ✓ An error message indicating that s is not a sentence of G

Types of Parsers:

1. Top down Parsers
2. Bottom up Parsers

- Top down parser builds the parse tree from root to leaves.
- Bottom up parser builds the parse tree from leaves to root.
- In both the cases input is scanned from left to right one symbol at a time.



Context Free Grammar (CFG)

- A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple

  G=(V, T, P, S)

  where

  V is a set of non-terminal .

  T is a set of terminals.

  P is a set of Production rules,

  $$\mathbf{P: V \rightarrow (V \cup T)^*}$$

  S is the start symbol.

- A context-free grammar is a set of recursive rules used to generate patterns of strings.
- The language generated using Context Free Grammar is called as **Context Free Language**.

Example:

G = (V , T , P , S)

Where,

V = { S }

T = { a , b }

P = { $S \rightarrow aSbS$, $S \rightarrow bSaS$, $S \rightarrow \in$ }

S = { S }

Parse Tree/ Syntax Tree/ Derivation Tree

Parse Tree:

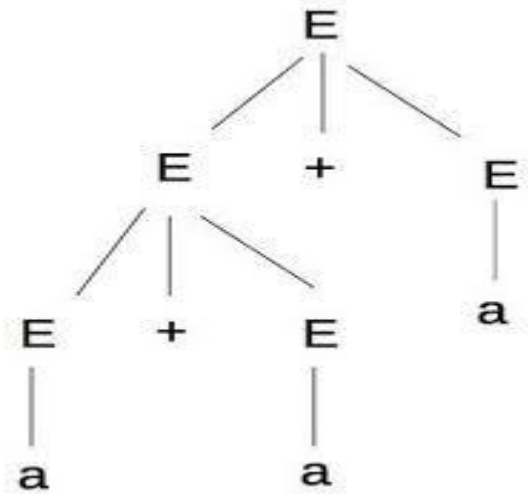
- The diagrammatical representation of a derivation is called as a **parse tree** or **derivation tree.**
- Root node of a parse tree is the start symbol of the grammar.
- Each leaf node of a parse tree represents a terminal symbol.
- Each interior node of a parse tree represents a non-terminal symbol.
- Concatenating the leaves of a parse tree from the left produces a string of terminals, called as **yield of a parse tree**.

Example:

Construct Parse tree for the string w=a+a+a

G:

E → E+E | E*E |E| a



Derivations

Derivation: Starting with the start symbol, non-terminals are rewritten using productions rules until only terminals remain.

There are two types of derivations:

1.Left Most Derivations(LMD)

2.Right Most Derivations(RMD)

Left Most Derivations (LMD):

A left most derivation is obtained by applying rule of production to the left most variable/ non terminal in each step of derivation.

Example:

Let any set of production rules in a CFG be

X → X+X | X*X |X| a

The leftmost derivation for the string **"a+a*a"** is

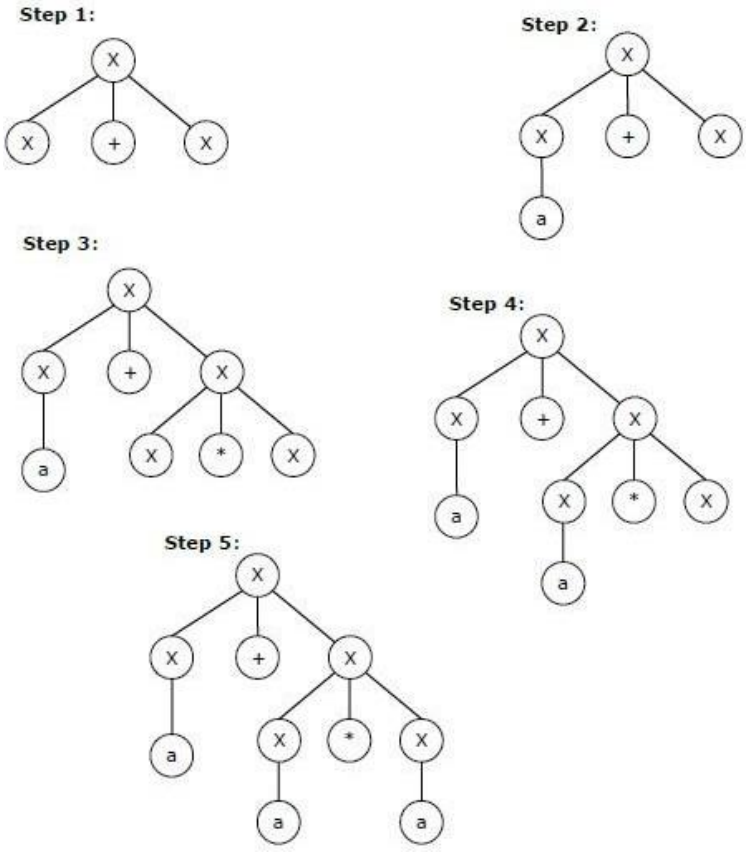
X → **X**+X X →a

→ a+**X** X →X*X

→ a + **X***X X →a

→ a+a***X** X →a

→ a+a*a



Derivations...

Right Most Derivations (LMD):

A right most derivation is obtained by applying rule of production to the right most variable/ non terminal in each step of derivation.

Example:

Let any set of production rules in a CFG be

X → X+X | X*X |X| a

The leftmost derivation for the string **"a+a*a"** is

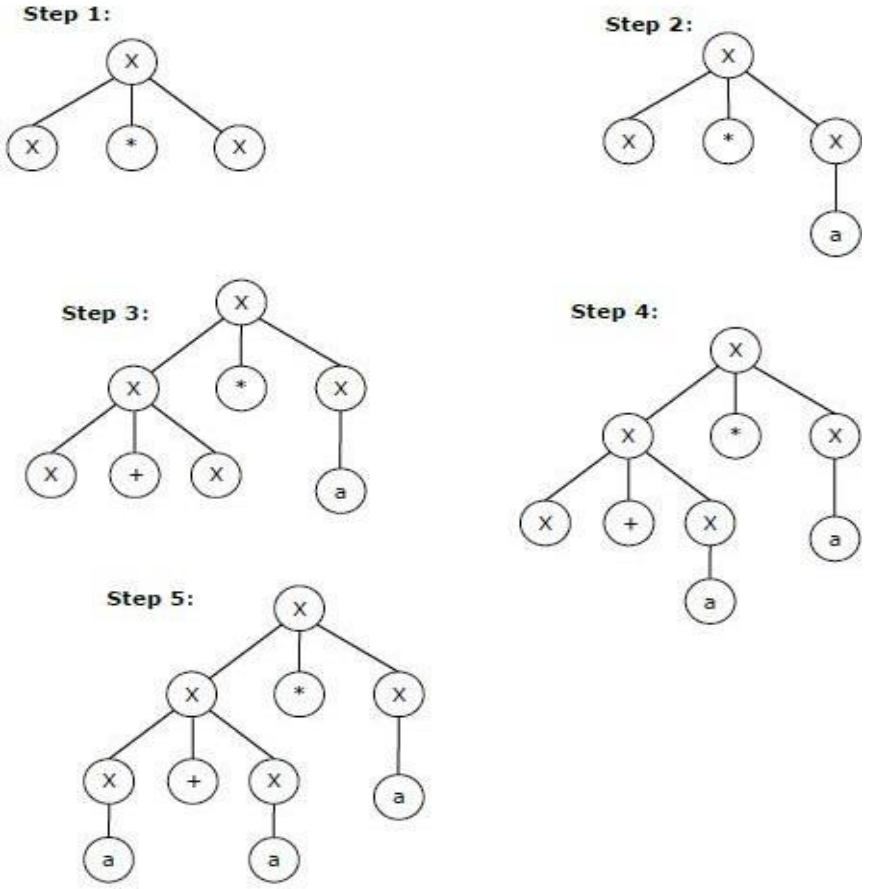
X → X*X X →a

→ X*a X →X+X

→ X+X*a X →a

→ X+a*a X →a

→ a+a*a



Derivations...

Example:

Consider the following grammar

G:

S → aB / bA

A→ aS / bAA / a

B->bS/aBB/b

find LMD,RMD for string w = aaabbabbba

LMD:

S → a**B**

→ aa**B**B (Using B → aBB)

→ aaa**B**BB (Using B → aBB)

→ aaab**B**B (Using B → b)

→ aaabb**B** (Using B → b)

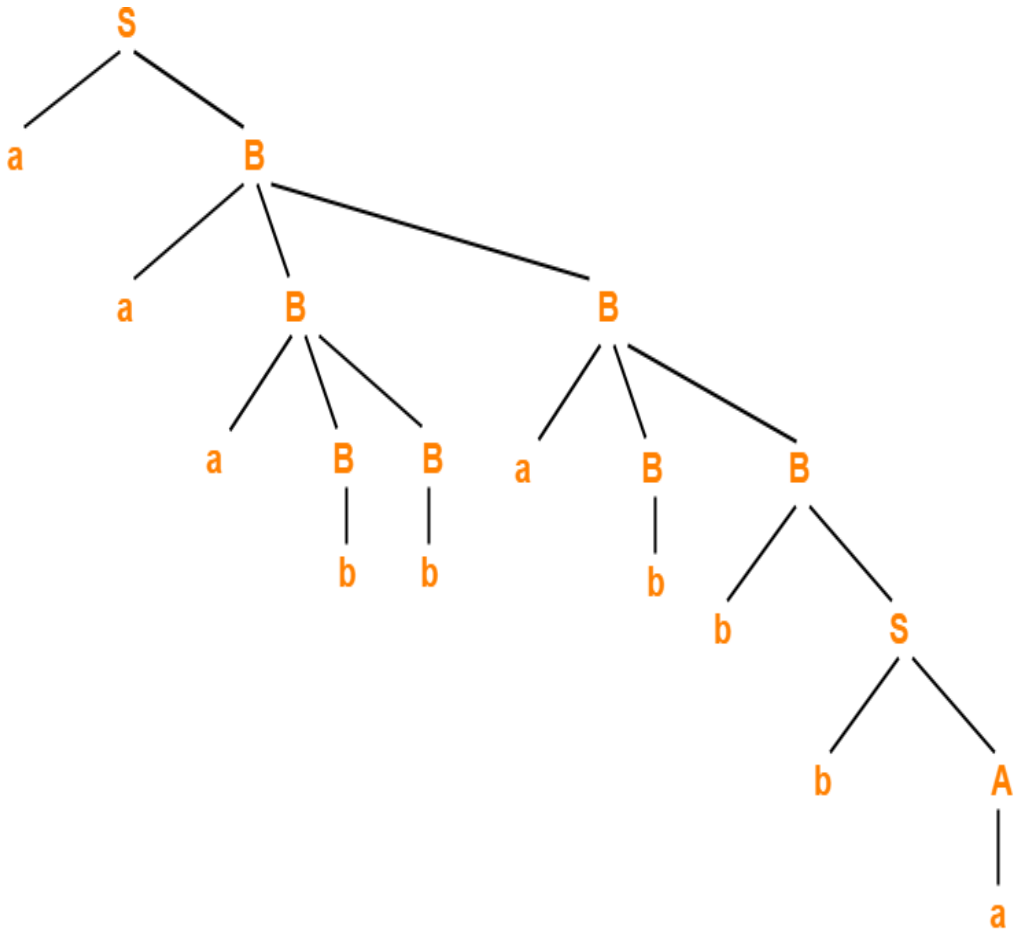
→ aaabba**B**B (Using B → aBB)

→ aaabbab**B** (Using B → b)

→ aaabbabb**S** (Using B → bS)

→ aaabbabbb**A** (Using S → bA)

→ aaabbabbba (Using A → a)



Leftmost Derivation Tree

Derivations...

Example:

Consider the following grammar

G:

S → aB / bA

A → aS / bAA / a

B->bS/aBB/b

find LMD,RMD for string w = aaabbabbba

RMD:

S → a**B**

→ aaB**B** (Using B → aBB)

→ aaBaB**B** (Using B → aBB)

→ aaBaBb**S** (Using B → bS)

→ aaBaBbb**A** (Using S → bA)

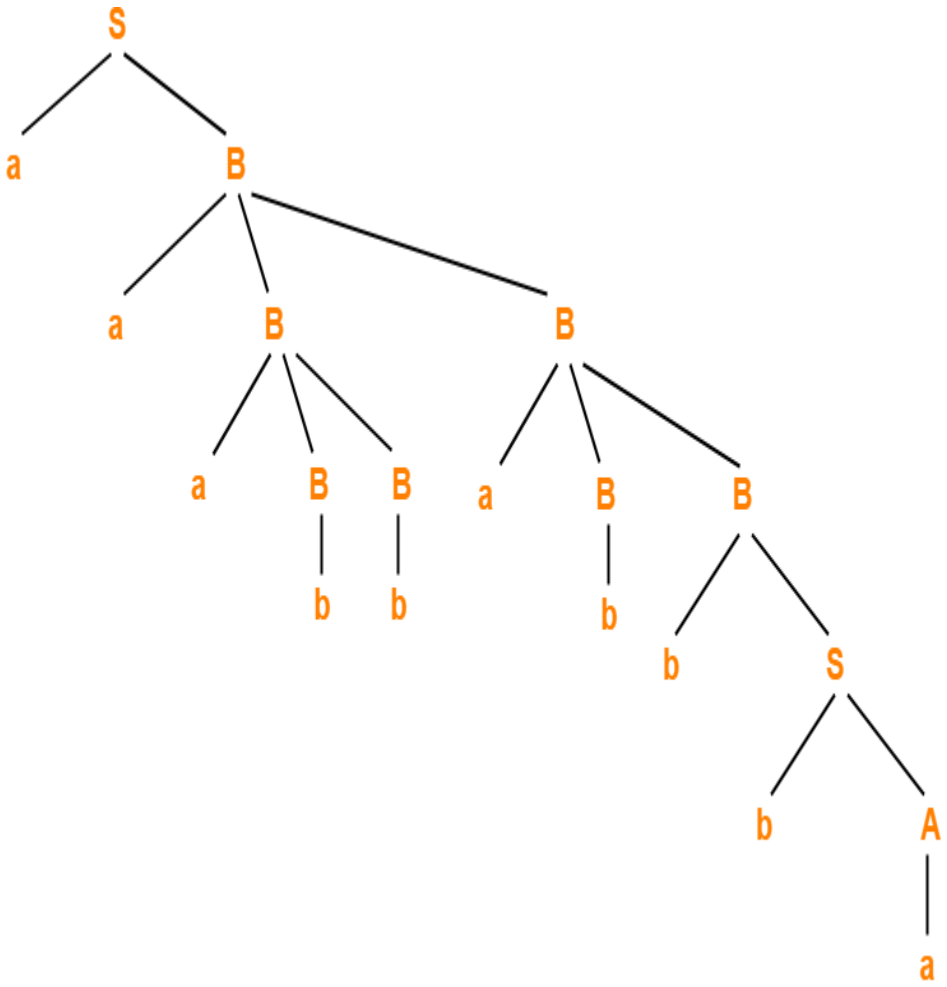
→ aaBa**B**bba (Using A → a)

→ aa**B**abbba (Using B → b)

→ aaaB**B**abbba (Using B → aBB)

→ aaa**B**babbba (Using B → b)

→ aaabbabbba (Using B → b)



Rightmost Derivation Tree

Ambiguous Grammar

Ambiguous Grammar:

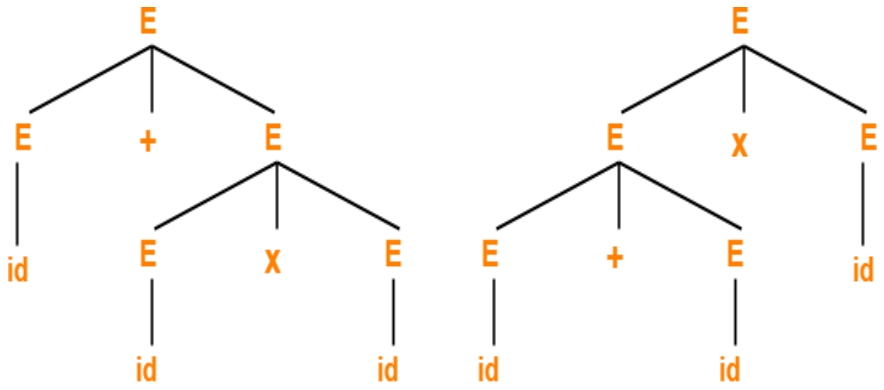
A grammar is said to ambiguous if for any string generated by it, it produces more than one Parse Tree or **Leftmost D**erivation (LMD) or **Rightmost D**erivation (RMD).

Example-

Consider the following grammar-

$$E \rightarrow E + E / E \times E / id$$

Let w = id + id x id be string generated by G



Parse Tree-01

Parse Tree-02

$E \rightarrow \mathbf{E} + E$
$\rightarrow id + \mathbf{E}$
$\rightarrow id + \mathbf{E} \times E$
$\rightarrow id + id \times \mathbf{E}$
$\rightarrow id + id \times id$

Leftmost Derivation-01

$E \rightarrow \mathbf{E} \times E$
$\rightarrow \mathbf{E} + E \times E$
$\rightarrow id + \mathbf{E} \times E$
$\rightarrow id + id \times \mathbf{E}$
$\rightarrow id + id \times id$

Leftmost Derivation-02

$E \rightarrow E + \mathbf{E}$
$\rightarrow E + E \times \mathbf{E}$
$\rightarrow E + \mathbf{E} \times id$
$\rightarrow \mathbf{E} + id \times id$
$\rightarrow id + id \times id$

Rightmost Derivation-01

$E \rightarrow E \times \mathbf{E}$
$\rightarrow \mathbf{E} \times id$
$\rightarrow E + \mathbf{E} \times id$
$\rightarrow \mathbf{E} + id \times id$
$\rightarrow id + id \times id$

Rightmost Derivation-02

There fore , This grammar is an ambiguous grammar.

Unambiguous Grammar

Unambiguous Grammar:

A grammar is said to Unambiguous if for any string generated by it, it produces Exactly one Parse Tree or **Leftmost D**erivation (LMD) or **Rightmost D**erivation (RMD).

Example-

Unambiguous grammar:

X -> AB

A -> Aa / a

B -> b

Problems:

1.Check whether the given grammar G is ambiguous or not.

S → aSb | SS

S → ε

2. Check whether the given grammar G is ambiguous or not.

A → AA

A → (A)

A → a

Left Recursive Grammars

Left Recursive Grammar:

- A production of grammar is said to have **left recursion** if the leftmost variable of its RHS is same as variable of its LHS.
- A grammar containing a production having left recursion is called as Left Recursive Grammar.

Example of Left Recursive Grammar:

G: $A \rightarrow ABd / Aa / a$

$B \rightarrow Be / b$

- Top down parsers cannot handle the Left Recursive Grammars. Therefore, left recursion has to be eliminated from the grammar.

Pre-processing steps in Predictive Parsing:

1) Elimination of Left Recursion:

If a Grammar G is a Left Recursive

$$\mathbf{A \rightarrow A\alpha_1 / A\alpha_2 ..... / A\alpha_m / \beta_1 / \beta_2 / \beta_3 ..... / \beta_n}$$

After eliminating Left Recursion ,We get

$$\mathbf{A \rightarrow \beta_1 A^l / \beta_2 A^l / \beta_3 A^l ..... / \beta_n A^l}$$

$$\mathbf{A^l \rightarrow \alpha_1 A^l / \alpha_2 A^l / ..... / \alpha_m A^l / \varepsilon}$$

Left Recursive Grammars...

Example 1:

Eliminate the left Recursion in the

G: A → ABd / Aa / a

B → Be / b

C →c

A → ABd / Aa / a

After eliminating Left Recursion ,We get

$A \rightarrow aA^{l}$

$A^{l} \rightarrow Bd\ A^{l} / a\ A^{l} / \varepsilon$

B → Be / b

After eliminating Left Recursion ,We get

$B \rightarrow bB^{l}$

$B^{l} \rightarrow e\ B^{l} / \varepsilon$

Grammar After Eliminating Left Recursion is

$A \rightarrow aA^{l}$

$A^{l} \rightarrow Bd\ A^{l} / a\ A^{l} / \varepsilon$

$B \rightarrow bB^{l}$

$B^{l} \rightarrow e\ B^{l} / \varepsilon$

$C \rightarrow c$

Example 2:

Eliminate left recursion in the Grammar

E → E + T / T

T → T * F / F

F → id

The grammar after eliminating left recursion is

E → TE‘

E‘ → +TE‘ / ∈

T → FT‘

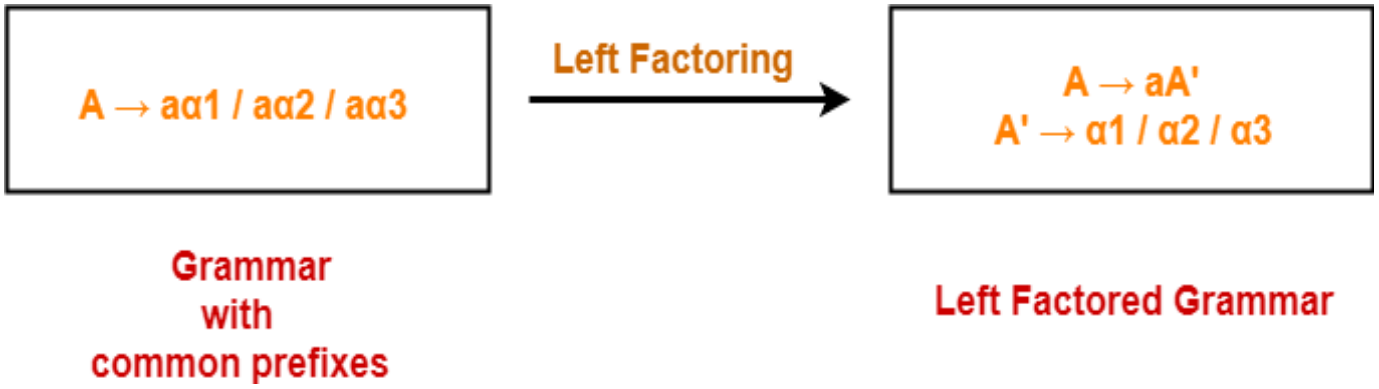
T‘ → *FT‘ / ∈

F → id

Left Factoring

2) Left Factoring:

- It is a grammar transformation that is useful for producing a grammar useful for predictive parsing.
- If $A \rightarrow \alpha\beta_1 / \alpha\beta_2$ are two A-productions ,both these productions starts with same string in RHS , then such grammars are said to be having common prefixes.
- Left factoring is a process by which the grammar with common prefixes is transformed to make it useful for Top down parsers.



Example 1:

Do left factoring in the following grammar-

S → iEtS / iEtSeS / a

E → b

The left factored grammar is:

S → iEtSS‘ / a

S‘ → eS / ∈

E → b

Left Factoring...

Example 2:

Do left factoring in the following grammar-

A → aAB / aBc / aAc

Solution-

Step 1:

A → aA‘

A‘ → AB / Bc / Ac

Again, this is a grammar with common prefixes.

Step 2:

A → aA‘

A‘ → AD / Bc

D → B / c

This is a left factored grammar.

Example 3:

Do left factoring in the following grammar-

S → bSSaaS / bSSaSb / bSb / a

Solution-

Step-01:

S → bSS‘ / a

S‘ → SaaS / SaSb / b

Again, this is a grammar with common prefixes.

Step-02:

S → bSS‘ / a

S‘ → SaA / b

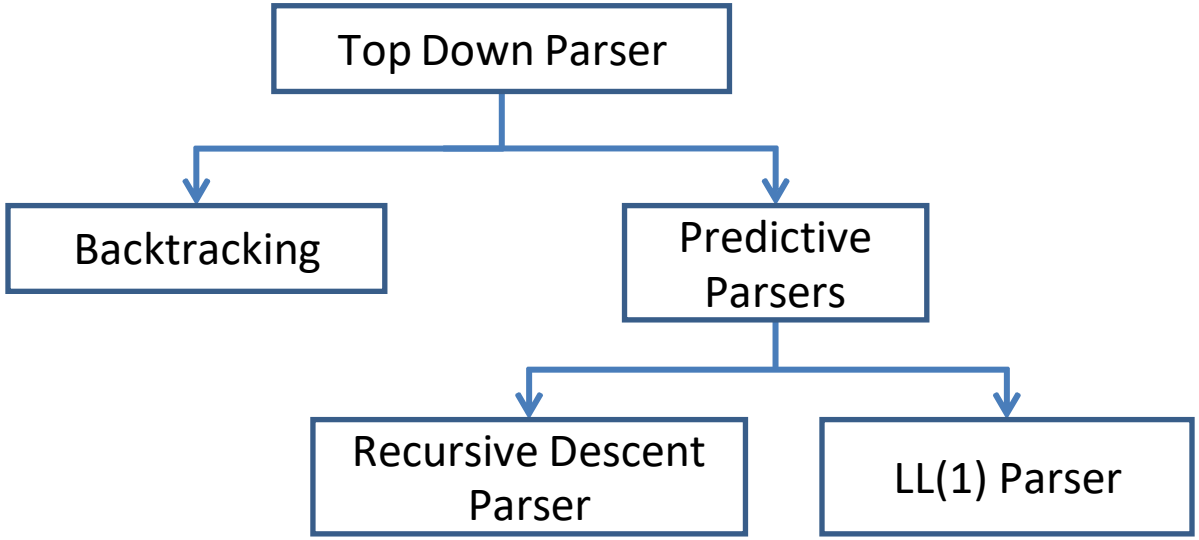
A → aS / Sb

This is a left factored grammar.

Top Down Parsing

Top Down Parsers:

- Top-down parsers build parse trees from the top (root) to the bottom (leaves).
- Top down parsers are classified as follow:



Backtracking:

- Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched).
- It means, if one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production.
- This technique may process the input string more than once to determine the right production.

Backtracking...

Example:

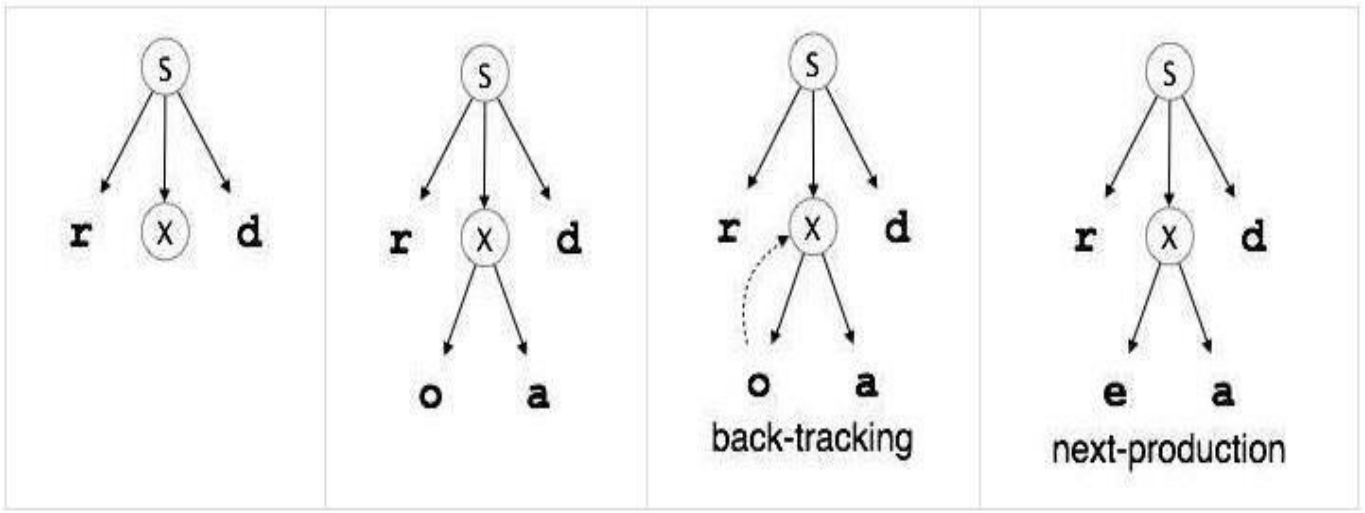
G:

S → rXd | rZd

X → oa | ea

Z → ai

and input **w**=—read‖



- It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. _r‘.
- The very production of S (S → rXd) matches with it. So the top-down parser advances to the next input letter (i.e. _e‘).
- The parser tries to expand non-terminal _X‘ and checks its production from the left (X → oa). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, (X → ea).
- Now the parser matches all the input letters in an ordered manner. The string is accepted and parsing is successful.

Recursive Descent Parsing

Recursive Descent Parser:

- It is a top-down parser builds the parse tree from the top to down, starting with the start non-terminal.
- It is a Predictive Parser where no Backtracking is required.
- In this parsing technique each non terminal is associated with a recursive procedure.
- The RHS of the production rule is directly converted to code of the respective procedure.
- If the RHS of production rule is containing a non terminal ,then it will invoke the respective procedure.
- If it is a terminal then it is matched with lookahead from the input string, lookahead pointer is moved one position to right if match is found.
- These procedures are responsible for matching the non terminal with next part of the input.
- If the production rule have many alternatives then all the alternatives are combined into a single body of the procedure.
- Since, it is a top down parsing technique the parser is activated by calling the procedure of start symbol.

Recursive Descent Parsing...

Example:

G:

E --> i E'

E' --> + i E' | e

E()

```
{
    if (lookahead == 'i')
    {
        match('i');
        E'();
    }
    else if  (lookahead == _$')
        printf("Parsing Successful");
    else
        return error;
}
```

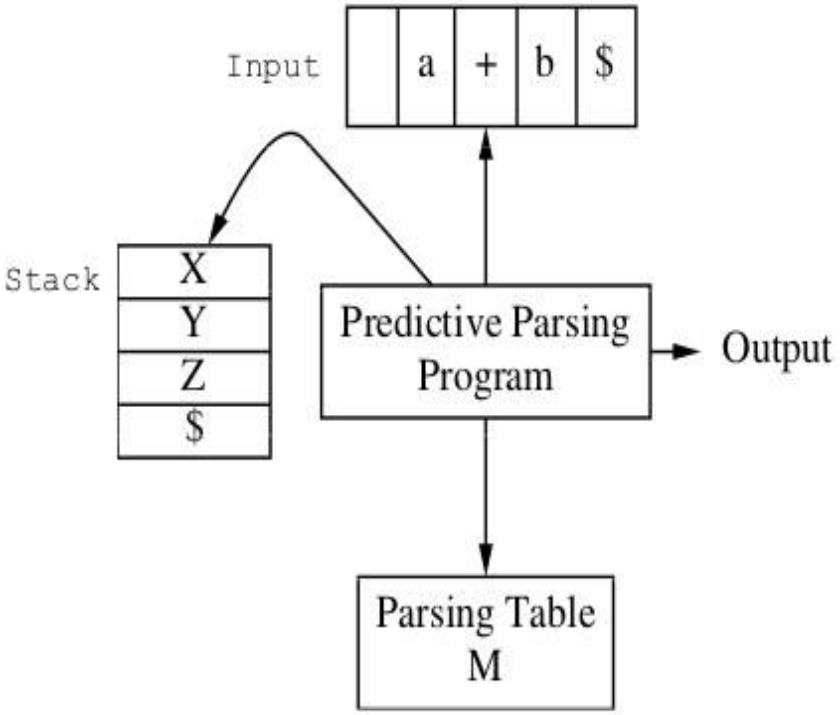
E'()

```
{
   if (lookahead == '+')
        {
            match('+');
            if (lookahead == _i')
            match('i');
            E'();
         }
  }
match(char t)
{
        if (lookhead== t)
        {
            lookahead = getchar();
         }
         else
            printf("Error");
}
```

Predictive LL(1)

Predictive LL(1):

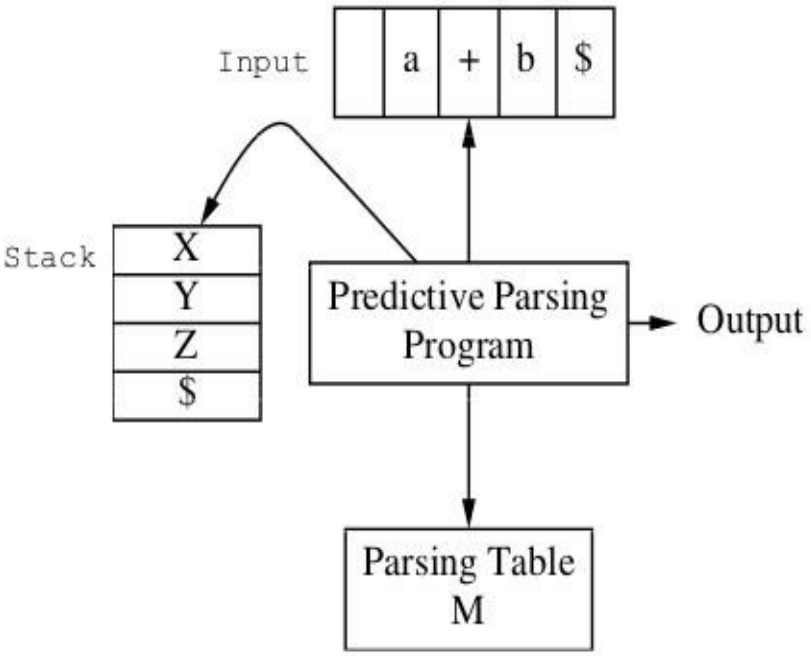
- It is a non recursive top down parser.
- In LL(1), 1st **L** represents that the scanning of the Input from Left to Right.
- Second **L** shows that in this Parsing technique we are going to use Left most Derivation Tree.
- **1** represents the number of look ahead, means how many symbols are going to see when you want to make a decision.
- The predictive parser has an **input, a stack, a parsing table, and an output.**
- The **input contains the string to be parsed, followed by $, the right end marker.**
- The **stack contains a sequence of grammar symbols, preceded by $, the bottom-of stack** marker.
- The Stack holds **left most derivation**.
- The **parsing table is a two dimensional array M[A ,a], where A is a nonterminal, and** a is a terminal or the symbol $.



Predictive LL(1)...

The parser is controlled by a program that behaves as follows:

- The program determines X, the symbol on top of the stack, and _**a**,,the current input symbol.
- These two symbols determine the action of the parser.



There are three possibilities:

1. If **X = a = $,** the parser halts and announces successful completion of parsing.
2. If **X = a ≠ $,** the parser pops X off the stack and advances the input pointer to the next input symbol.
3. If **X is a nonterminal**, the program consults entry M[X, a] of the parsing table M.

This entry will be either an X-production of the grammar or an error entry.

- If M[X, a] = {X → UVW}, the parser replaces X on top of the stack by WVU (with U on top).
- If M[X, a] = error, the parser calls an error recovery routine.

First & Follow

- The Construction of predictive parser is aided by two functions associated with a grammar G.
- These Functions, First and Follow allows us to fill the entries of predictive parsing table for grammar G

FIRST :

Step for finding FIRST:

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

FOLLOW:

Step for finding FOLLOW:

1) FOLLOW(S) = { $ } // where S is the starting Non-Terminal and $ is the input right end marker.
2) if there is a production $A \rightarrow \alpha B \beta$,then everything in FIRST(β) except for $\in$ is placed in FOLLOW(B).
3) if there is a production $A \rightarrow \alpha B$ or a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains $\in$ then everything in FOLLOW(A) is in FOLLOW(B).

First & Follow...

➢**Example 1:**

$E \rightarrow T\ E'$

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

$T \rightarrow F\ T'$

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

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

Then:

$FIRST(E) = FIRST(T) = FIRST(F) = \{(\ , \mathbf{id}\}$

$FIRST(E') = \{+, \varepsilon\}$

$FIRST(T') = \{*, \varepsilon\}$

$FOLLOW(E) = FOLLOW(E') = \{)\ , \$\}$

$FOLLOW(T) = FOLLOW(T') = \{+,), \$\}$

$FOLLOW(F) = \{+, *,), \$\}$

Example 2:

1. $S \rightarrow A\ a$
2. $A \rightarrow B\ D$
3. $B \rightarrow b$
4. $B \rightarrow \epsilon$
5. $D \rightarrow d$
6. $D \rightarrow \epsilon$

$First(S) = \{b, d, a\}$

$First(A) = \{b, d, \epsilon\}$

$First(B) = \{b, \epsilon\}$

$First(D) = \{d, \epsilon\}$

$Follow(S) = \{\$\}$

$Follow(A) = \{a\}$

$Follow(B) = \{d, a\}$

$Follow(D) = \{a\}$

First & Follow...

➢ **Example 3:**

S → A
A → aBA'
A' → dA' / ∈
B → b
C → g

First(S) = First(A) = { a }
First(A) = { a }
First(A') = { d , ∈ }
First(B) = { b }
First(C) = { g }

Follow(S) = { $ }
Follow(A) = { $ }
Follow(A') = { $ }
Follow(B) = { d , $ }
Follow(C) = NA

Example 4:

S → AaAb / BbBa
A → ∈
B → ∈

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

Follow Functions-

Follow(S) = { $ }
Follow(A) = { a , b }
Follow(B) = { a , b }

LL(1) Parsing table Construction

Steps Involved in Predictive parsing table construction:

Step 1: for each production A →α of the grammar do steps 2 &3

Step 2: for each terminal ‗a' in FIRST(**α**) **,** add A →α to M[A , a]

Step 3: if ∈ is in FIRST(**α**) **, add** A →α to M[A , b] for each terminal ‗b' in FOLLOW(A).

Step 4: Make each undefined entry of M be Error.

Construct LL(1) Parsing table for the grammar

G:

E → E+T|T
T → T*F|F
F → id|(E)

After Eliminating Left Recursion

E → TE‟
E‟ → +TE‟|ε
T → FT‟
T‟ → *FT‟|ε
F → id|(E)

and Parse the string id+id*id

Find FIRST and FOLLOW:

NON TERMINAL	FIRST	FOLLOW
E	{ (,id }	{),$}
E'	{+, ε}	{),$}
T	{ (,id }	{+,),$}
T'	{*, ε}	{+,),$}
F	{ (,id }	{*,+,),$}

LL(1) Parsing table Construction

G: $E \rightarrow TE'$

$E' \rightarrow +TE'|\varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT'|\varepsilon$

$F \rightarrow id|(E)$

NON TERMINAL	FIRST	FOLLOW
E	{ (,id }	{),$}
E'	{+, ε}	{),$}
T	{ (,id }	{+,),$}
T'	{*, ε}	{+,),$}
F	{ (,id }	{*,+,),$}

LL(1) Parsing Table:

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

Note: All undefined entries are Errors.

LL(1) Parsing

Parsing the input string "id+id*id" using LL(1) parser:

STACK	INPUT	OUTPUT
$E	id + id * id$	
$E' T	id + id * id$	$E \rightarrow T E'$
$E' T' F	id + id * id$	$T \rightarrow F T'$
$E' T' id	id + id * id$	$F \rightarrow id$
$E' T'	+ id * id$	
$E'	+ id * id$	$T' \rightarrow \varepsilon$
$E' T +	+ id * id$	$E' \rightarrow + T E'$
$E' T	id * id$	
$E' T' F	id * id$	$T \rightarrow F T'$
$E' T' id	id * id$	$F \rightarrow id$
$E' T'	* id$	
$E' T' F *	* id$	$T' \rightarrow * F T'$
$E' T' F	id$	
$E' T' id	id$	$F \rightarrow id$
$E' T'	$	
$E'	$	$T' \rightarrow \varepsilon$
$	$	$E' \rightarrow \varepsilon$

Therefore , LL(1) Parsing is successful

LL(1) Parsing Example

➢ **Show that Grammar is not LL(1).**

S → iEtSS' | a

S' → eS | ϵ

E → b

FIRST(S) = { i, a }

FIRST(S') = {ϵ, e }

FIRST(E) = { b }

FOLLOW(S) = { $, e }

FOLLOW(S') = { $, e}

FOLLOW(E) = { t }

Nonter-minals	a	b	e	i	t	$
S	S→a			S→iEtSS'		
S'			S'→eS S'→ϵ			S'→ϵ
E		E→b				

The entry M[S‟,e] contains multiple entries so the grammar is not LL(1)

LL(1) Parsing Example

- **Construct LL(1) Parsing table for the Grammar and parse string w= int*int**

G: $E \rightarrow T + E \mid T$

$T \rightarrow int \mid int * T \mid (E)$

Given grammar must be converted to Left Factored Grammar

$E \rightarrow T X$

$X \rightarrow + E \mid \varepsilon$

$T \rightarrow (E) \mid int Y$

$Y \rightarrow * T \mid \varepsilon$

	FIRST	FOLLOW
E	{ (, int }	{$,)}
X	{+, ε}	{$,)}
T	{ (, int }	{+,$,)}
Y	{*, ε}	{+,$,)}

	int	*	+	(	)	$
E	E->TX			E->TX		
X			X->+E		X->ε	X->ε
T	T->int Y			T->(E)		
Y		Y->*T	Y->ε		Y->ε	Y->ε

LL(1) Parsing Example

Parsing the string " int*int" using parsing table

	int	*	+	(	)	$
E	E->TX			E->TX		
X			X->+E		X->ε	X->ε
T	T->int Y			T->(E)		
Y		Y->*T	Y->ε		Y->ε	Y->ε

STACK	INPUT	OUTPUT
$E	int*int$	E->TX
$XT	int*int$	T->int Y
$XYint	int*int$	POP
$XY	*int$	Y->*T
$XT*	*int$	POP
$XT	int$	T->int Y
$XYint	int$	POP
$XY	$	Y->ε
$X	$	X->ε
$	$	Accept

Bottom Up Parsing

- Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node.
- we start from a sentence or input string and then apply production rules in reverse manner (reduction) in order to reach the start symbol.
- The process of parsing halts successfully as soon as we reach to start symbol.

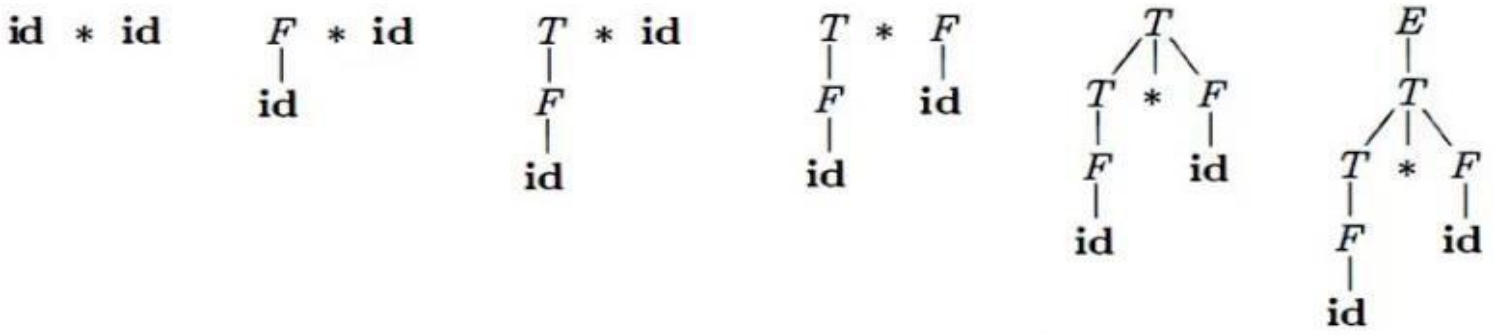
Example:

$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow id$
Input string is **id * id**

Rightmost derivation →

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

Thus this process is like tracing out the right most derivations in reverse.



Bottom Up Parsing

Handle:

- It is a substring that matches the right side of the production and we can reduce such substring by left hand side Non-terminal of production rule.

Example:

G: E → E + E

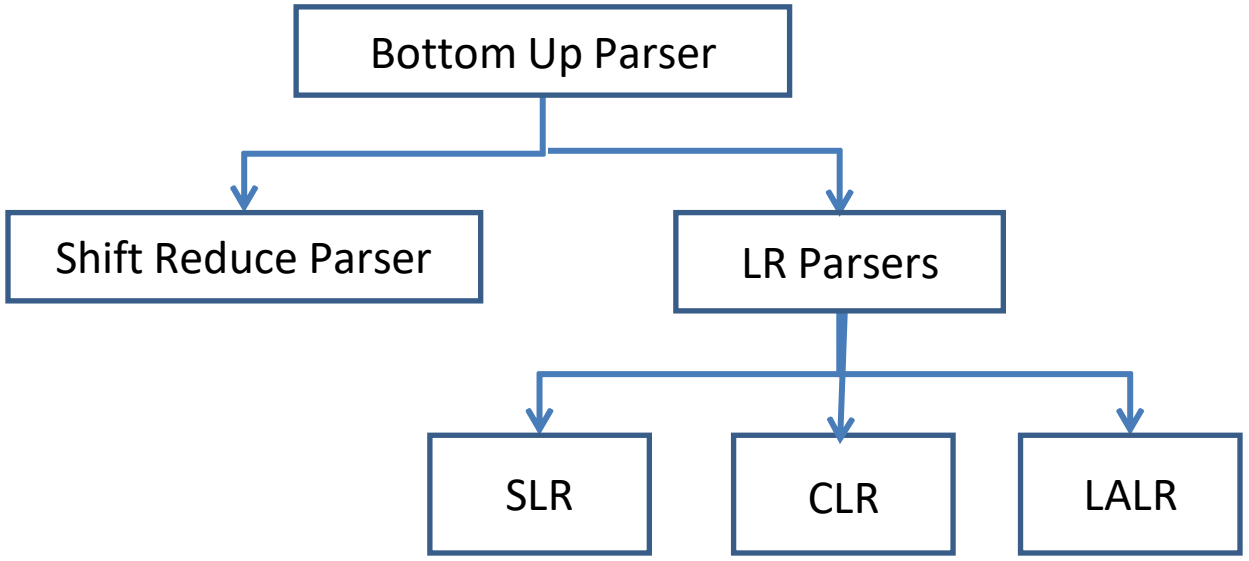
E → E * E

E → (E)

E → id

RIGHT SENTENTIAL FORM	HANDLE	REDUCTION PRODUCTION
$id_1 + id_2 * id_3$	id1	E → id
$E + id_2 * id_3$	id_2	E → id
$E + E * id_3$	id_3	E → id
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
E		

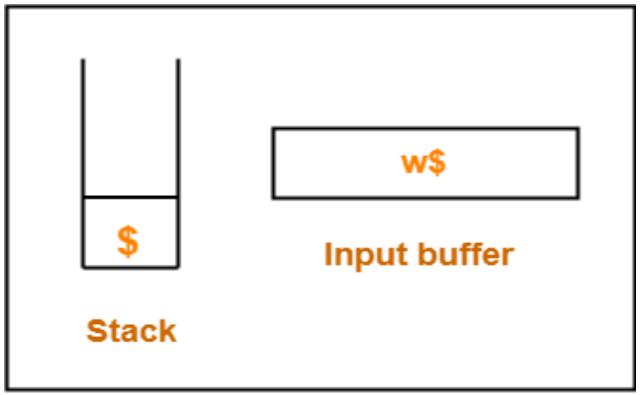
Bottom Up Parsers:



Shift Reduce Parser

SR Parser:

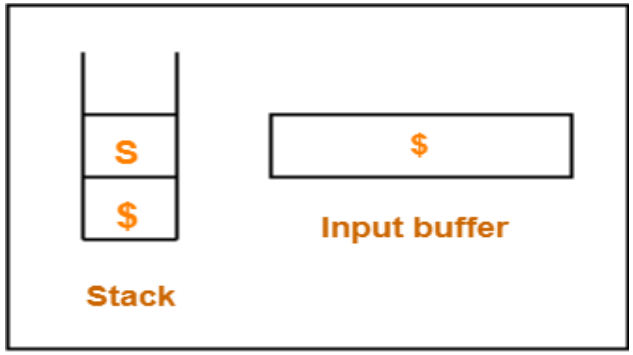
- Shift reduce parsing is a process of reducing a string to the start symbol of a grammar.
- Shift reduce parsing uses a stack to hold the grammar and an input tape to hold the string.
- A shift-reduce parser can possibly make the following four actions:
  1. **Shift:** In a shift action , the next symbol is shifted onto the top of the stack.
  2. **Reduce :** In a reduce action , the handle appearing on the stack top is replaced with the appropriate non-terminal symbol.
  3. **Accept :** In an accept action , the parser reports the successful completion of parsing.
  4. **Error :** In this state , the parser becomes confused and is not able to make any decision . It can neither perform shift action nor reduce action nor accept action.
- Initial Configuration of SR Parser is:
  - Stack contains only the $ symbol.
  - Input buffer contains the input string with $ at its end.
- The parser works by:
  - Moving the input symbols on the top of the stack.
  - Until a handle β appears on the top of the stack , then handle is reduced to LHS of production rule.



Initial Configuration

Shift Reduce Parser...

- Final Configuration of SR Parser:
  - Stack is left with only the start symbol and the input buffer becomes empty.(Successful Parsing)
  - An Error is detected.(Unsuccessful Parsing).
- **Example :** Consider the following grammar-



Final Configuration

S –> S + S
S –> S * S
S –> id and Parse the string **"id + id + id"** using SR Parser.

Stack	Input Buffer	Parsing Action
$	id+id+id$	Shift
$id	+id+id$	Reduce S->id
$S	+id+id$	Shift
$S+	id+id$	Shift
$S+id	+id$	Reduce S->id
$S+S	+id$	Reduce S->S+S
$S	+id$	Shift
$S+	id$	Shift
$S+id	$	Reduce S->id
$S+S	$	Reduce S->S+S
$S	$	Accept

Shift Reduce Parser...

➢ **Example :** Consider the following grammar-

E →E-E

E →E*E

E →id

Parse the input string **id-id*id** using a shift-reduce parser.

STACK	INPUT	ACTION
$	id-id*id$	Shift
$id	-id*id$	Reduce E →id
$E	-id*id$	Shift
$E-	id*id$	Shift
$E-id	*id$	Reduce E →id
$E-E	*id$	Shift
$E-E*	id$	Shift
$E-E*id	$	Reduce E →id
$E-E*E	$	Reduce E →E*E
$E-E	$	Reduce E →E-E
$E	$	Accept

Note:

If the Incoming operator has more priority than in stack operator then perform Shift otherwise perform reduce operation.

Shift Reduce Parser...

➢ **Example :** Consider the following grammar-

$S \rightarrow (L) \mid a$

$L \rightarrow L , S \mid S$

Parse the input string (a , (a , a)) using a shift-reduce parser.

Stack	Input Buffer	Parsing Action
$	(a,(a,a))$	Shift
$(	a,(a,a))$	Shift
$(a	,(a,a))$	Reduce S → a
$(S	,(a,a))$	Reduce L → S
$(L	,(a,a))$	Shift
$(L,	(a,a))$	Shift
$(L,(	a,a))$	Shift
$(L,(a	,a))$	Reduce S → a
$(L,(S	,a))$	Reduce L → S
$(L,(L	,a))$	Shift
$(L,(L,	a))$	Shift
$(L,(L,a	))$	Reduce S → a
$(L,(L,S	))$	Reduce L → L , S
$(L,(L	))$	Shift
$(L,(L)	)$	Reduce S → (L)
$(L,S	)$	Reduce L → L , S
$(L	)$	Shift
$(L)	$	Reduce S → (L)
$S	$	Accept

LR Parser

- LR parser is one type of bottom up parsing, which is used to parse the large class of grammars.
- In the LR(K) parsing,

  Where,

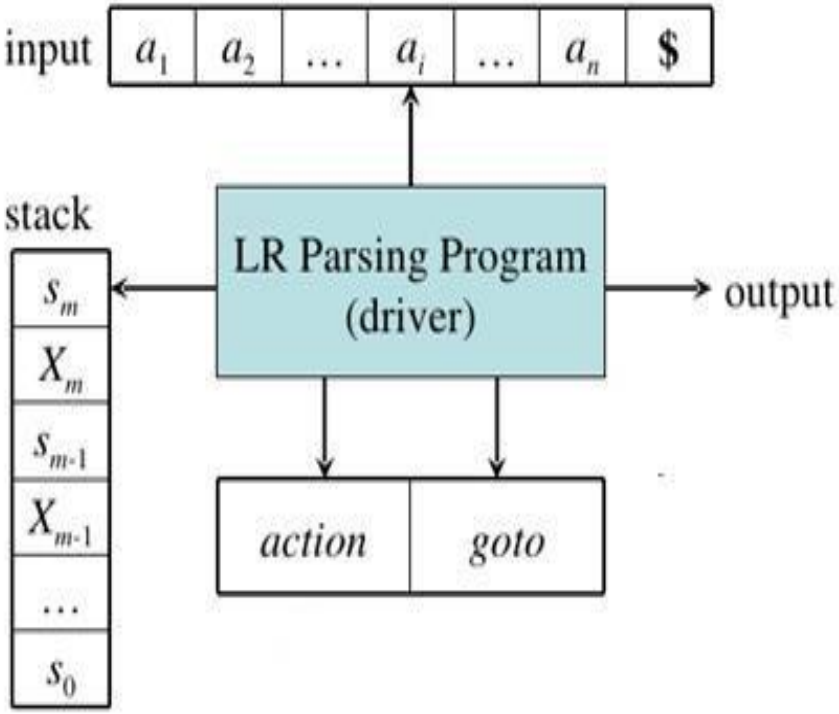
  "L" stands for left-to-right scanning of the input,

  "R" stands for constructing a right most derivation in reverse, and

  "K" is the number of input symbols of the look ahead used to make number of parsing decision.

LR Parser Model:

- It consists of an Input , an Output, a Stack , LR Parser program and a Parsing table which has two parts (Action and Goto).
- Input buffer holds the input ,the parser program reads character from it one at a time.
- The stack holds a sequence of the form $s_0\ X_1\ s_1\ X_2\ s_2\ \ldots\ X_m\ s_m$, where $\mathbf{S_m}$ is on the top.
- Each X_i is a grammar symbol and S_i is a state.



LR Parser...

- The Parser program driving LR Parser behaves as follow:
  - It determines S_m , the state currently on top of stack and a_i ,the current input symbol.
  - It then consults action [S_m, a_i]into the parse table which can have one of the four values:
    1. Shift s, where s is a state.
    2. Reduce by a grammar production A—> β.
    3. Accept(Successful Parsing)
    4. Error.
- A Configuration of LR Parser is a pair whose first component is stack content and Second component is the input.

$$(s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m, \quad a_i a_{i+1} \ldots a_n \$)$$

stack *Input*

- The next move of a parser is determined by reading the S_m , the state currently on top of stack and a_i ,the current input symbol.

LR Parser...

- The parser behaves based on the entry in the parser table.

LR Parser...

- LR Parser behaves as follow(Parsing Process):

1. If action[s_m,a_i] = shift s **then** push current input symbol a_i, and next state s on to the stack, and advance input one position to right:

(s_0 X_1 s_1 X_2 s_2 … X_m s_m a $_i$ s, a_{i+1} … a_n **$**)

2. If action[s_m,a_i] = reduce A → β find r=|β| **then**

pop 2*r symbols, push A, and Then push , the entry for GOTO[s_{m-r}, A], onto the stack :

(s_0 X_1 s_1 X_2 s_2 … X_{m-r} s_{m-r} A s , a_i a_{i+1} … a_n **$**)

3. If action[s_m,a_i] = accept,parsing is successful.

4. If action[s_m, a_i] = error **then** attempt recovery.

Types of LR Parsers:

1. SLR(1) Parser
2. CLR(1) Parser
3. LALR(1) Parser

- All the above parsers will follow the same parsing process.

SLR Parser...

LR(0) Items:

- An **LR (0) item** is a production with dot at some position on the right side of the production.
- **LR(0) items** is useful to indicate that how much of the input has been scanned up to a given point in the process of parsing.
- For example, production T → T * F leads to four LR(0) items:

  T → · T * F

  T → T · * F

  T → T * · F

  T → T * F ·

- That production $A \rightarrow \varepsilon$ has one item $[A \rightarrow \bullet]$

Augmented Grammar:

- If G is a grammar with start symbol S then G‘, the augmented grammar for G, is the grammar with new start symbol S‘ and a production S‘ -> S.
- The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of input.

  Example: G: S -> AA

  A -> aA | b

  The augmented grammar for the above grammar will be

  G‘: S‘ -> S

  S -> AA , A -> aA | b

SLR Parser...

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 every item in I is added to closure(I).
2. If A -> α . Bβ is in closure(I) and B -> γ is a production then add the item B -> .γ to I, If it is not already there. We apply this rule until no more items can be added to closure(I).

Example:

G‟: S‘ ->S

S ->AA

A ->aA/b

Closure(S‘ ->S)= S‘ ->.S

S ->.AA

A ->.aA/.b

Closure(S ->AA)=S ->.AA

A ->.aA/.b

Closure(A ->aA/b)= A ->.aA/.b

SLR Parser...

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

Example:

I ={ E' → .E, E → .E+T, E → .T,
T → .T*F, T → .F,
F → .(E), F → .id }

goto(I,E) = { E' → E., E → E.+T }

goto(I,T) = { E → T., T → T.*F }

goto(I,F) = {T → F. }

goto(I,() = { F → (.E), E → .E+T, E → .T, T → .T*F, T → .F,
F → .(E), F → .id }

goto(I,id) = { F → id. }

Construction of The Canonical LR(0) Collection

- To create the SLR parsing tables for a grammar G, we will create the canonical LR(0) collection of the grammar G'.

- ***Algorithm***:

  C is { closure({S'→ • S}) }

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

  for each I in ***C*** and each grammar symbol X

  if goto(I,X) is not empty and not in ***C***

  add goto(I,X) to ***C***

- goto function is a DFA on the sets in C.

SLR Parser...

Canonical LR(0) Items:

Construct SLR Parsing Table and Parse the String id*id + id

G:

$E \rightarrow E+T|T$

$T \rightarrow T*F|F$

$F \rightarrow (E) \mid id$

Augmented Grammar G‟:

$E' \rightarrow E$

$E \rightarrow E+T$

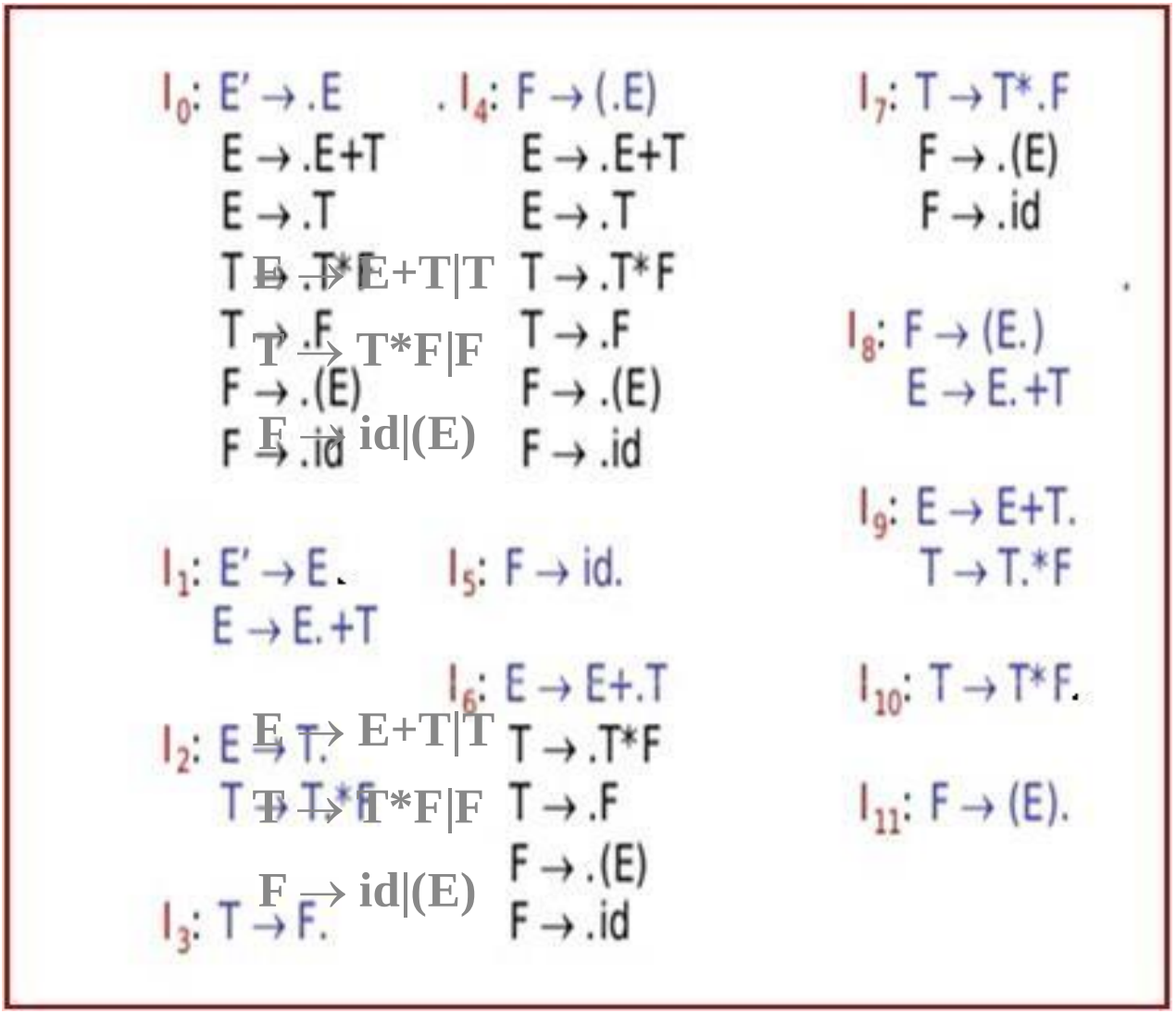
$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

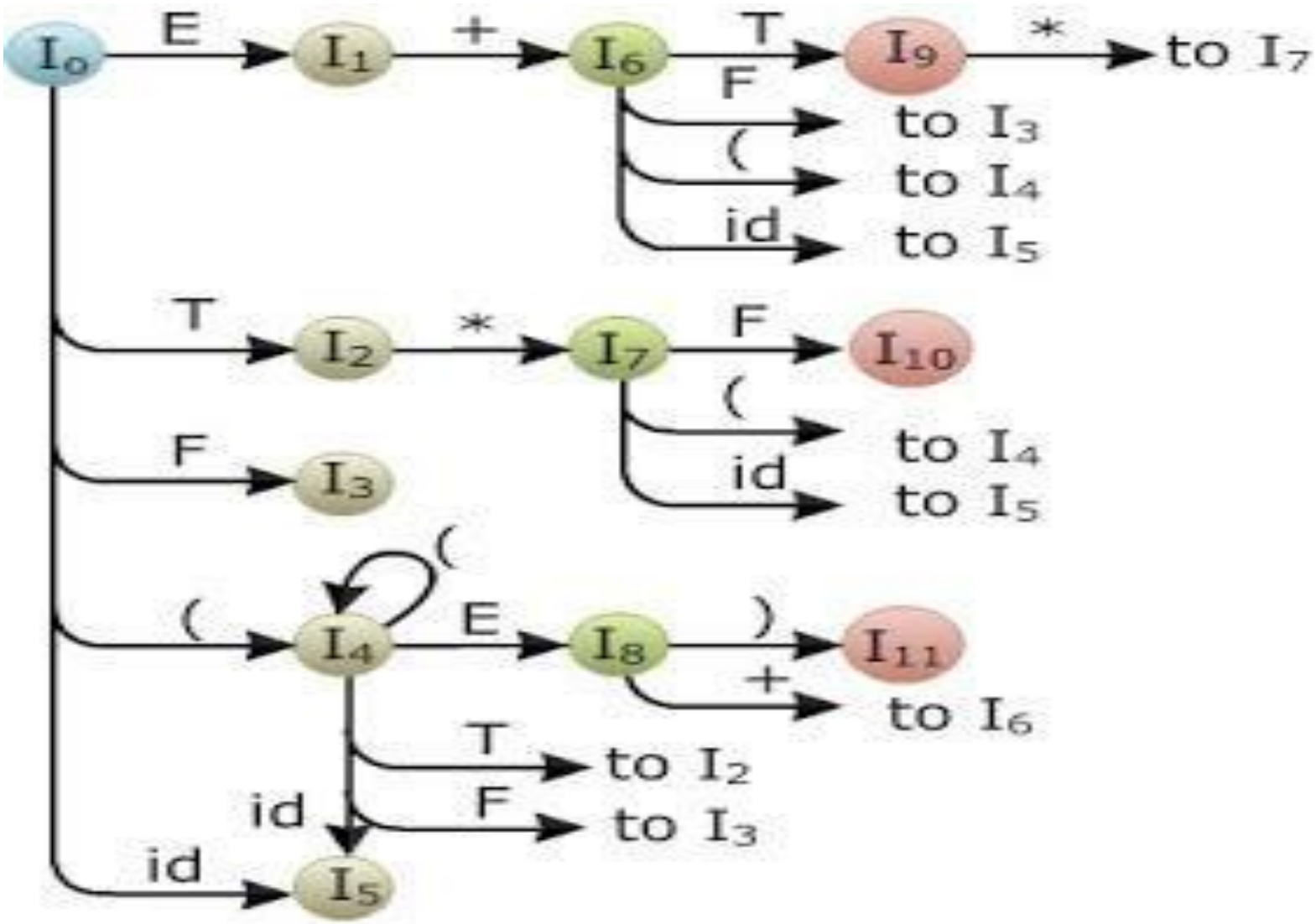
$F \rightarrow (E)$

$F \rightarrow id$



SLR Parser...

Goto Graph:



SLR Parser...

Construction of SLR Parsing Table:

1. Construct the canonical collection of sets of LR(0) items for G'.
   $C \leftarrow \{I_0,...,I_n\}$

2. Create the parsing action table as follows
   - If a is a terminal, $A \rightarrow \alpha.a\beta$ in I_i and $goto(I_i,a)=I_j$ then action[i,a] is ***shift j***.
   - If $A \rightarrow \alpha.$ is in I_i, then action[i,a] is ***reduce*** $A \rightarrow \alpha$ for all a in FOLLOW(A) where $A \neq S'$.
   - If $S' \rightarrow S.$ is in I_i, then action[i,$] is ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not SLR(1).

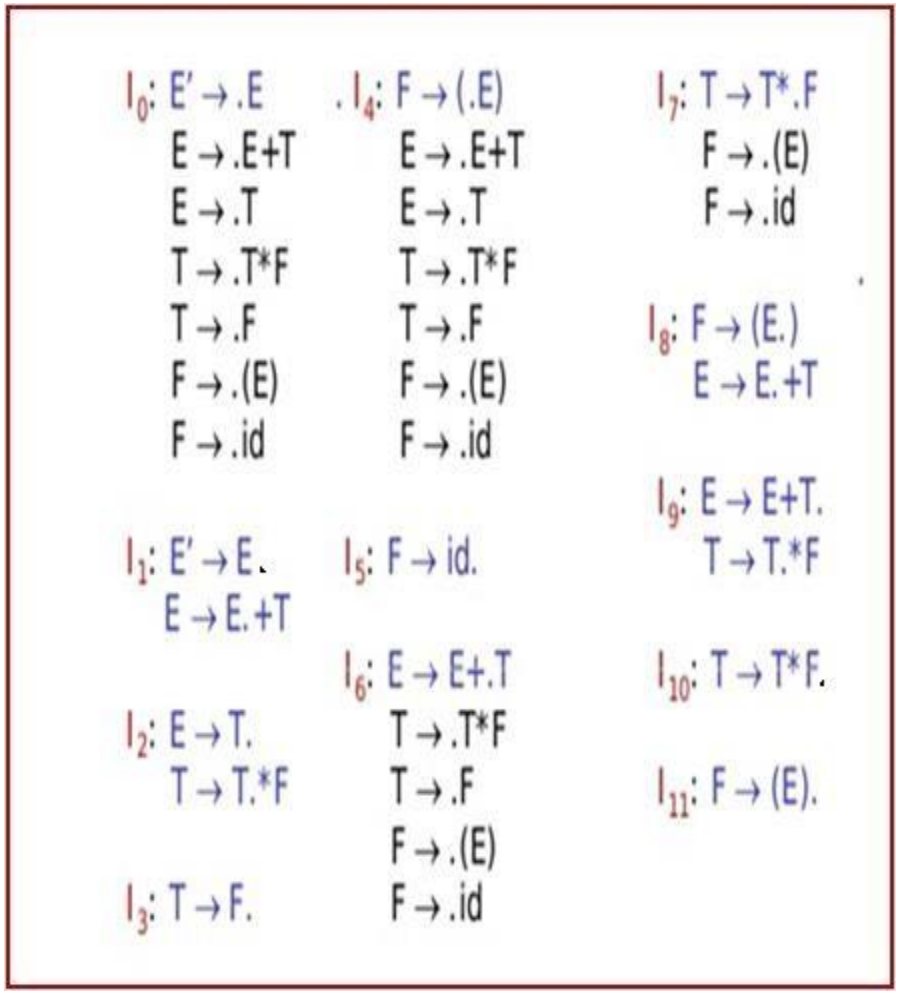
3. Create the parsing goto table
   - for all non-terminals A, if $goto(I_i,A)=I_j$ then goto[i,A]=j

4. All entries not defined by (2) and (3) are errors.

5. Initial state of the parser contains $S' \rightarrow .S$

SLR Parser...

SLR Parsing Table:



Action Table Goto Table

state	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

SLR Parser...

SLR Parsing:

1. $E \rightarrow E+T$
3. $T \rightarrow T*F$ 4. $T \rightarrow F$
5. $F \rightarrow (E)$ 6. $F \rightarrow id$

	Action Table						Goto Table		
state	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

stack	input	action
0	id*id+id$	shift 5
0id5	*id+id$	reduce by F→id
0F3	*id+id$	reduce by T→F
0T2	*id+id$	shift 7
0T2*7	id+id$	shift 5
0T2*7id5	+id$	reduce by F→id
0T2*7F10	+id$	reduce by T→T*F
0T2	+id$	reduce by E→T
0E1	+id$	shift 6
0E1+6	id$	shift 5
0E1+6id5	$	reduce by F→id
0E1+6F3	$	reduce by T→F
0E1+6T9	$	reduce by E→E+T
0E1	$	accept

CLR Parser

- CLR represents canonical LR Parser.
- The Grammar used for constructing this parser is called as CLR Grammar or LR(1) Grammar.
- This Parser uses LR(1) items to represent the states of the parser.
- The LR(1) items are of the form **[A →α.Xβ , a]** which is having two components.
- LR(1) item = LR(0) item + lookahead.
- The first component is an LR(0) item indicates that up to what position in the grammar rule parsing is completed.
- Second component is a terminal or $, which represents the actual follow.

Closure operation LR(1) Items:

1. Start with *closure*(*I*) = *I*
2. If $[A \rightarrow \alpha \bullet B\beta, a] \in closure(I)$ then for each production $B \rightarrow \gamma$ in the grammar and each terminal $b \in \text{FIRST}(\beta a)$, add the item $[B \rightarrow \bullet\gamma, b]$ to *I* if not already in *I*
3. Repeat 2 until no new items can be added

Goto operation LR(1) Items :

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

CLR Parser...

Construction of canonical set of LR(1) Items:

C is { closure({S'→.S,$}) }

repeat the followings until no more set of LR(1) items can be added to ***C***.

  for each I in ***C*** and each grammar symbol X

    if goto(I,X) is not empty and not in ***C***

      add goto(I,X) to ***C***

➢ **Example:**

Construct the CLR Parsing table and parse the string ―adad‖ for the Grammar **G:**

S → CC

C → cC

C → d

Augmented Grammar G‟: S‘ → S

S → CC

C → cC

C → d

CLR Parser...

G“: S‘ → S

S → CC

C → aC

C → d

I_0: S‘→.S , $

S→.CC, $

C→.aC , a / d

C→.d , a / d

I_1:goto(I_0,S)

S‘→S. , $

I_2: goto(I_0,C)

S→ C.C, $

C→.aC , $

C→.d , $

I_3: goto(I_0,a)

C→ a.C , a / d

C→.aC , a / d

C→.d , a / d

I_4: goto(I_0,d)

C→d. , a / d

I_5: goto(I_2,C)

S→ CC. , $

I_6: goto(I_2,a)

C→ a.C , $

C→.aC , $

C→.d , $

I_7: goto(I_2,d)

C→d. , $

I_8: goto(I_3,C)

C→aC. , a / d

I_9: goto(I_6,C)

C→aC. , $

A →α.Bβ , a

First(β*a*)

CLR Parser...

G‘: S‘ → S
S → CC
C → aC
C → d

I_0: S‘→.S , $
S→.CC, $
C→.aC , a / d
C→.d , a / d

I_1:goto(I_0,S)
S‘→S. , $

I_2: goto(I_0,C)
S→ C.C, $
C→.aC , $
C→.d , $

I_3: goto(I_0,a)
S→ a.C , a / d
C→.aC , a / d
C→.d , a / d

A →α.Bβ , a
First(β*a*)

I_4: goto(I_0,d)
C→d. , a / d

I_5: goto(I_2,C)
S→ CC. , $

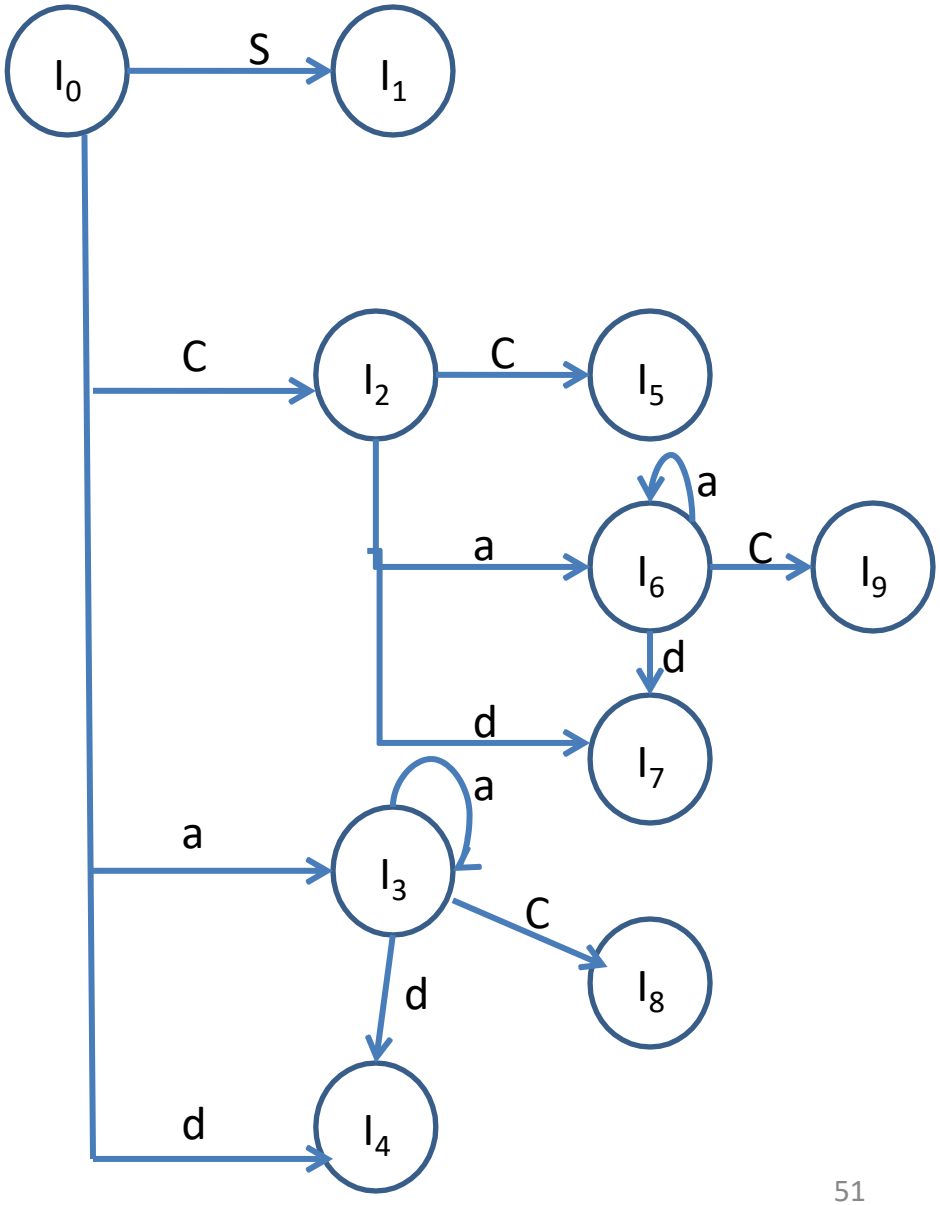
I_6: goto(I_2,a)
S→ a.C , $
C→.aC , $
C→.d , $

I_7: goto(I_2,d)
C→d. , $

I_8: goto(I_3,C)
C→aC. , a / d

I_9: goto(I_6,C)
C→aC. , $

Goto Graph:



CLR Parser...

LR(1) Parsing table construction:

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

CLR Parser...

CLR Parsing table :

G‘: S‘ → S
S → CC
C → aC
C → d

I_0: S‘→.S , $
S→.CC, $
C→.aC , a / d
C→.d , a / d

I_1:goto(I_0,S)
S‘→S. , $

I_2: goto(I_0,C)
S→ C.C, $
C→.aC , $
C→.d , $

I_3: goto(I_0,a)
C→ a.C , a / d
C→.aC , a / d
C→.d , a / d

I_4: goto(I_0,d)
C→d. , a / d

I_5: goto(I_2,C)
S→ CC. , $

I_6: goto(I_2,a)
C→ a.C , $
C→.aC , $
C→.d , $

I_7: goto(I_2,d)
C→d. , $

I_8: goto(I_3,C)
C→aC. , a / d

I_9: goto(I_6,C)
C→aC. , $

STATE	ACTION			GOTO	
	a	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

CLR Parser...

CLR Parsing table :

STATE	ACTION			GOTO	
	a	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Parsing the string "adad":

Stack	Input	Action
$0	adad$	S3
$0a3	dad$	S4
$0a3d4	ad$	Reduce C -> d
$0a3C8	ad$	Reduce C -> aC
$0C2	ad$	S6
$0C2a6	d$	S7
$0C2a6d7	$	Reduce C -> d
$0C2a6C9	$	Reduce C -> aC
$0C2C5	$	Reduce S -> CC
$0S1	$	accept

LALR Parser

- LALR stands for LookAhead LR Parser.
- The LALR parsing table construction is same as CLR parsing table construction,only at the the set of LR(1) items having same core components i.e. similar first components are detected and merged together as a single state in the parsing table.
- In this parsing method ,the parse table is considerably smaller than the CLR Parsing table.

Example:

In CLR example, the items **(I_3 , I_6) , (I_4 , I_7) and (I_8 , I_9) have similar core components.**

I_3 and I_6 are merged as I_{36}

I_{36} :

S→ a.C , a / d /$

C→.aC , a / d /$

C→.d , a / d /$

I_4 and I_7 are merged as I_{47}

I_{47} : C→d. , a / d /$

I_8 and I_9 are merged as I_{89}

I_{89} : C→aC. , a / d /$

I_0: S'→.S , $

S→.CC, $

C→.aC , a / d

C→.d , a / d

I_1:goto(I_0,S)

S'→S. , $

I_2: goto(I_0,C)

S→ C.C, $

C→.aC , $

C→.d , $

I_5: goto(I_2,C)

S→ CC. , $

LALR Parser...

LALR Parsing table :

STATE	ACTION			GOTO	
	a	*d*	$	*S*	*C*
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Parsing the string “adad”:

Stack	Input	Action
$0	adad$	S36
$0a36	dad$	S47
$0a36d47	ad$	Reduce C -> d
$0a36C89	ad$	Reduce C -> aC
$0C2	ad$	S36
$0C2a36	d$	S47
$0C2a36d47	$	Reduce C -> d
$0C2a36C89	$	Reduce C -> aC
$0C2C5	$	Reduce S -> CC
$0S1	$	accept

Error recovery in parsing

What should happen when your parser finds an error in the user‘s input?

- Stop immediately and signal an error .
- Record the error but try to continue.

Error Recovery Strategies:

1. Panic Mode
2. Phrase Level
3. Error Productions
4. Global Correction

1. Panic Mode:

- When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to synchronizing tokens .
- Typical synchronizing tokens are delimiters, such as a semicolon, opening or closing parenthesis.
- Simplest method to implement.
- When multiple errors in the same statement are rare, this method is quite adequate.

2. Phrase Level :

- On discovering an error, a parser may perform local correction on the remaining input.
- For example, it may replace a prefix of the remaining input by some string that allows the parser to continue.

Error recovery in parsing

- A typical local correction would be to:
  - Replace a comma by a semicolon,
  - Delete an extraneous semicolon, or
  - Insert a missing semicolon.
- Major drawback: Situations in which the actual error has occurred before the point of detection.

3.Error Productions :

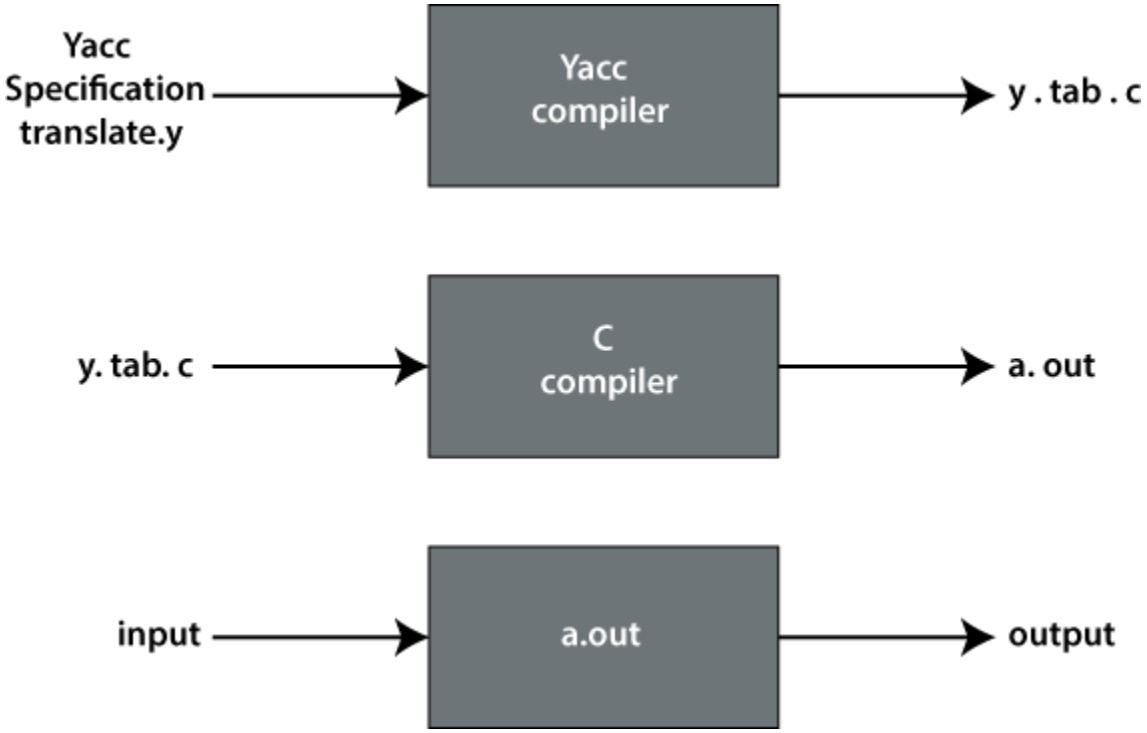
- If we have a good idea of the common errors then augment the grammar with error productions that generate the erroneous constructs.
- Use the grammar augmented by these error productions to construct a parser.
- If an error production is used by the parser, generate an appropriate error diagnostic message.

4. Global Correction:

- The parser examines the whole program and tries to find out the closest match for it which is error free.
- When an erroneous input (statement) X is fed, it creates a parse tree for some closest error-free statement Y.
- This may allow the parser to make minimal changes in the source code, but due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.

YACC –Automatic Parser Generator

- YACC stands for **Yet Another Compiler Compiler**.
- It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar.
- The input of YACC is the rule or grammar, and the output is a C program.



- The Unix command transforms the YACC specification file **translate.y** into a C program called **y.tab.c,** which is a representation of LALR parser written in C.

YACC –Automatic Parser Generator...

- By compiling y.tab.c along with the ly library, we will get the desired object program a.out that performs the operation defined by the original YACC program.
- A YACC source program contains three parts:

Declarations

%%

Translation rules

%%

Supporting C rules

Declarations Part:

- This part of YACC has two sections; both are optional.
- The first section has ordinary C declarations, which is delimited by %{ and %}.
- This section contains only the include statements .
- In second section we can declare the grammar tokens. Ex %token DIGIT
- Token declared in this section can be used by second and third part of YACC specification.

YACC –Automatic Parser Generator...

Translation rules:

- This part contains translation rules and associated semantic actions.
- This part is enclosed between %% &%%.

A set of productions:

<head> -> <body1> | <body2> | ….. | <body n>

would be written in YACC as

```
<head>   :   <body1>      {<semantic action>1}
         |   <body2>      {<semantic action>2}
             …..
         |   <body n>     {<semantic action>n}
         ;
```

- The semantic action of YACC is a set of C statements. In a semantic action, the symbol $$ considered to be an attribute value associated with the head‘s non-terminal.
- While $i considered as the value associated with the ith grammar production of the body.

Supporting C rules:

- The third part of a YACC Specification consists of supporting C- routines.
- A lexical analyzer by the name yylex() must be provided.

YACC –Automatic Parser Generator...

Example:

```
%{
#include <ctype.h>
%}
%token DIGIT
%%
line    : expr _\n‘          { printf(—%d\n‖, $1); }
        ;
expr    : expr _+‘ term      { $$ = $1 + $3; }
        |  term
        ;
term    : term _*‘ factor    { $$ = $1 * $3; }
        | factor
        ;
factor  : _(_ expr ‘)‘       { $$ = $2; }
        |  DIGIT
        ;
%%
```

```
yylex()
{
int c;
c = getchar();
if (isdigit(c)) {
yylval = c-_0‘;
return DIGIT;
}
return c;
}
```

Compiler Design

UNIT – III:

Semantic analysis: Intermediate forms of source Programs – abstract syntax tree, polish notation and three address codes. Attributed grammars, Syntax directed translation,

Conversion of popular Programming languages language Constructs into Intermediate code forms, Type checker.

Semantic analysis:

- Semantic Analysis is the third phase of Compiler.
- It makes sure that declarations and statements of program are semantically correct.
- Both syntax tree of previous phase and symbol table are used to check the consistency of the given code.
- It gathers type information and stores it in either syntax tree or symbol table. This type information is subsequently used by compiler during intermediate-code generation.
- Type checking is an important part of semantic analysis .
- Errors recognized by semantic analyzer are as follows:
  - Type mismatch
  - Undeclared variables
  - Reserved identifier misuse

Semantic Analysis

Functions of Semantic Analysis:

- **Type Checking :**
  Ensures that data types are used in a way consistent with their definition.
- **Label Checking:**
  A program should contain labels references.
- **Flow Control Check:**
  Keeps a check that control structures are used in a proper manner.(Example: no break statement outside a loop).

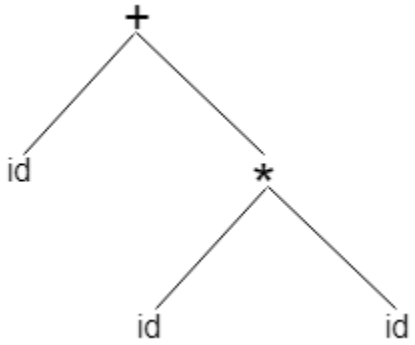
Example:

```
float x = 10.1;
float y = x*30;
```

In the above example integer 30 will be type casted to float 30.0 before multiplication, by semantic analyzer.

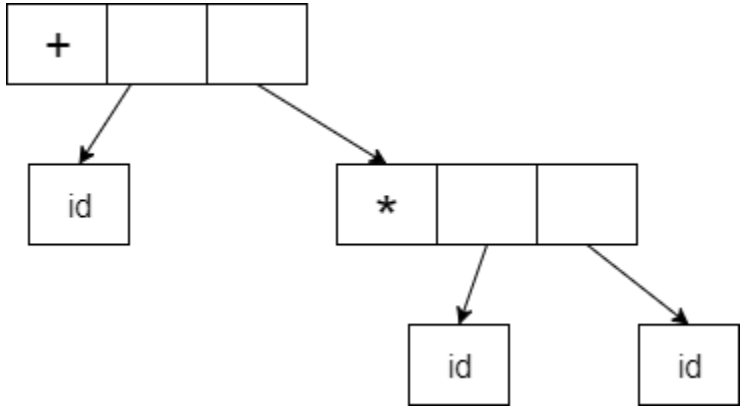
Intermediate forms of source Program

- An Intermediate source form is an internal form of a program created by compiler while translating the source program from high level language to assembly level or machine level code.
- Intermediate representation of source program can be done using:
  I. Abstract Syntax Tree
  II. Postfix Notation
  III. Three Address Code



Abstract Syntax Tree:

- It is a tree structure representation of the abstract syntactic structure of source code written in a programming language.
- Each node of a tree denotes a construct occurring in the source code.
- This hierarchal structure consists of operands in leaf nodes and operators in the interior nodes.
- The operator that will be evaluated first is placed near the bottom of the tree.
- The operator that will be evaluated at end is placed at the root of the tree.



id+id*id

Intermediate forms of source Program

Postfix Notation:

- It is a notation form for expressing arithmetic, logic and algebraic equations.
- Its most basic distinguishing feature is that operators are placed on the right of their operands.
- It is a linearised form of the syntax tree.
- Syntax tree can be converted into a postfix notation and vice versa.

Example – The postfix representation of the expression

Infix notation: (a – b) * (c + d) + (a – b)

Postfix notation : ab – cd + *ab -+

Three Address Code:

- Three-address code is used to represent an intermediate code.
- Three address code is a sequence of statements of the general form:

x=y op z

- Each instruction in three address code consist of
  - At most three addresses or operands
  - At most one operator to represent an expression excluding the assignment operator
  - Value computed at each instruction is stored in temporary variable generated by compiler.

Intermediate forms of source Program

Example:

a= (-c * b) + (-c * d)

Three address code is :

t_1 = -c

t_2 = b*t_1

t_3 = -c

t_4 = d * t_3

t_5 = t_2 + t_4

a = t_5

Implementation of Three Address Code:

There are 3 representations of three address code:

1. Quadruple
2. Triples
3. Indirect Triples

Quadruple:

- It is a record structure consists of 4 fields namely op, arg1, arg2 and result.
- op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

Intermediate forms of source Program

- The contents of fields arg1, arg2 and result are pointers to the symbol table entries for the names represented by these fields.
- Temporary names must be entered into symbol tables as they are created.

Example :

a = – c*b + – c*b

#	Op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	t1	b	t2
(2)	uminus	c		t3
(3)	*	t3	b	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

Quadruple representation

Triples:

- This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used.
- It consist of only three fields namely op, arg1 and arg2.
- The fields arg1 and arg2 are either pointers to symbol table or pointers into the triple structure.

Intermediate forms of source Program

Example :

a = – c*b + – c*b

#	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

Triples representation

Indirect Triples:

- This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

Example :

a = – c*b + – c*b

#	Op	Arg1	Arg2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

List of pointers to table

#	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Indirect Triples representation

Attribute Grammar

- Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information.
- A finite, possibly empty set of attributes is associated with each distinct symbol in the grammar.
- Each attribute has well-defined domain of values, such as integer, float, character, string, etc.
- It is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language.
- It can pass values or information among the nodes of a parse tree.

Example:

$E \rightarrow E + T$ { E.value = E.value + T.value }

Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.

- Based on the way the attributes get their values, they can be broadly divided into two categories :
  - **I. Synthesized Attributes**
  - **II. Inherited Attributes.**

Attribute Grammar...

Synthesized attributes:

- These attributes get values from the attribute values of their child nodes.

  Ex: S → ABC

- If S is taking values from its child nodes (A,B,C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.
- As in our previous example (E → E + T), the parent node E gets its value from its child node.
- Synthesized attributes never take values from their parent nodes or any sibling nodes.

Inherited attributes:

- In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings.

  Ex: S → ABC

- A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

SDT & SDD

Syntax Directed Translation:

- In syntax directed translation scheme embeds program fragments called semantic actions within the production bodies.

  Ex: E->E+T { print'+'}

  F->id { print id.val}

- Semantic Actions are enclosed within the curly braces.

Syntax Directed Definition:

- In syntax directed definition, the grammar is associated with some notations called as semantic rules.
- Grammar + semantic rule = SDD
- In SDD Grammar symbols are associated with attributes and productions are associated with semantic rules.
- **num.lexval** is the attribute returned by the lexical analyzer.

Example:

Production	Semantic Rules
E → E + T	E.val := E.val + T.val
E → T	E.val := T.val
T → T * F	T.val := T.val + F.val
T → F	T.val := F.val
F → (F)	F.val := F.val
F → num	F.val := num.lexval

S-Attributed and L-Attributed Definition

S-Attributed Definition:

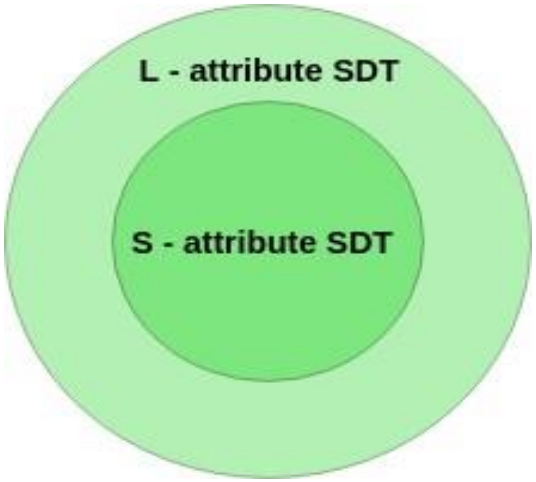
- If an SDT uses only synthesized attributes, it is called as S-attributed SDT.
- S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.
- Semantic actions are placed in rightmost place of RHS.

L-attributed SDT:

- If an SDT uses both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from left siblings only, it is called as L-attributed SDT.
- Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.
- Semantic actions are placed anywhere in RHS.

For example:

A -> XYZ {Y.S = A.S, Y.S = X.S, Y.S = Z.S} is not an L-attributed grammar since Y.S = A.S and Y.S = X.S are allowed but Y.S = Z.S violates the L-attributed SDT definition as attributed is inheriting the value from its right sibling.



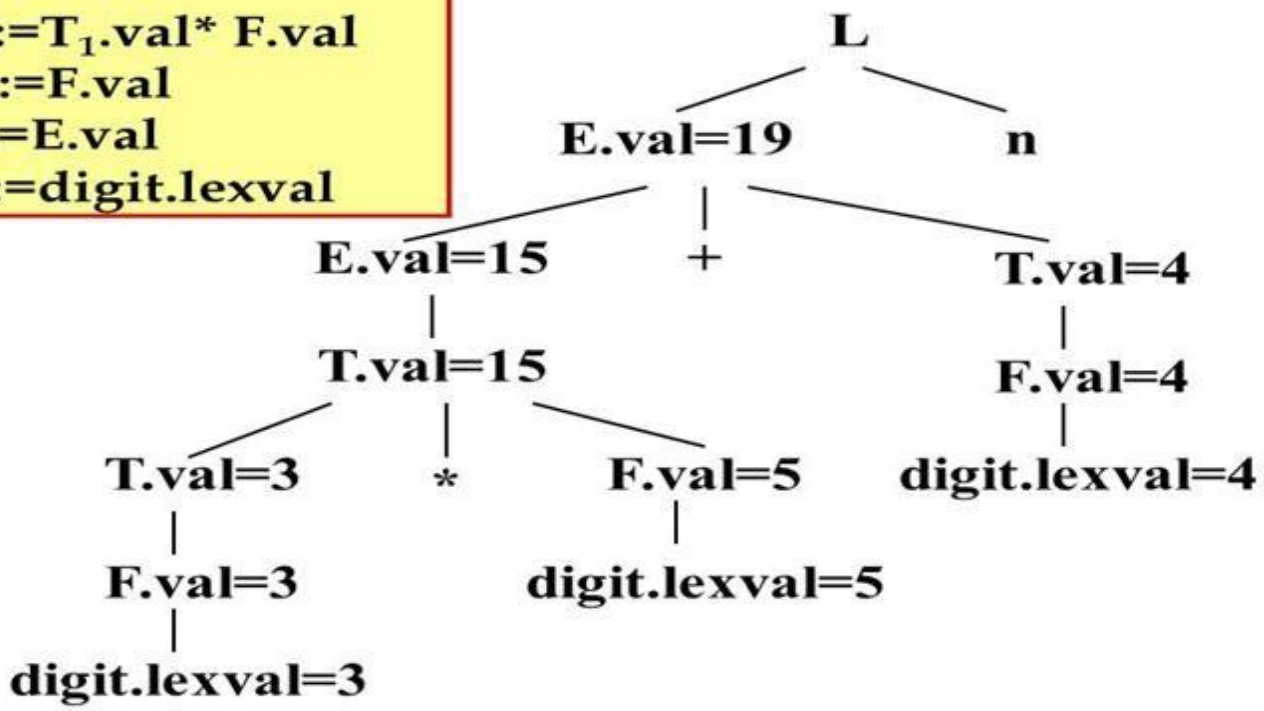
Note: If a definition is S-attributed, then it is also L-attributed but not vice-versa.

Annotated Parse Tree

- An **Annotated Parse Tree** is a **parse tree** showing the values of the attributes at each node.
- The process of computing the attribute values at the nodes is called annotating or decorating the **parse tree**.

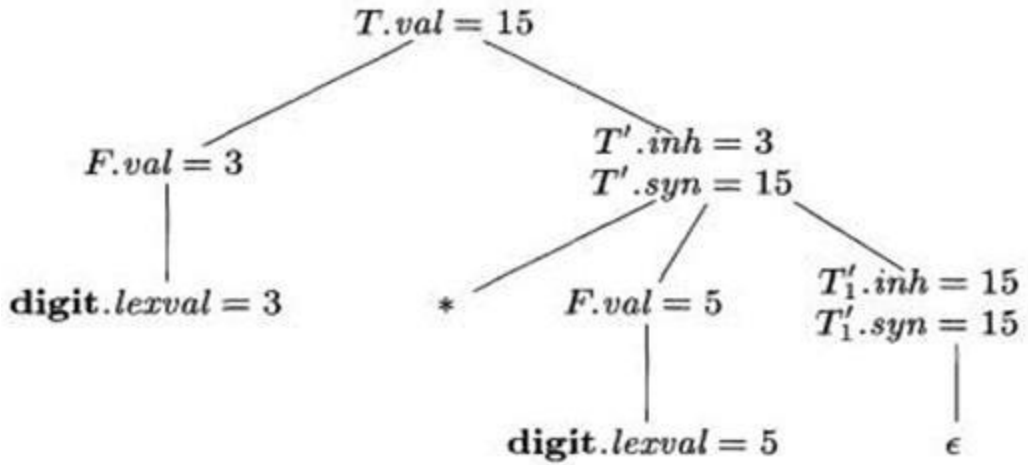
Prod.	Semantic Rules
L→En	print(E.val)
$E \rightarrow E_1+T$	E.val := E_1.val+T.val
E→T	E.val :=T.val
$T \rightarrow T_1*F$	T.val :=T_1.val* F.val
T→F	T.val :=F.val
F→ (E)	F.val :=E.val
F→digit	F.val :=digit.lexval

Annotated Parse Tree for 3*5+4 **n**



Annotated Parse Tree...

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow * F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

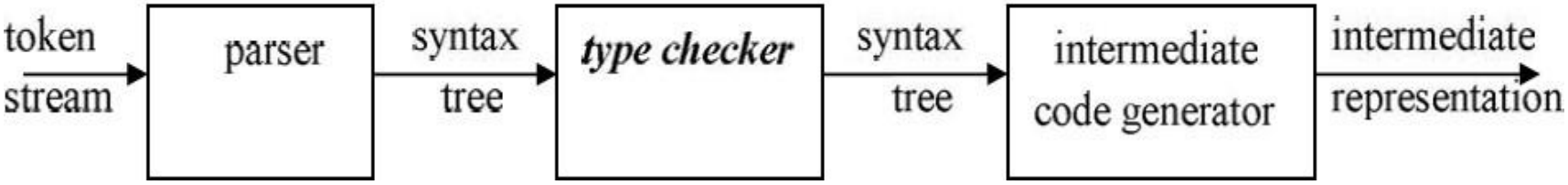


Annotated parse tree for $3 * 5$

Type Checking

- Type checking is the process of verifying that each operation executed in a program respects the type system of the language.
- There are two types of type checking:
    1. Static Type Checking
    2. Dynamic Type checking
- **Static type checking** is performed during compile time , it means that the type of a variable is known at compile time.
- For some languages, the programmer must specify what type each variable is (e.g C, C++, Java)
- In Static Typing, variables generally are not allowed to change types.
- **Dynamic type checking** is performed at runtime.
- For example, Python is a dynamically typed language. It means that the type of a variable is allowed to change over its lifetime. Other dynamically typed languages are -Perl, Ruby, PHP, JavaScript etc.

Position of type checker



Type Checking...

Type checking of Expressions:

In the following rules, the attribute *type* for E gives the type expression assigned to the expression generated by E.

1. $E \rightarrow$ **literal** { E.*type* : = *char* }
   $E \rightarrow$ **num** { E.*type* : = *integer* }
   Here, constants represented by the tokens **literal** and **num** have type *char* and *integer*.

2. $E \rightarrow$ **id** { E.*type* : = *lookup* (**id**.*entry*) }
   lookup (e) is used to fetch the type saved in the symbol table entry pointed to by e.

3. $E \rightarrow E_1$ **mod** E_2 { *E.type* : = **if** E_1. *type* = *integer* **and**
   E_2. *type* = *integer* **then** *integer*
   else *type_error* }
   The expression formed by applying the mod operator to two subexpressions of type integer has type integer; otherwise, its type is *type_error*.

4. $E \rightarrow E_1$ [E_2] { *E.type* : = if E_2.*type* = *integer* **and**
   E_1.*type* = *array(s,t)* **then** *t*
   else *type_error* }
   In an array reference E_1 [E_2] , the index expression E_2 must have type integer. The result is the element type *t* obtained from the type *array(s,t)* of E_1.

Type Checking...

Type checking of Statements:

Statements do not have values; hence the basic type *void* can be assigned to them. If an error is detected within a statement, then *type_error* is assigned.

Translation scheme for checking the type of statements:

1. Assignment statement:

$S \rightarrow$ **id** : = E { S.*type* : = **if** **id**.*type* = E.*type* **then** *void*
else *type_error* }

2. Conditional statement:

$S \rightarrow$ **if** E **then** S_1 { S.*type* : = **if** E.*type* = *boolean* **then** S_1.*type*
else *type_error* }

3. While statement:

$S \rightarrow$ while E do S_1 { S.*type* : = **if** E.*type* = *boolean* **then** S_1.*type*
else *type_error* }

4. Sequence of statements:

$S \rightarrow S_1 ; S_2$ { S.*type* : = **if** S_1.*type* = *void* and
S_1.*type* = *void* **then** *void*
else *type_error* }

Type Checking...

Type checking of Functions:

The rule for checking the type of a function application is :

$E \rightarrow E_1 (E_2)$ { $E.type$: = **if** $E_2.type = s$ **and**
$E_1.type = s \rightarrow t$ **then** t
else *type_error* }

****

Compiler Design

UNIT – IV:

Symbol Tables: Symbol table format, organization for block structures languages, hashing, tree structures representation of scope information. Block structures and non block structure storage allocation: static, Runtime stack and heap storage allocation, storage allocation for arrays, strings and records.

Code optimization: Consideration for Optimization, Scope of Optimization, local optimization, loop optimization, frequency reduction, folding, DAG representation.

Symbol Table:

- Symbol table is an important data structure used in a compiler.
- Symbol table is used to store the information about the occurrence of various entities such as objects, classes, variable name, interface, function name etc.
- it is used by both the analysis and synthesis phases.

Symbol table is used by various phases of compiler as follows :-

- **Lexical Analysis:** Creates new table entries in the table, example like entries about token.
- **Syntax Analysis:** Adds information regarding attribute type, scope, dimension, line of reference, use,etc in the table.

Symbol Table

- **Semantic Analysis:** Uses available information in the table to check for semantics i.e. to verify that expressions and assignments are semantically correct(type checking) and update it accordingly.
- **Intermediate Code generation:** Refers symbol table for knowing how much and what type of run- time is allocated and table helps in adding temporary variable information.
- **Code Optimization:** Uses information present in symbol table for machine dependent optimization.
- **Code generation:** Generates code by using address information of identifier present in the table.

Symbol Table Operations:

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

Symbol Table...

Symbol Table Format:

- Symbol table consists of names and its properties like type , values, size ,scope etc..
- There are two types of name representations:
  1. Fixed Length Name
  2. Variable Length Name

Name	Properties/Attributes

1. Fixed Length Name Representation:

- A fixed space for each name is allocated in symbol table.
- In this type of storage, if name is too small then there is wastage of space.
- The name can be referred by pointer to symbol table entry.

Example:

SUM, A, PI, MAX are the variables that are stored in the symbol table . Memory space is wasted in case of variables A and PI as their length is less than name field in the symbol table.

Name			Properties/Attributes
S	U	M	
A			
P	I		
M	A	X	

Symbol Table...

2. Variable Length Name Representation:

- A fixed space is not allocated for name in the symbol table.
- The name is stored with the help of starting index and length of each name.

Example:

- Instead of storing the names SUM, A, B and MAX in the symbol table directly , these name are stored in an array and they are separated with delimiter.
- The staring index of each name in the array and its length including delimiter is stored in the name field of symbol table.

Name		Properties/Attributes
Starting Index	Length	
0	4	
5	2	
6	2	
8	4	

0	1	2	3	4	5	6	7	8	9	10	11
S	U	M	$	A	$	B	$	M	A	X	$

Symbol Table...

Organization for Block Structures Languages:

- The block structured language is a kind of language in which sections of source code is within some matching pair of delimiters such as "{" and "}" or begin and end.
- Such a section gets executed as one unit or one procedure or a function or it may be controlled by some conditional statements (if, while, do-while).
- Normally, block structured languages support structured programming approach Example: C, C++, JAVA, ALGOL,PASCAL etc.
- Non-block structured languages does not contain any blocks ,Examples are LISP, FORTRAN and SNOBOL.

Implementation of Symbol Table:

The following data structures are used for organization of block structured languages:

1. Linear List
2. Self-Organizing List
3. Hashing
4. Tree Structure

Symbol Table...

1. Linear List:

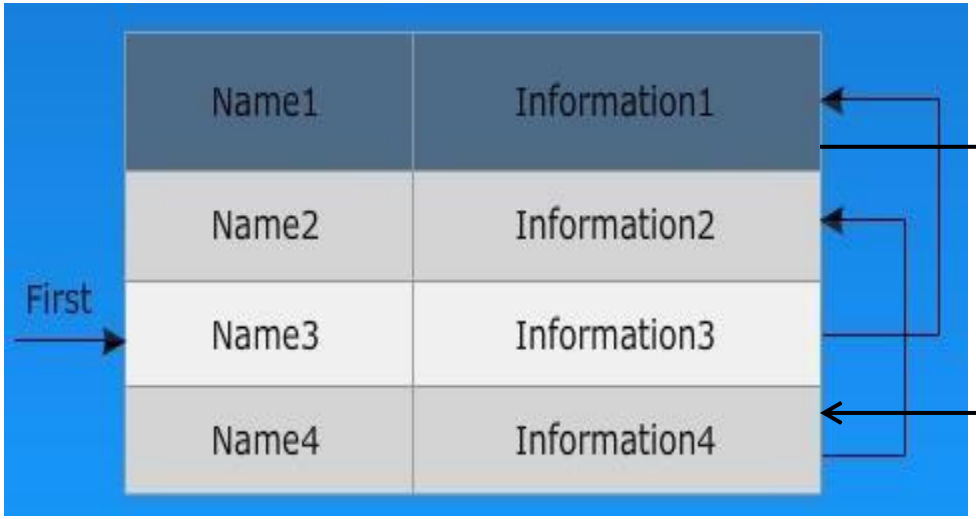
- Linear list of records is the easiest way to implement the symbol table.
- In this method, an array is used to store names and associated information.
- The new names are added to the symbol table in the order they arrive.
- The pointer "available" is maintained at the end of all stored records.
- To retrieve the information about some name we start from beginning of array and go on searching up to available pointer. If we reach at pointer available without finding a name we get an error "use of undeclared name".
- While inserting a new name we should ensure that it is not already present. If it is already present then another error occurs, i.e., "Multiple Defined Name".

Name 1	Information 1
Name 2	Information 2
.	.
.	.
.	.
.	.
Name n	Information n

Symbol Table...

2. Self Organizing List:

- In this method, symbol table is implemented using linked list.
- A link field is added to each record.
- We search the records in the order pointed by the link of link field.
- A pointer "First" is maintained to point to first record of the symbol table
- When the name is reference or created, it is moved to the front of the list.
- The most frequently referred names will tend to be at the front of the list. Hence, access time to most frequently referred names will be the least.

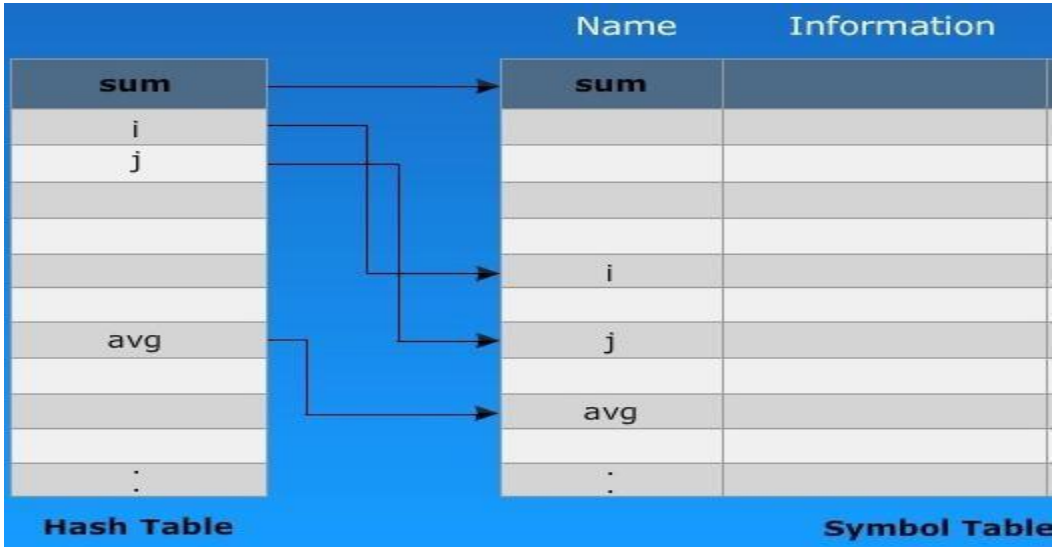


- The names are referenced in the order as Name3,Name1,Name4 and Name2.

Symbol Table...

3. Hashing:

- Hashing is an important technique used to search the records of symbol table.
- In hashing scheme, two tables are maintained – hash table and symbol table
- The hash table consists of K entries from 0, 1, 2, … to K-1. These entries are basically pointers to symbol table pointing to the names of symbol table.
- To determine whether the 'Name' is in symbol table, we use a hash function 'h' such that h (name) will result any integer between 0 to K-1. We can search any name by Position = h (name).
- Using the position we can obtain the exact locations of name in symbol table.
- The hash table and symbol table are shown below:



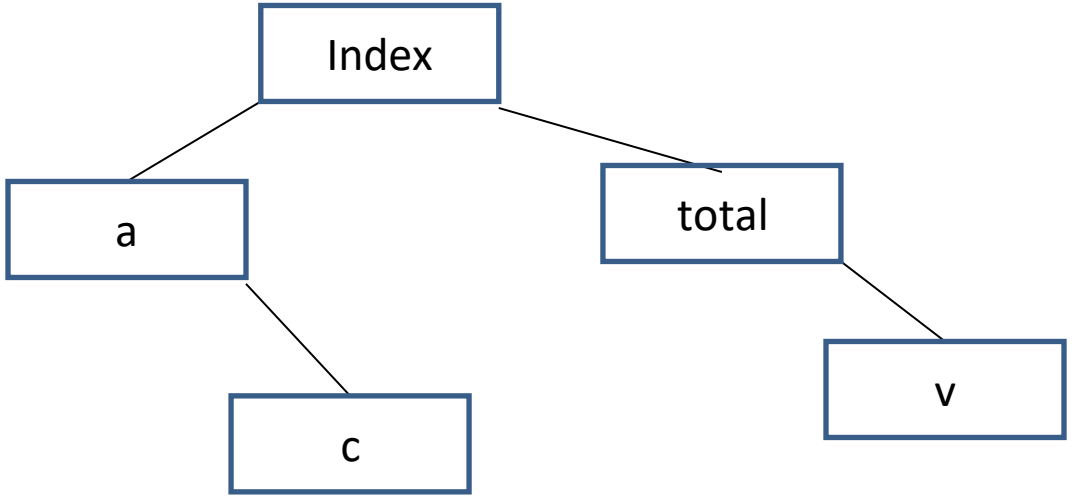
Symbol Table...

4. Tree Structure:

- When the scope information is presented in hierarchical manner then it forms a tree structure representation which is an efficient approach for symbol table organization.
- This organization uses binary search tree for storing the names in symbol table.
- We add two links left and right in each record in the search trees.
- Whenever a name is to be added first, the name is searched in the tree.
- If it does not exist then a record for new name is created and added at the proper position.
- Each node of tree has following format:

Left Child	Name of Symbol	Information	Right Child

- Example: variables such as Index , a, total ,c , v are organized as follow:



Block structures and Non Block structure storage allocation

- Storage allocation refers to process of mapping the data code into appropriate location in the main memory.
- Compiler must carry out the storage allocation and provide access to variables and data.
- Storage allocation strategies are:

1. Static Storage Allocation

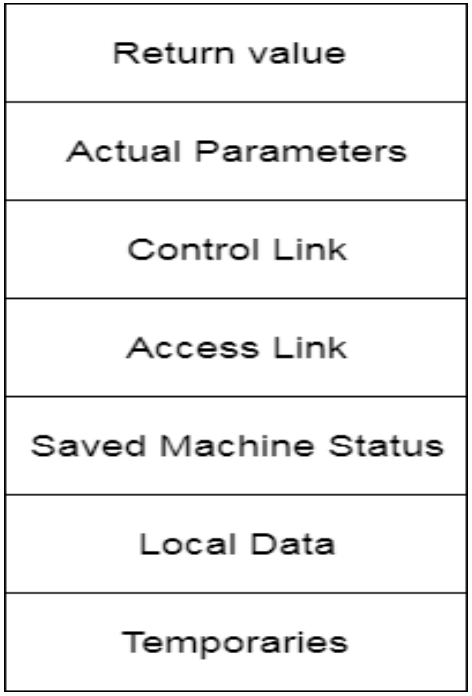
- For any program if we create memory at compile time, memory will be created in the static area.
- For any program if we create memory at compile time only, memory is created only once.
- It don't support dynamic data structure i.e memory is created at compile time and deallocated after program completion.
- The drawback with static storage allocation is recursion is not supported.
- Another drawback is size of data should be known at compile time
- Eg- FORTRAN was designed to permit static storage allocation.

II. Stack Storage Allocation

- Stack allocation is a procedure in which stack is used to organize the storage.
- The stack used in stack allocation is known as control stack.
- In this type of allocation, creation of data objects is performed dynamically.
- In this activation records are created for the allocation of memory.

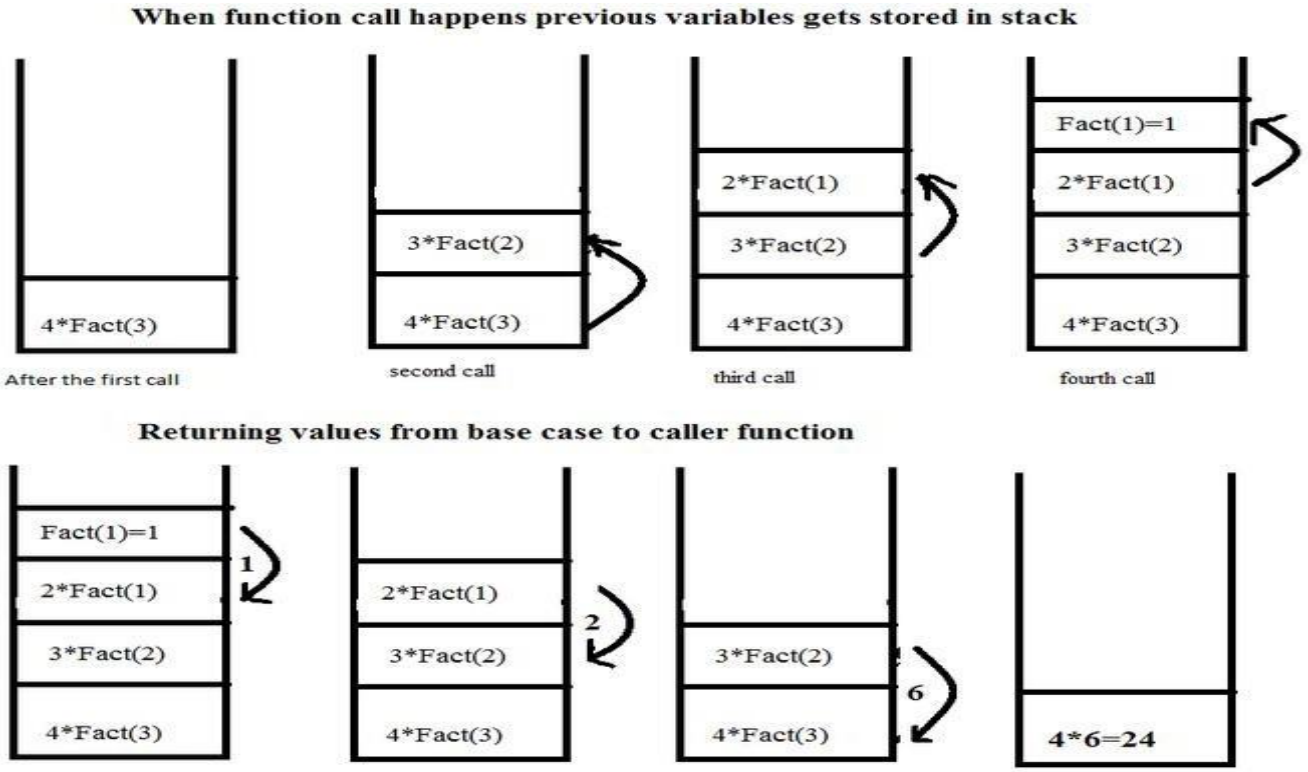
Block structures and Non Block structure storage allocation...

- These activation records are pushed onto the stack using Last In First Out (LIFO) method.
- Locals are stored in the activation records at run time and memory addressing is done by using pointers and registers .
- Recursion is supported in stack allocation.
- **Activation record contains 7 fields :**



1. **Return Value:** It is used by calling procedure to return a value to calling procedure.
2. **Actual Parameter:** It is used by calling procedures to supply parameters to the called procedures.
3. **Control Link:** It is an optional field .It points to activation record of the caller. It also known as dynamic link field.
4. **Access Link:** It is an optional field . It is used to refer to non-local data held in other activation records. It also known as static link field.
5. **Saved Machine Status:** It holds the information about status of machine before the procedure is called.
6. **Local Data:** It holds the data that is local to the execution of the procedure.
7. **Temporaries:** It stores the value that arises in the evaluation of an expression.

Block structures and Non Block structure storage allocation...



III. Heap Allocation:

- Heap is a contiguous memory , Heap allocation is an allocation procedure in which heap is used to manage the allocation of memory.
- Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required.
- Size of Heap-memory is quite larger as compared to the Stack-memory.
- Heap-memory is accessible or exists as long as the whole application runs.
- It maintains a linked list for the free blocks and reuse the deallocated space using best fit.

Local Optimization Techniques

If the scope of the optimization is limited to certain specific block of statements then it is called as **local optimization**.

- Common Sub Expression Elimination
- Copy Propagation
- Dead Code Elimination
- Constant Folding
- Loop optimization techniques
  - Code Motion / Frequency Reduction
  - Induction variable Elimination
  - Reduction in Strength

Common Sub Expression Elimination:

It is a compiler optimization technique of finding redundant expression evaluations, and replacing them with a single computation . This saves the time overhead resulted by evaluating the expression for more than once .

Before

```
t6 := 4*i
x := a[t6]
t7 := 4*i
t8 := 4*j
t9 := a[t8]
a[t7] := t9
t10 := 4*j
a[t10] := x
goto B2
```

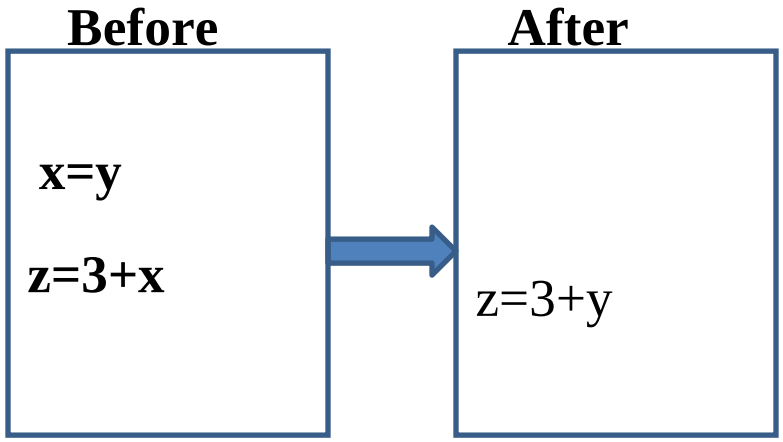
After

```
t6 := 4*i
x := a[t6]
t8 := 4*j
t9 := a[t8]
a[t6] := t9
a[t8] := x
goto B2
```

Copy Propagation

It is the process of replacing the occurrences of targets of direct assignments with their values. A direct assignment is an instruction of the form x = y , which simply assigns the value of y to x .

Example:



Dead Code Elimination

Code that is unreachable or that does not affect the program can be eliminated.

Example :

```
 Function1()
{
int a=10,b=20,c,d;
c=a+b;
d=b/a’
print(c);
return;
print(d);  // Dead Code
}
```

Here, the value of d will not print and function will return

Constant Folding

Constant folding is the process of recognizing and evaluating constant expressions at compile time rather than computing them at runtime.

Example:

Before:

X=10+20*3/2;

After

X=40;

If an Expression contains all the literals ,they must be folded to a single value.

Loop Optimization Techniques:

➢ **Code Motion/ Frequency Reduction:**

Moving the code outside the loop, whose value does not change for all the iterations .

Example:

Original code	Optimized code

`t = 10;`
 	
`for(i=0;i<N;i++)`
`{`
`  a = t * 10;`
`  b = 20 * i;`
`  y+= a * b;`
`}`
	`t = 10;`
	`a = t * 10;`
	
	`for(i=0;i<N;i++)`
	`{`
	`  b = 20 * i;`
	`  y+= a * b;`
	`}`

➢Induction Variable Elimination:

A variable is said to be Induction variable, if the value of a variable changes for every iteration in side the loop i.e. increase or decrease with fixed value . if the loop contains such variables then we have to eliminate or minimize such variables inside the loop.

Example:

```
int a[SIZE]; int b[SIZE];
void f (void)
{
 int i1, i2, i3;
for (i1 = 0, i2 = 0, i3 = 0; i1 < SIZE; i1++)
            a[i2++] = b[i3++];
            return;
}
```

The code fragment below shows the loop after induction variable elimination.

```
int a[SIZE]; int b[SIZE];
void f (void)
{
 int i1;
for (i1 = 0; i1 < SIZE; i1++)
            a[i1] = b[i1];
return;
}
```

➢ Reduction in Strength :

It is an loop optimization technique in which expensive operations are replaced with equivalent and less expensive operations.

Exponent is replaced with multiplication ,multiplication is replaced with addition in order reduce the strength of an expression.

Example:

Before

```
C=8;
for(i=0;i<=10;i++)
{
A[i]=c*i;
}
```

After

```
C=8;
K=0;
for(i=0;i<=10;i++)
{
A[i]=k;
K=k+c;
}
```

Directed Acyclic Graph (DAG):

Directed Acyclic Graph (DAG) is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG is used to represent the flow graph.

DAG consists of :

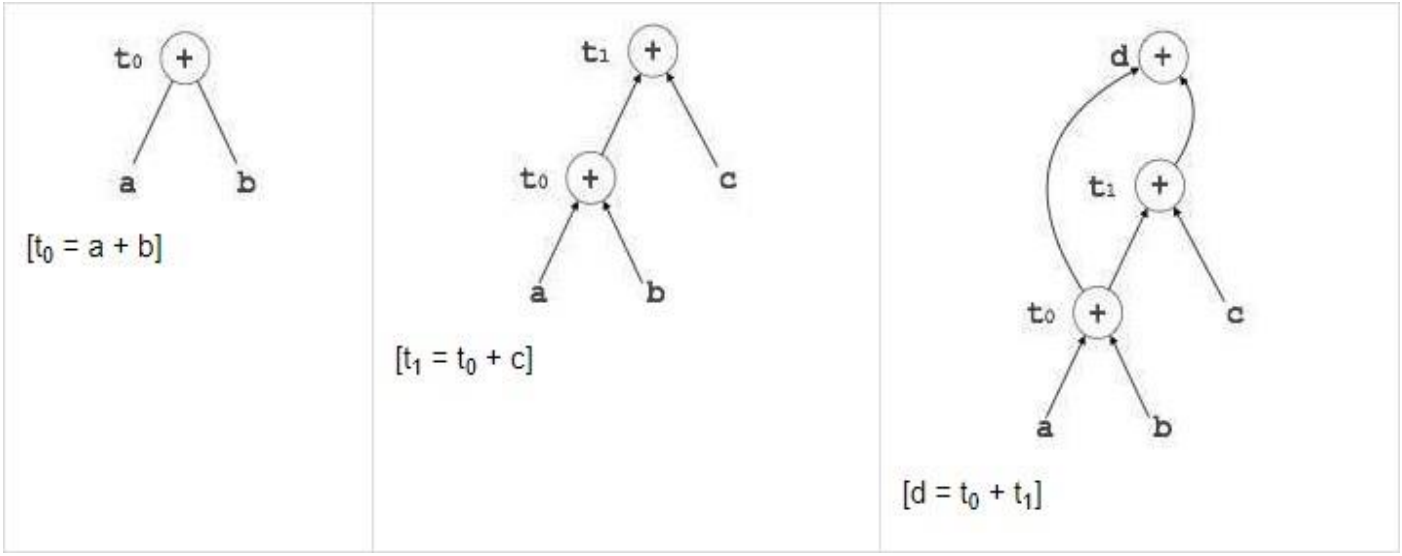
- Leaf nodes represent identifiers, names or constants.
- Interior nodes represent operators.
- Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

Example:

$t_0 = a + b$

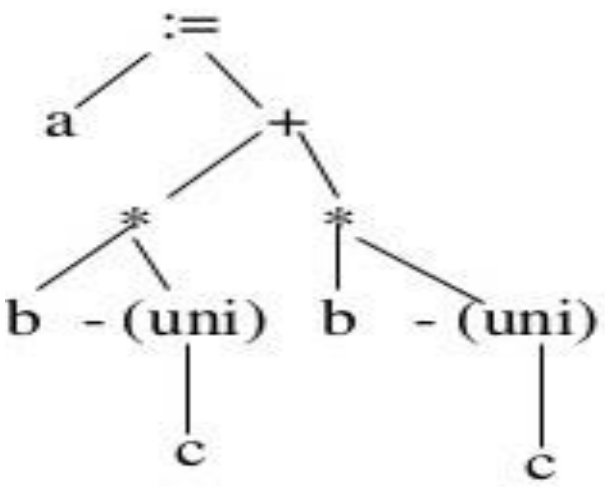
$t_1 = t_0 + c$

$d = t_0 + t_1$

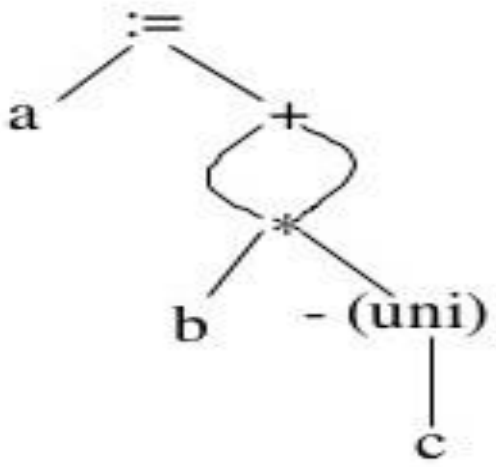


AST and DAG:

a := b *-c + b*-c



AST



DAG

Compiler Design

UNIT – V:

Data flow analysis: Flow graph, data flow equation, global optimization, redundant sub expression elimination, Induction variable elements, Live variable analysis, Copy propagation.

Object code generation: Object code forms, machine dependent code optimization, register allocation and assignment generic code generation algorithms, DAG for register allocation.

Basic block: Basic block is a set of statements that always executes in a sequence one after the other.

The characteristics of basic blocks are:

- There is no possibility of branching or getting halt in the middle.
- All the statements execute in the same order they appear without losing the flow control of the program.

Example:

Basic block

```
(1) T1 = b + c
(2) T2 = T1 + d
(3) a = T2
```

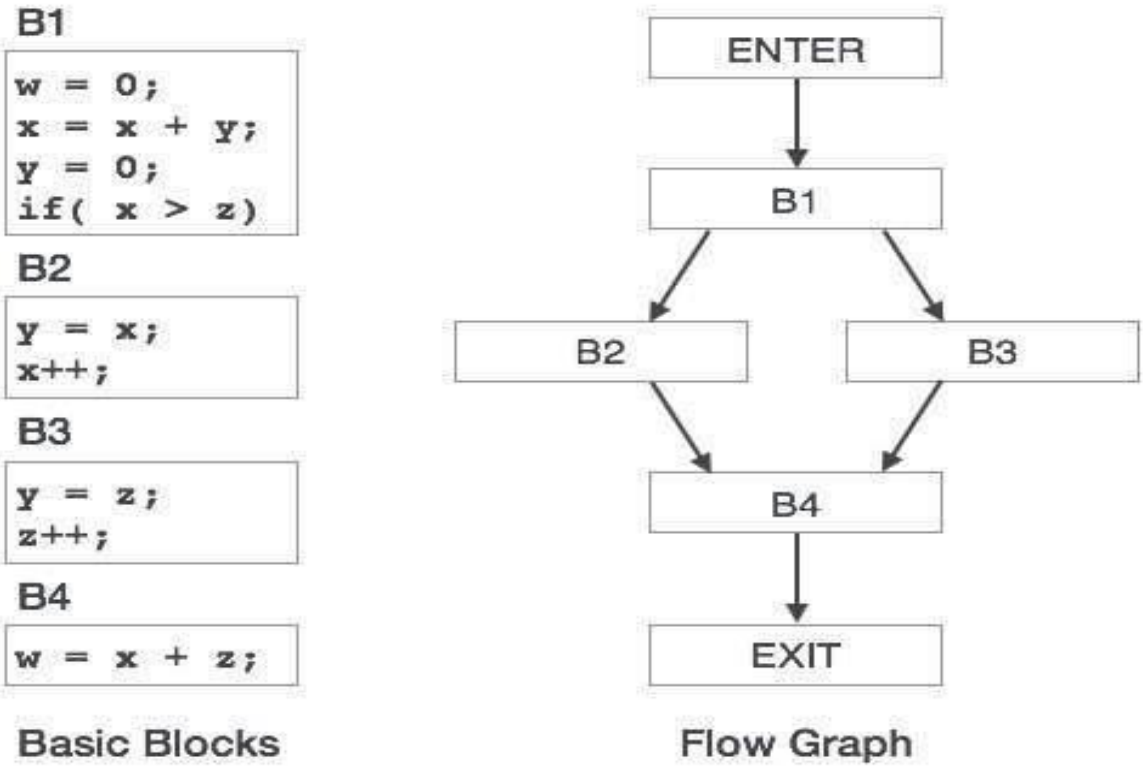
Not a Basic block

```
(1) If A<B goto (4)
(2) T1 = 0
(3) goto (5)
(4) T1 = 1
(5)
```

Flow Graph :

- A graph representation of three-address statements, called a flow graph.
- A flow graph consists of set of basic blocks and edges .
- Edges represents the flow of information between the basic blocks.
  and block represents computations.
- It is used for data flow analysis through which we can achieve the global optimization.
- We can construct a flow graph for given three address code.

Example:

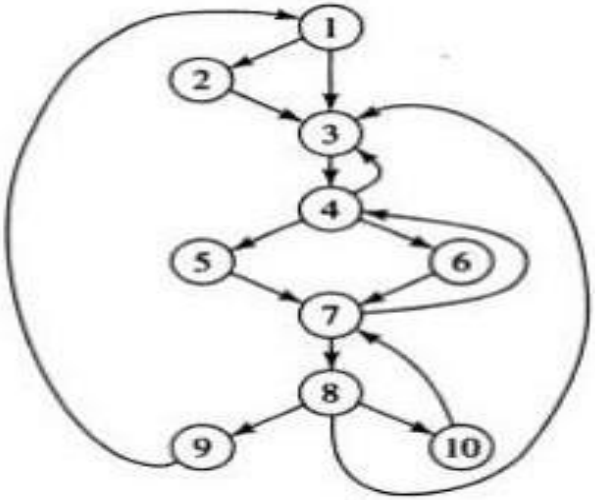


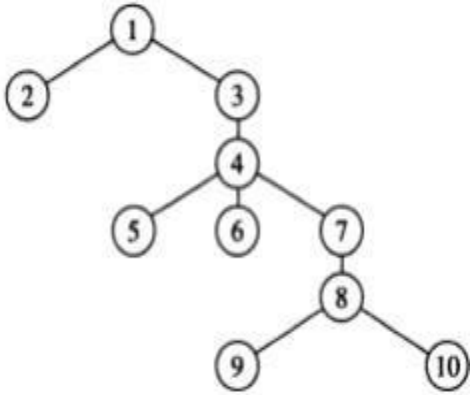
Dominators in flow graph:

- In a flow graph, a node d dominates node n, if every path from initial node of the flow graph to n goes through d. This will be denoted by d dom n.
- Every initial node dominates all the remaining nodes in the flow graph.
- Every node dominates itself.

Example:

- D(1)={1}
- D(2)={1,2}
- D(3)={1,3}
- D(4)={1,3,4}
- D(5)={1,3,4,5}
- D(6)={1,3,4,6}
- D(7)={1,3,4,7}
- D(8)={1,3,4,7,8}
- D(9)={1,3,4,7,8,9}
- D(10)={1,3,4,7,8,10}





Loops in flow graph:

- A loop must have a **single entry point, called the header**. This entry point-dominates all nodes in the loop.
- There must be **at least one way to iterate** the loop(i.e.)at least one path back to the header.
- One way to find all the loops in a flow graph is to search for edges in the flow graph **whose heads dominate their tails**. If a→b is an edge, b is the head and a is the tail. These types of edges are called as back edges.

Example:

Back edges:

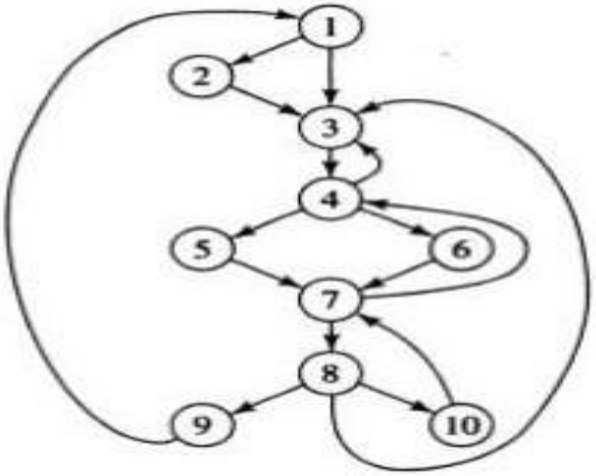
i) 7→4 4 DOM 7

ii) 10 →7 7 DOM 10

iii) 4→3 3 DOM 4

iv) 8→3 3 DOM 8

v) 9 →1 1 DOM 9



Natural loop:

For a back edge n → d, we define the natural loop of the edge to be d plus the set of nodes that can reach n without going through d. Node d is the header of the loop.

Example : if back edge is 7→4 ,then natural loop is{4,5,6,7}.

Constructing a Flow graph for given Three Address Code

Algorithm:

Step 1: Identifying leader in a Basic Block –

- First statement is always a leader
- Statement that is target of conditional or un-conditional statement is a leader
- Statement that follows immediately a conditional or un-conditional statement is a leader

Step 2: For each leader construct the basic block which consists of all the instructions up to but not including next leader or the end of intermediate code.

Step 3: Draw a flow graph

Example:

Three address code is:

1.i=0

2.if(i>10) goto 6

3.a[i]=0

4.i=i+1

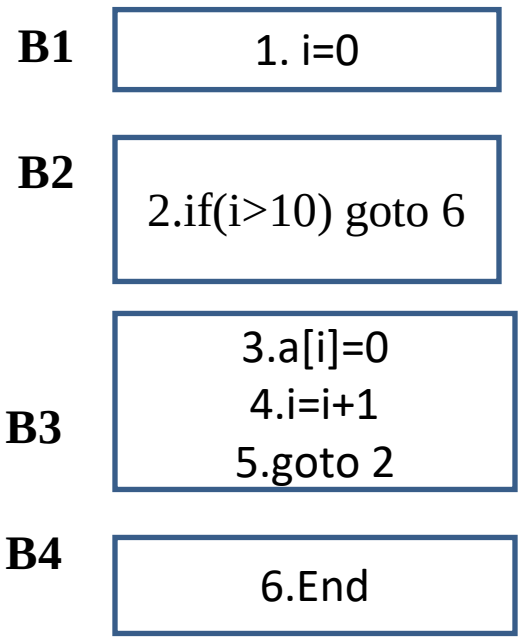
5 goto 2

6 End

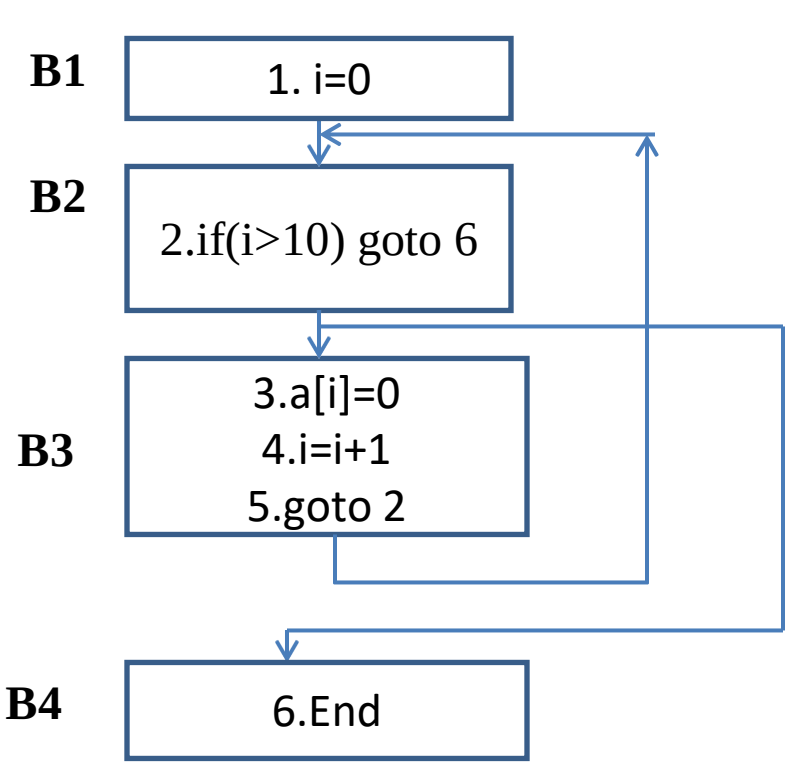
Step1: Identifying the Leader

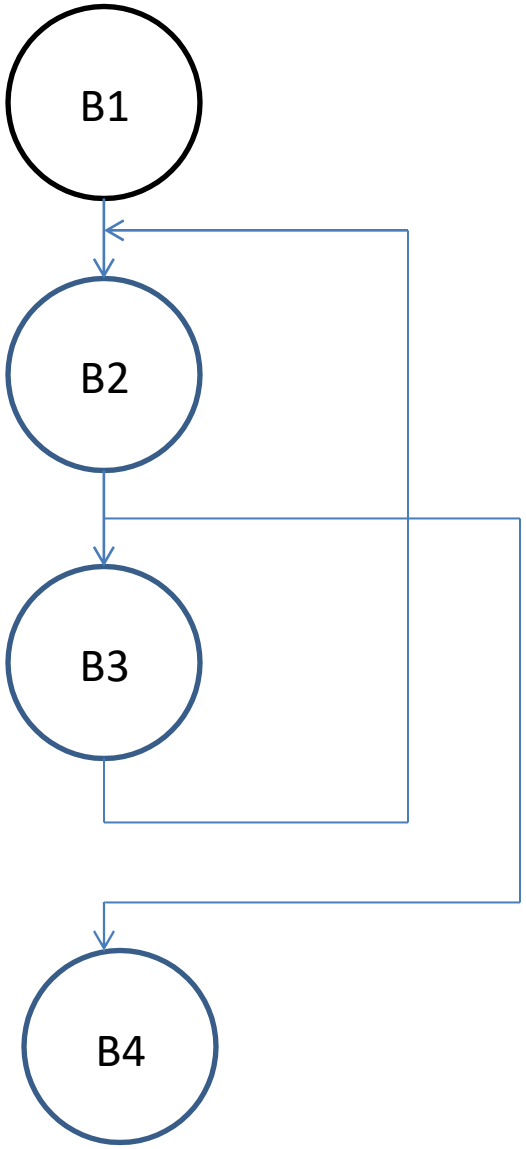
1.i=0 _____L
2.if(i>10) goto 6 ----------L
3.a[i]=0_____L
4.i=i+1
5 goto 2
6 End_____L

Step2: Constructing of basic blocks flow graph

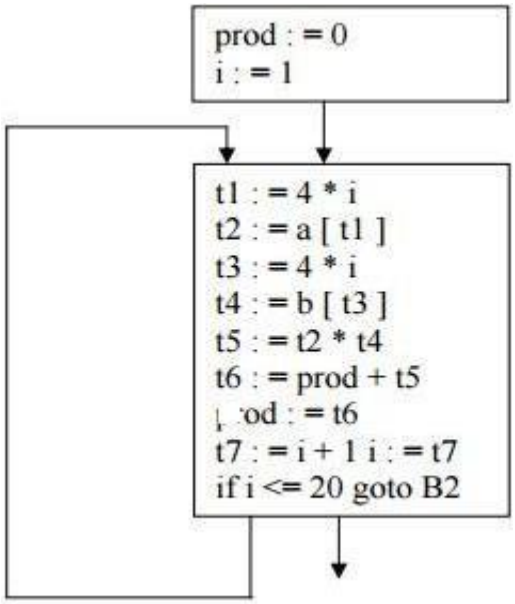


Step3: Constructing of flow graph





```
begin
        prod :=0;
        i:=1;
        do begin
        prod :=prod+ a[i] * b[i];
        i :=i+1;
        end
        while i <= 20
end
```



The three-address code for the above source program is given as :

(1) prod := 0 ----------L
(2) i := 1
(3) t1 := 4* i----------L
(4) t2 := a[t1] /*compute a[i] */
(5) t3 := 4* i
(6) t4 := b[t3] /*compute b[i] */
(7) t5 := t2*t4
(8) t6:= prod+t5
(9) prod=t6
(10) t7 **:= i+1**
(11) i=t7
(12) if i<=20 goto (3)

Basic block 1: Statement (1) to (2)
Basic block 2: Statement (3) to (12)

Live variable Analysis & Data Flow Equation

Live variable Analysis:

- It is data flow analysis performed by the compiler to find the variables that are live at the exit of each program point.
- A Variable is said to be live if it hold a value that may be needed in future.
- For a Basic block B:
  - **In[B]**= Live variable at the beginning of the Block B
  - **Out[B]**= Live variable at the End of the Block B
- Live variable analysis is done using **Data Flow Equation**

Where,

$$\text{Out}[B] = \text{Gen}[B] \cup (\text{In}[B] - \text{Kill}[B])$$

GEN[B] = set of all definitions inside B that are "visible" immediately after the block .

KILL[B] = union of the definitions in all the basic blocks of the flow graph, that are killed by individual statements in B.

Algorithm to find In and Out of each block in a flow graph

Input: $kill(B)$ and $gen(B)$ for every basic bloc B.

Output: $in(B)$ and $out(B)$ for every basic bloc B.

$in(B) = \phi$

for each B **repeat**

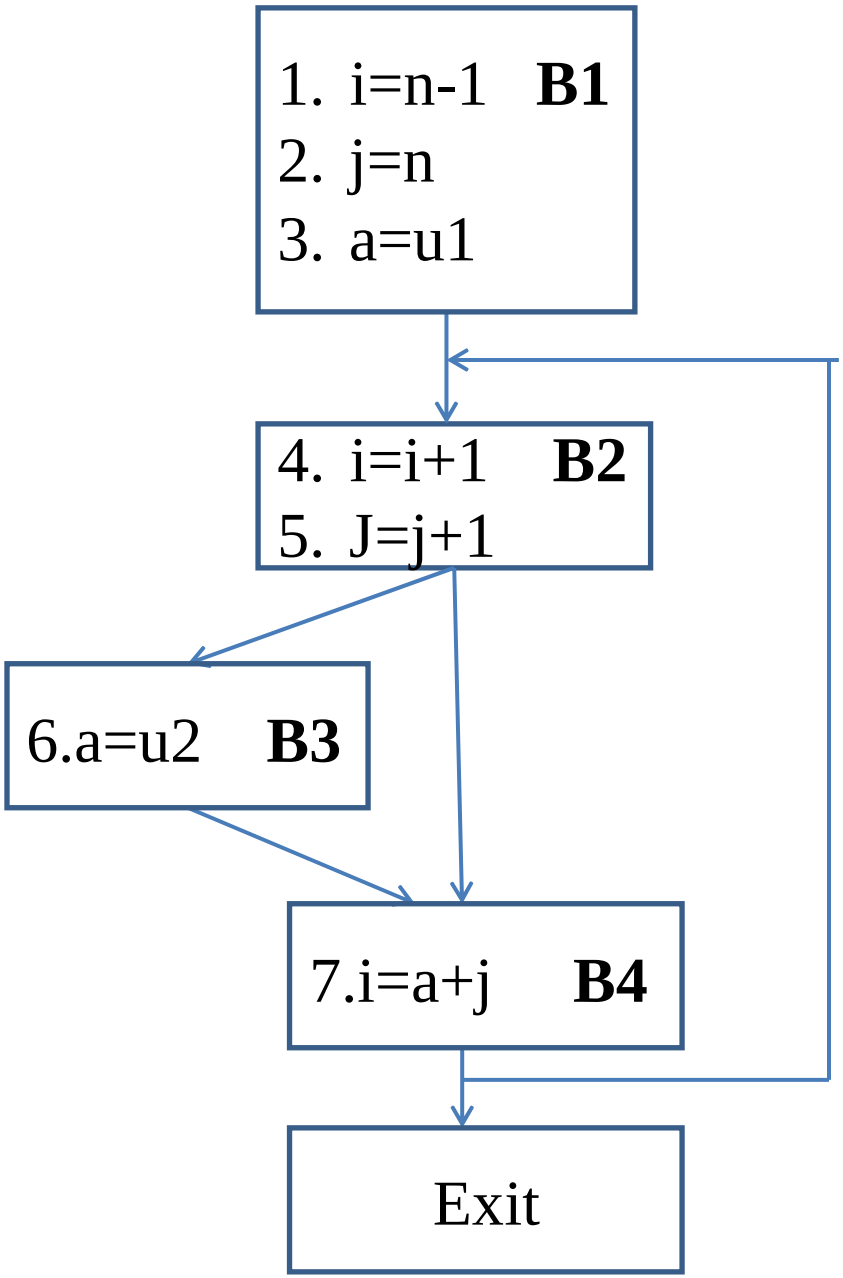
$out(B) := gen(B)$

while changes to any $out(B)$ occur **repeat**

$$in(B) := \bigcup_{B' \in \text{pred}(B)} out(B')$$

$$out(B) := gen(B) \bigcup (in(B) - kill(B))$$

Finding In and Out for the following flow graph:



Step1: Finding the Predecessors of all blocks

Blocks	Predecessor
B1	Φ
B2	B1,B4
B3	B2
B4	B2,B3

Step2: Finding the Gen and Kill of all blocks

Blocks	Gen	Kill
B1	{1,2,3}	{4,5,6,7}
B2	{4,5}	{1,2,7}
B3	{6)	{3}
B4	{7}	{1,4)

Step 3: Finding In and Out for all Blocks

Blocks	In	Out
B1	Φ	{1,2,3}
B2	Φ	{4,5}
B3	Φ	{6)
B4	Φ	{7}

Iteration-1:

In[B]= Φ and Out[B]=Gen[B]

Iteration-2:

In 2nd and subsequent Iterations In and Out values are calculated using previous iteration and following equations :

In[B]=In[B] U Out[P] where, P is Predecessor of B

Out[B]=Gen[B]U (In[B]-Kill[B])

Blocks	In	Out
B1	Φ	{1,2,3}
B2	{1,2,3,7}	{3,4,5}
B3	{4,5}	{4,5,6,7)
B4	{4,5,6}	{5,6,7}

Working:

In[B1]=In[B1] U Out[Predecessor(B1)]

= Φ U Out[Φ]

=Φ U Φ

= Φ

Out[B1]=Gen[B1]U(In[B1]-Kill[B1])

= {1,2,3}U(Φ- {4,5,6,7})

={1,2,3}U Φ

={1,2,3}

Similarly ,we have find In and Out for all blocks

Iteration-3

Working:

In[B2]=In[B2] U Out[Predecessor(B2)]

= {1,2,3,7} U Out[B1,B4]

={1,2,3,7} U Out[B1] U Out[B4]

= {1,2,3,7}U{1,2,3}U{5,6,7}

= {1,2,3,5,6,7}

Out[B2]=Gen[B2]U(In[B2]-Kill[B2])

= {4,5}U({1,2,3,5,6,7}-{1,2,7})

={4,5}U {3,5,6}

={3,4,5,6}

Blocks	In	Out
B1	Φ	{1,2,3}
B2	{1,2,3,5,6,7}	{3,4,5,6}
B3	{3,4,5}	{4,5,6)
B4	{3,4,5,6}	{3,5,6,7}

Iteration-4:

Blocks	In	Out
B1	Φ	{1,2,3}
B2	{1,2,3,5,6,7}	{3,4,5,6}
B3	{3,4,5,6}	{4,5,6)
B4	{3,4,5,6}	{3,5,6,7}

Iteration-5:

Blocks	In	Out
B1	Φ	{1,2,3}
B2	{1,2,3,5,6,7}	{3,4,5,6}
B3	{3,4,5,6}	{4,5,6)
B4	{3,4,5,6}	{3,5,6,7}

Since Iteration 4 and 5 are Identical we have stop the process. Finally we get In and Out of each block.

Peephole optimization:

- **Peephole optimization** is a type of Code Optimization performed on a **small part of the code** or **small set of instruction.**

- The small set of instructions or small part of code on which peephole optimization is performed is known as **peephole** or **window**.

- It basically works on the theory of replacement in which a part of code is **replaced by shorter and faster code without change in output.**

- Peephole is the machine dependent optimization.

- **Objectives of Peephole Optimization:**
  - To improve performance
  - To reduce memory footprint
  - To reduce code size

- **Peephole optimization techniques:**
  - Redundant instruction elimination
  - Unreachable code
  - Flow of control optimization
  - Algebraic expression simplification
  - Reduction in Strength

Redundant instruction elimination :

At compilation level, the compiler searches for instructions redundant in nature. Multiple loading and storing of instructions may carry the same meaning even if some of them are removed.

Example:

```
MOV x, R0
MOV R0, R1
```

We can delete the first instruction and re-write the sentence as:

```
MOV x, R1
```

Unreachable code:

Unreachable code is a part of the program code that is never accessed because of programming constructs. Programmers may have accidently written a piece of code that can never be reached.

Flow of control optimization:

If the program control jumps back and forth without performing any significant task. These jumps can be removed.

```
...
MOV R1, R2
GOTO L1
...
L1 : GOTO L2
L2 : INC R1
```

In this code , label L1 can be removed as it passes the control to L2. So instead of jumping to L1 and then to L2, the control can directly reach L2, as shown below:

```
...
MOV R1, R2
 GOTO L2
...
L2 : INC R1
```

Algebraic expression simplification:

Algebraic expressions can be made simple by applying simplification rules.

For example

The expression **a = a + 0** can be replaced by **a** itself and the expression a = a + 1 can simply be replaced by INC a.

Reduction in Strength:

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine.

For example,

- x^2 is invariably cheaper to implement as x*x .
- 2*x is invariably cheaper to implement as x+x

Code Generation:

Register Allocation and Assignment:

- The selection of set of variables that will reside in registers at a point in the program is called **register allocation.**
- The picking of specific register that a variable will reside in is called as **register assignment**.

Register and Address Descriptors:

- A **register descriptor** is used to keep track of what is currently in each registers. The register descriptors show that initially all the registers are empty.
- An **address descriptor** stores the location where the current value of the name can be found at run time.

Code-generation algorithm:

- **getReg** : Code generator uses getReg function to determine the status of available registers and the location of name values. It works as follows:
  - If variable Y is already in register R, it uses that register.
  - Else if some register R is available, it uses that register.
  - Else if both the above options are not possible, it chooses a register that requires minimal number of load(MM to Registers) and store(Registers to MM) instructions.

The algorithm takes as input a sequence of three-address statements constituting a basic block. For each three-address statement of the form x = y op z **,** perform the following actions:

1. Invokes a function *getreg* to determine the location L where the result of the computation y op z should be stored.
2. Consult the address descriptor for y to determine y', the current location of y. Prefer the register for y' if the value of y is currently both in memory and a register. If the value of y is not already in L, generate the instruction **MOV y' , L** to place a copy of y in L.
3. Generate the instruction **OP z' , L** where z' is a current location of z. Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors.
4. If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of x : = y op z , those registers will no longer contain y or z.

Example:

Generate Code for following three address code:

t : = a – b

u : = a – c

v : = t + u

d : = v + u

Statements	Code Generated	Register descriptor	Address descriptor
		Register empty	
t : = a - b	MOV a, R_0 SUB b, R0	R_0 contains t	t in R_0
u : = a - c	MOV a , R1 SUB c , R1	R_0 contains t R1 contains u	t in R_0 u in R1
v : = t + u	ADD R_1, R_0	R_0 contains v R_1 contains u	u in R_1 v in R_0
d : = v + u	ADD R_1, R_0 MOV R_0, d	R_0 contains d	d in R_0 d in R_0 and memory

DAG for Register Allocation

- Code generation from DAG is much simpler than the linear sequence of three address code.
- DAG can be used to rearrange sequence of instructions and generate and efficient code.
- The steps involved in the algorithm to generate code from DAG include :
  - **Rearranging the order** – To optimize the code generation, the instructions are rearranged and this is referred to as heuristic reordering .
  - **Labelling the tree for register information** – To know the number of registers required to generate code, the labels of the nodes are numbered which indicate the number of registers required to evaluate that node.
  - **Tree traversal to generate code** – This reordered labelled tree is traversed to generate code based on the target language's instruction.

Rearranging the order – Heuristic reordering :

- Rearranging the nodes involves changing the order of independent statements of the DAG which will help efficient utilization of the registers.
- This rearranging of nodes also helps in reducing the final cost of assembly level code.

DAG for Register Allocation

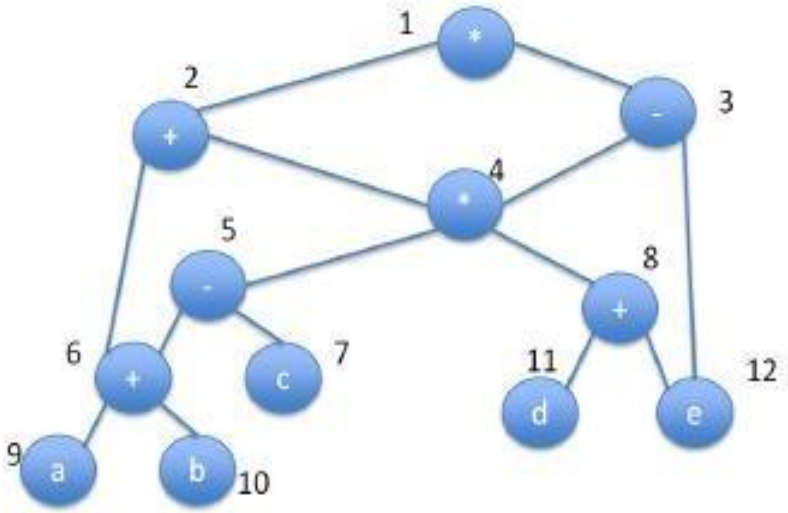
Algorithm:

```
Node_listing ( )
{
  while unlisted interior nodes remain do
  begin
    select an unlisted node  n, all of whose parents have been listed ;
    list n;
    while the leftmost child m of n has no unlisted parents and is not a leaf do
    /* since n was just listed, m is not yet listed*/
      begin
          list m;
          n = m
      end
  end
}
```

- Final order = reverse of the order of listing of nodes .

DAG for Register Allocation

Example:



- The listed nodes are “1234568”. This string is reversed to yield, “8654321”.
- This indicates we need to evaluate node 8 followed by 6, 5, 4, 3, 2 and finally 1.
- The following is the sequence of instruction after rearranging.

1. t8 := d +e
2. t6 := a +b
3. t5 := t6 - c
4. t4 := t5 * t8
5. t3 := t4 – e
6. t2 := t6 + t4
7. t1:= t2 + t3

DAG for Register Allocation

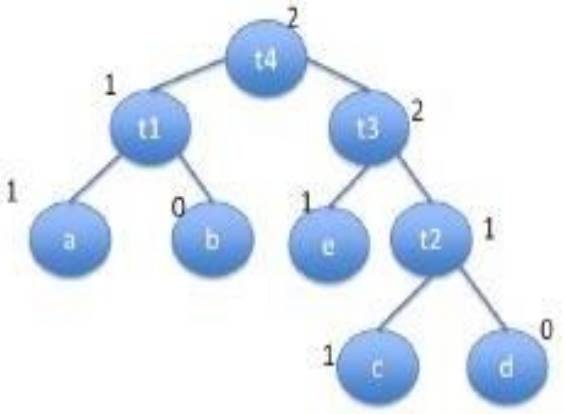
Labelling the tree for register information :

- A node 'n' is labelled using the following equation:

$$label(n) = \begin{cases} \max(l1, l2) & if\ l1 \neq l2 \\ l1 + 1 & if\ l1 = l2 \end{cases}$$

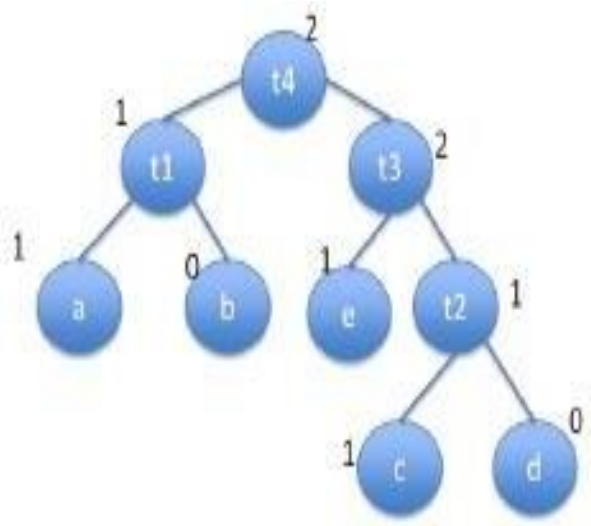
- Where l1 is the left child label and l2 is right child label.

```
Node_labelling( )
{
        if n is a leaf then
          if n is leftmost child of its parents then
            label (n) = 1
          else
            label (n) = 0
       else
        begin /* n is an interior node */
              let n1, n2 , …. , nk be the children of n ordered by label ,
              so label (n1) >= label (n2) >= ….>= label (nk) ;
              label (n) = max (label(ni) + i - 1)
         end
}
```



DAG for Register Allocation

- We use post order traversal for label computation.
- Node ‘a’ is labelled 1 since it is the left most leaf node. ‘b’ is labelled 0 as it is the right leaf node. Parent of a, b is assigned max (1,0) which is 1.
- We then assign ‘e’ with a value of ‘1’ as it is a left leaf node, ‘c’ and ‘d’ with the values of ‘1’ and ‘0’ as they are the left and right leaf nodes.
- Node t2 is labelled ‘1’ which is the maximum of nodes ‘c’ and ‘d’ label.
- Node t3 is assigned ‘2’ as its children have a label ‘1’ and this node’s label is computed as ‘1’ + label (e).
- Root’s label is given as ‘2’ as its right child has a maximum value of ‘2’.



******THANK YOU******