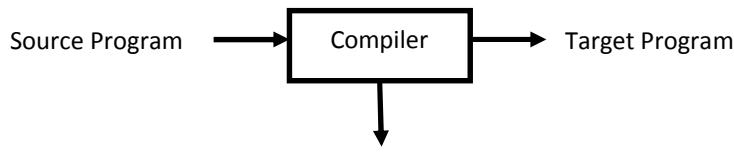


### 1. Explain overview of translation process.

- A translator is a kind of program that takes one form of program as input and converts it into another form.
- The input is called *source program* and output is called *target program*.
- The source language can be assembly language or higher level language like C, C++, FORTRAN, etc...
- There are three types of translators,
  1. Compiler
  2. Interpreter
  3. Assembler

### 2. What is compiler? List major functions done by compiler.

- A compiler is a program that reads a program written in one language and translates it into an equivalent program in another language.



**Fig.1.1. A Compiler**

Major functions done by compiler:

- Compiler is used to convert one form of program to another.
- A compiler should convert the source program to a target machine code in such a way that the generated target code should be easy to understand.
- Compiler should preserve the meaning of source code.
- Compiler should report errors that occur during compilation process.
- The compilation must be done efficiently.

### 3. Write the difference between compiler, interpreter and assembler.

#### 1. *Compiler v/s Interpreter*

No.	Compiler	Interpreter
1	Compiler takes entire program as an input.	Interpreter takes single instruction as an input.
2	Intermediate code is generated.	No Intermediate code is generated.
3	Memory requirement is more.	Memory requirement is less.
4	Error is displayed after entire program is checked.	Error is displayed for every instruction interpreted.
5	Example: C compiler	Example: BASIC

**Table 1.1 Difference between Compiler & Interpreter**

## 2. Compiler v/s Assembler

No.	Compiler	Assembler
1	It translates higher level language to machine code.	It translates mnemonic operation code to machine code.
2	Types of compiler, <ul style="list-style-type: none"> <li>• Single pass compiler</li> <li>• Multi pass compiler</li> </ul>	Types of assembler, <ul style="list-style-type: none"> <li>• Single pass assembler</li> <li>• Two pass assembler</li> </ul>
3	Example: C compiler	Example: 8085, 8086 instruction set

Table 1.2 Difference between Compiler & Assembler

**4. Analysis synthesis model of compilation. OR**

**Explain structure of compiler. OR**

**Explain phases of compiler. OR**

**Write output of phases of a complier. for  $a = a + b * c * 2$ ; type of a, b, c are float**

There are mainly two parts of compilation process.

1. **Analysis phase:** The main objective of the analysis phase is to break the source code into parts and then arranges these pieces into a meaningful structure.
2. **Synthesis phase:** Synthesis phase is concerned with generation of target language statement which has the same meaning as the source statement.

**Analysis Phase:** Analysis part is divided into three sub parts,

- I. Lexical analysis
- II. Syntax analysis
- III. Semantic analysis

**Lexical analysis:**

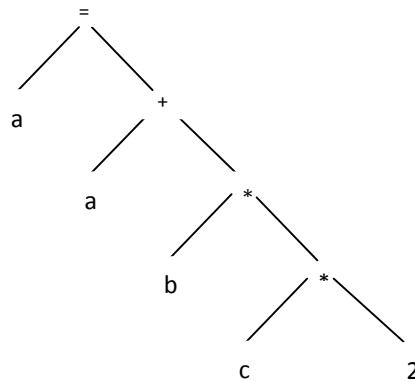
- Lexical analysis is also called linear analysis or scanning.
- Lexical analyzer reads the source program and then it is broken into stream of units. Such units are called token.
- Then it classifies the units into different lexical classes. E.g. id's, constants, keyword etc...and enters them into different tables.
- For example, in lexical analysis the assignment statement  $a := a + b * c * 2$  would be grouped into the following tokens:

a	Identifier 1
=	Assignment sign
a	Identifier 1
+	The plus sign
b	Identifier 2
*	Multiplication sign
c	Identifier 3
*	Multiplication

	sign
2	Number 2

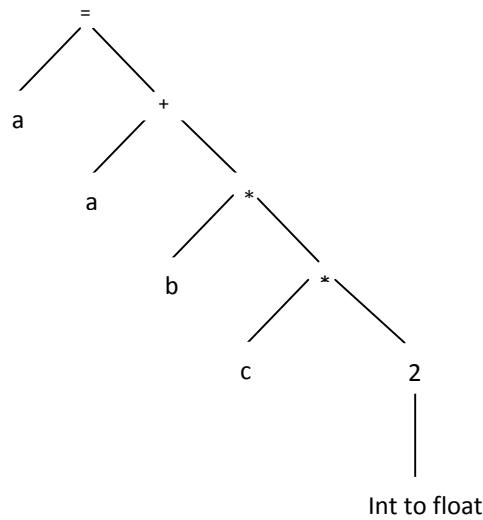
### Syntax Analysis:

- Syntax analysis is also called hierarchical analysis or parsing.
- The syntax analyzer checks each line of the code and spots every tiny mistake that the programmer has committed while typing the code.
- If code is error free then syntax analyzer generates the tree.



### Semantic analysis:

- Semantic analyzer determines the meaning of a source string.
- For example matching of parenthesis in the expression, or matching of if..else statement or performing arithmetic operation that are type compatible, or checking the scope of operation.



**Synthesis phase:** synthesis part is divided into three sub parts,

- I. Intermediate code generation
- II. Code optimization
- III. Code generation

### Intermediate code generation:

- The intermediate representation should have two important properties, it should be

- easy to produce and easy to translate into target program.
- We consider intermediate form called “three address code”.
- Three address code consist of a sequence of instruction, each of which has at most three operands.
- The source program might appear in three address code as,

```
t1= int to real(2)
t2= id3 * t1
t3= t2 * id2
t4= t3 + id1
id1= t4
```

### Code optimization:

- The code optimization phase attempt to improve the intermediate code.
- This is necessary to have a faster executing code or less consumption of memory.
- Thus by optimizing the code the overall running time of a target program can be improved.

```
t1= id3 * 2.0
t2= id2 * t1
id1 = id1 + t2
```

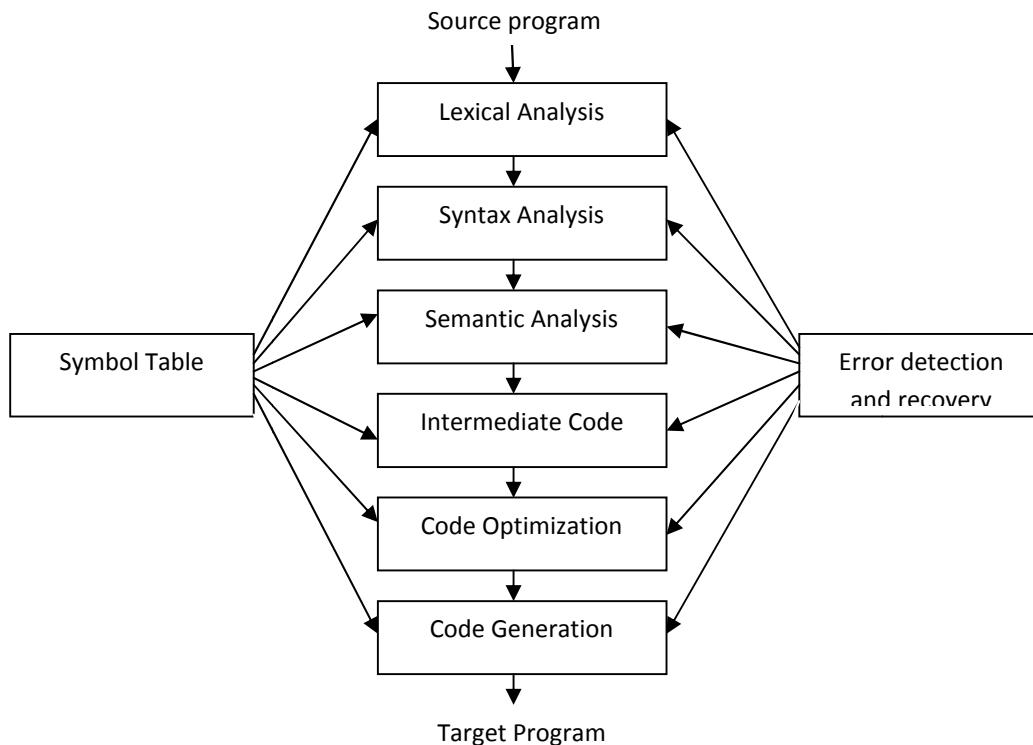
### Code generation:

- In code generation phase the target code gets generated. The intermediate code instructions are translated into sequence of machine instruction.

```
MOV id3, R1
MUL #2.0, R1
MOV id2, R2
MUL R2, R1
MOV id1, R2
ADD R2, R1
MOV R1, id1
```

### Symbol Table

- A **symbol table** is a data structure used by a language translator such as a compiler or interpreter.
- It is used to store names encountered in the source program, along with the relevant attributes for those names.
- Information about following entities is stored in the symbol table.
  - ✓ Variable/Identifier
  - ✓ Procedure/function
  - ✓ Keyword
  - ✓ Constant
  - ✓ Class name
  - ✓ Label name



**Fig.1.2. Phases of Compiler**

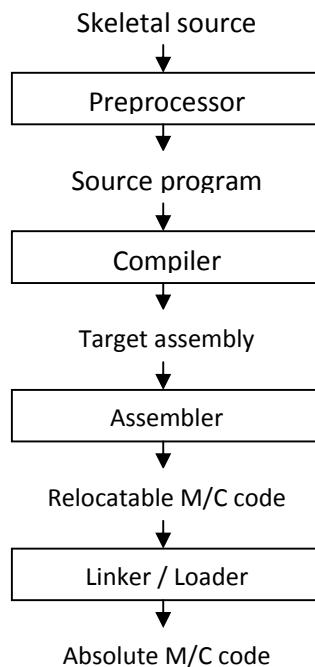
**5. The context of a compiler. OR  
Cousins of compiler. OR  
What does the linker do? What does the loader do? What does the Preprocessor do? Explain their role(s) in compilation process.**

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

#### Preprocessor

Preprocessor produces input to compiler. They may perform the following functions,

1. Macro processing: A preprocessor may allow user to define macros that are shorthand for longer constructs.
2. File inclusion: A preprocessor may include the header file into the program text.
3. Rational preprocessor: Such a preprocessor provides the user with built in macro for construct like while statement or if statement.
4. Language extensions: this processors attempt to add capabilities to the language by what amount to built-in macros. Ex: the language equal is a database query language embedded in C. statement beginning with ## are taken by preprocessor to be database access statement unrelated to C and translated into procedure call on routines that perform the database access.



**Fig.1.3. Context of Compiler**

### Assembler

Assembler is a translator which takes the assembly program as an input and generates the machine code as a output. An assembly is a mnemonic version of machine code, in which names are used instead of binary codes for operations.

### Linker

Linker allows us to make a single program from a several files of relocatable machine code. These file may have been the result of several different compilation, and one or more may be library files of routine provided by a system.

### Loader

The process of loading consists of taking relocatable machine code, altering the relocatable address and placing the altered instructions and data in memory at the proper location.

## 6. Explain front end and back end in brief. (Grouping of phases)

- The phases are collected into a front end and back end.

### Front end

- The front end consist of those phases, that depends primarily on source language and largely independent of the target machine.
- Front end includes lexical analysis, syntax analysis, semantic analysis, intermediate code generation and creation of symbol table.
- Certain amount of code optimization can be done by front end.

### Back end

- The back end consists of those phases, that depends on target machine and do not depend on source program.

- Back end includes code optimization and code generation phase with necessary error handling and symbol table operation.

### 7. What is the pass of compiler? Explain how the single and multi-pass compilers work? What is the effect of reducing the number of passes?

- One complete scan of a source program is called pass.
- Pass include reading an input file and writing to the output file.
- In a single pass compiler analysis of source statement is immediately followed by synthesis of equivalent target statement.
- It is difficult to compile the source program into single pass due to:
- **Forward reference:** a forward reference of a program entity is a reference to the entity which precedes its definition in the program.
- This problem can be solved by postponing the generation of target code until more information concerning the entity becomes available.
- It leads to multi pass model of compilation.
- In Pass I: Perform analysis of the source program and note relevant information.
- In Pass II: Generate target code using information noted in pass I.

#### Effect of reducing the number of passes

- It is desirable to have a few passes, because it takes time to read and write intermediate file.
- On the other hand if we group several phases into one pass we may be forced to keep the entire program in the memory. Therefore memory requirement may be large.

### 8. Explain types of compiler. OR

#### Write difference between single pass and multi pass compiler.

#### *Single pass compiler v/s Multi pass Compiler*

No.	Single pass compiler	Multi pass compiler
1	A one-pass compiler is a compiler that passes through the source code of each compilation unit only once.	A multi-pass compiler is a type of compiler that processes the source code or abstract syntax tree of a program several times.
2	A one-pass compiler is faster than multi-pass compiler.	A multi-pass compiler is slower than single-pass compiler.
3	One-pass compiler are sometimes called narrow compiler.	Multi-pass compilers are sometimes called wide compiler.
4	Language like Pascal can be implemented with a single pass compiler.	Languages like Java require a multi-pass compiler.

Table 1.3 Difference between Single Pass Compiler & Multi Pass Compiler

### 9. Write the difference between phase and pass.

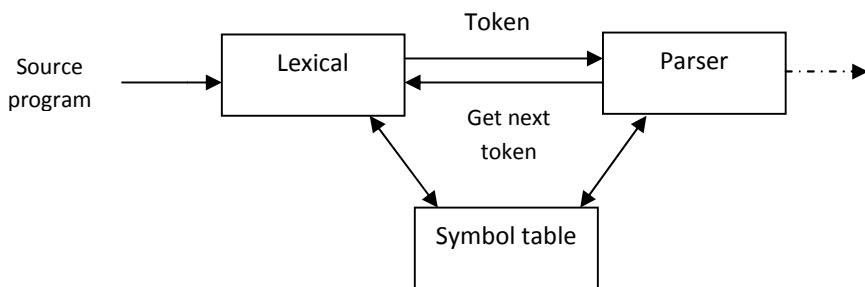
#### *Phase v/s Pass*

No.	Phase	Pass
1	The process of compilation is carried out in various step is called phase.	Various phases are logically grouped together to form a pass.
2	The phases of compilation are lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization and code generation.	The process of compilation can be carried out in a single pass or in multiple passes.

**Table 1.3 Difference between Phase & Pass**

### 1. Role of lexical analysis and its issues. OR How do the parser and scanner communicate? Explain with the block diagram communication between them.

- The lexical analyzer is the first phase of compiler. Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.
- This interaction is given in figure 2.1,



**Fig. 2.1 Communication between Scanner & Parser**

- It is implemented by making lexical analyzer be a subroutine.
- Upon receiving a “get next token” command from parser, the lexical analyzer reads the input character until it can identify the next token.
- It may also perform secondary task at user interface.
- One such task is stripping out from the source program comments and white space in the form of blanks, tabs, and newline characters.
- Some lexical analyzer are divided into cascade of two phases, the first called scanning and second is “lexical analysis”.
- The scanner is responsible for doing simple task while lexical analysis does the more complex task.

#### Issues in Lexical Analysis:

There are several reasons for separating the analysis phase of compiling into lexical analysis and parsing:

- Simpler design is perhaps the most important consideration. The separation of lexical analysis often allows us to simplify one or other of these phases.
- Compiler efficiency is improved.
- Compiler portability is enhanced.

### 2. Explain token, pattern and lexemes.

**Token:** Sequence of character having a collective meaning is known as *token*.

Typical tokens are,

1) Identifiers 2) keywords 3) operators 4) special symbols 5) constants

**Pattern:** The set of rules called *pattern* associated with a token.

**Lexeme:** The sequence of character in a source program matched with a pattern for a token is

called lexeme.

Token	Lexeme	Pattern
Const	Const	Const
If	If	If
Relation	<,<=,= ,>,>=,>	< or <= or = or > or >= or >
Id	Pi, count, n, I	letter followed by letters and digits.
Number	3.14159, 0, 6.02e23	Any numeric constant
Literal	"Darshan Institute"	Any character between " and " except "

Table 2.1. Examples of Tokens

**Example:**

**total = sum + 12.5**

Tokens are: total (id),

= (relation)

Sum (id)

+ (operator)

12.5 (num)

Lexemes are: total, =, sum, +, 12.5

### 3. What is input buffering? Explain technique of buffer pair. OR Which technique is used for speeding up the lexical analyzer?

There are mainly two techniques for input buffering,

- ✓ Buffer pair
- ✓ Sentinels

#### 1. Buffer pair:

- The lexical analysis scans the input string from left to right one character at a time.
- So, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character.
- We use a buffer divided into two N-character halves, as shown in figure 2.2. N is the number of character on one disk block.

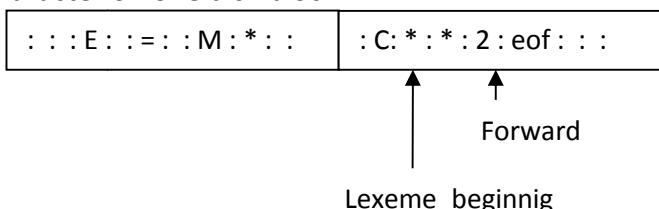


Fig. 2.2 An input buffer in two halves

- We read N input character into each half of the buffer.
- Two pointers to the input are maintained and string between two pointers is the current lexemes.

- Pointer *Lexeme Begin*, marks the beginning of the current lexeme.
- Pointer *Forward*, scans ahead until a pattern match is found.
- If forward pointer is at the end of first buffer half then second is filled with N input character.
- If forward pointer is at the end of second buffer half then first is filled with N input character.

code to advance forward pointer is given below,

```

if forward at end of first half then begin
    reload second half;
    forward := forward + 1;
end
else if forward at end of second half then begin
    reload first half;
    move forward to beginning of first half;
end
else forward := forward + 1;

```

Once the next lexeme is determined, *forward* is set to character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, *Lexeme Begin* is set to the character immediately after the lexeme just found.

### 2. Sentinels:

- If we use the scheme of Buffer pairs we must check, each time we move the forward pointer that we have not moved off one of the buffers; if we do, then we must reload the other buffer. Thus, for each character read, we make two tests.
- We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **EOF**.

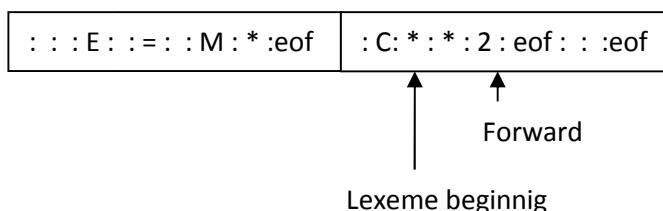


Fig.2.3. Sentinels at end of each buffer half

- Look ahead code with sentinels is given below:

```

forward := forward + 1;
if forward = eof then begin
    if forward at end of first half then begin
        reload second half;
        forward := forward + 1;
end
else if forward at the second half then begin
    reload first half;

```

```

move forward to beginning of first half;
end
else terminate lexical analysis;
end;

```

### 4. Specification of token.

#### Strings and languages

Terms for a part of string

Term	Definition
Prefix of S	A string obtained by removing zero or more trailing symbol of string S. e.g., ban is prefix of banana.
Suffix of S	A string obtained by removing zero or more leading symbol of string S. e.g., nana is suffix of banana.
Sub string of S	A string obtained by removing prefix and suffix from S. e.g., nan is substring of banana.
Proper prefix, suffix and substring of S	Any nonempty string x that is respectively prefix, suffix or substring of S, such that $s \neq x$
Subsequence of S	A string obtained by removing zero or more not necessarily contiguous symbol from S. e.g., baaa is subsequence of banana.

Table 2.2. Terms for a part of a string

#### Operation on languages

Definition of operation on language

Operation	Definition
Union of L and M Written $L \cup M$	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
concatenation of L and M Written $LM$	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L written $L^*$	$L^*$ denotes “zero or more concatenation of” L.
Positive closure of L written $L^+$	$L^+$ denotes “one or more concatenation of” L.

Table 2.3. Definitions of operations on languages

### 5. Regular Expression & Regular Definition.

#### Regular Expression

1. 0 or 1  
 $0+1$
2. 0 or 11 or 111  
 $0+11+111$

3. Regular expression over  $\Sigma = \{a, b, c\}$  that represent all string of length 3.  
 $(a+b+c)(a+b+c)(a+b+c)$
4. String having zero or more a.  
 $a^*$
5. String having one or more a.  
 $a^+$
6. All binary string.  
 $(0+1)^*$
7. 0 or more occurrence of either a or b or both  
 $(a+b)^*$
8. 1 or more occurrence of either a or b or both  
 $(a+b)^+$
9. Binary no. end with 0  
 $(0+1)^*0$
10. Binary no. end with 1  
 $(0+1)^*1$
11. Binary no. starts and end with 1.  
 $1(0+1)^*1$
12. String starts and ends with same character.  
 $0(0+1)^*0$       or       $a(a+b)^*a$   
 $1(0+1)^*1$       or       $b(a+b)^*b$
13. All string of a and b starting with a  
 $a(a/b)^*$
14. String of 0 and 1 end with 00.  
 $(0+1)^*00$
15. String end with abb.  
 $(a+b)^*abb$
16. String start with 1 and end with 0.  
 $1(0+1)^*0$
17. All binary string with at least 3 characters and 3rd character should be zero.  
 $(0+1)(0+1)0(0+1)^*$
18. Language which consist of exactly two b's over the set  $\Sigma = \{a, b\}$   
 $a^*ba^*ba^*$
19.  $\Sigma = \{a, b\}$  such that 3rd character from right end of the string is always a.  
 $(a+b)^*a(a+b)(a+b)$
20. Any no. of a followed by any no. of b followed by any no. of c.  
 $a^*b^*c^*$
21. It should contain at least 3 one.  
 $(0+1)^*1(0+1)^*1(0+1)^*1(0+1)^*$
22. String should contain exactly Two 1's  
 $0^*10^*10^*$
23. Length should be at least be 1 and at most 3.  
 $(0+1) + (0+1) (0+1) + (0+1) (0+1) (0+1)$

24. No.of zero should be multiple of 3  
 $(1^*01^*01^*01^*)^*+1^*$
25.  $\Sigma=\{a,b,c\}$  where a are multiple of 3.  
 $((b+c)^*a(b+c)^*a(b+c)^*a(b+c)^*)^*$
26. Even no. of 0.  
 $(1^*01^*01^*)^*$
27. Odd no. of 1.  
 $0^*(10^*10^*)^*10^*$
28. String should have odd length.  
 $(0+1)((0+1)(0+1))^*$
29. String should have even length.  
 $((0+1)(0+1))^*$
30. String start with 0 and has odd length.  
 $0((0+1)(0+1))^*$
31. String start with 1 and has even length.  
 $1(0+1)((0+1)(0+1))^*$
32. Even no of 1  
 $(0^*10^*10^*)^*$
33. String of length 6 or less  
 $(0+1+\wedge)^6$
34. String ending with 1 and not contain 00.  
 $(1+01)^+$
35. All string begins or ends with 00 or 11.  
 $(00+11)(0+1)^*+(0+1)^*(00+11)$
36. All string not contains the substring 00.  
 $(1+01)^* (\wedge+0)$
37. Language of all string containing both 11 and 00 as substring.  
 $((0+1)^*00(0+1)^*11(0+1)^*)+ ((0+1)^*11(0+1)^*00(0+1)^*)$
38. Language of C identifier.  
 $(\_+L)(\_+L+D)^*$

### Regular Definition

- A regular definition gives names to certain regular expressions and uses those names in other regular expressions.
- Here is a regular definition for the set of Pascal identifiers that is define as the set of strings of letters and digits beginning with a letters.

$$\begin{aligned} \text{letter} &\rightarrow A \mid B \mid \dots \mid Z \mid a \mid b \mid \dots \mid z \\ \text{digit} &\rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9 \\ \text{id} &\rightarrow \text{letter} (\text{letter} \mid \text{digit})^* \end{aligned}$$

- The regular expression id is the pattern for the identifier token and defines letter and digit. Where letter is a regular expression for the set of all upper-case and lower case letters in the alphabet and digit is the regular expression for the set of all decimal digits.

### 6. Reorganization of Token.

- Here we address how to recognize token.
- We use the language generated by following grammar,

$$\begin{aligned} \text{stmt} \rightarrow & \text{if expr then stmt} \\ & | \text{if expr then stmt else stmt} \\ & | \epsilon \end{aligned}$$

$$\text{expr} \rightarrow \text{term relop term}$$

$$| \text{term}$$

$$\text{term} \rightarrow \text{id} | \text{num}$$

- Where the terminals if, then, else, relop, id and num generates the set of strings given by the following regular definitions,

$$\text{if} \rightarrow \text{if}$$

$$\text{then} \rightarrow \text{then}$$

$$\text{relop} \rightarrow < | \leq | = | <> | > | \geq$$

$$\text{letter} \rightarrow \text{A} | \text{B} | \dots | \text{Z} | \text{a} | \text{b} | \dots | \text{z}$$

$$\text{digit} \rightarrow 0 | 1 | 2 | \dots | 9$$

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

$$\text{num} \rightarrow \text{digit}^+ ()?(\text{E}(+/-)?\text{digit}^+ )?$$

- For this language the lexical analyzer will recognize the keyword if, then, else, as well as the lexeme denoted by **relop**, **id** and **num**.
- num represents the unsigned integer and real numbers of pascal.
- Lexical analyzer will isolate the next token in the input buffer and produces token and attribute value as an output.

### 7. Transition Diagram.

- A stylized flowchart is called transition diagram.
- Positions in a transition diagram are drawn as a circle and are called states.
- States are connected by arrows called edges.
- Edge leaving state have label indicating the input character.
- The transition diagram for unsigned number is given in Fig.2.4.

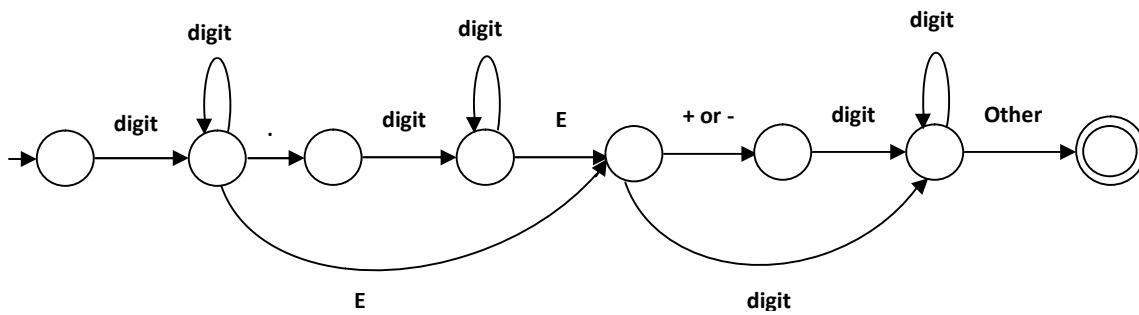


Fig. 2.4. Transition diagram for unsigned number

- Transition diagram for the token **relop** is shown in figure 2.5.

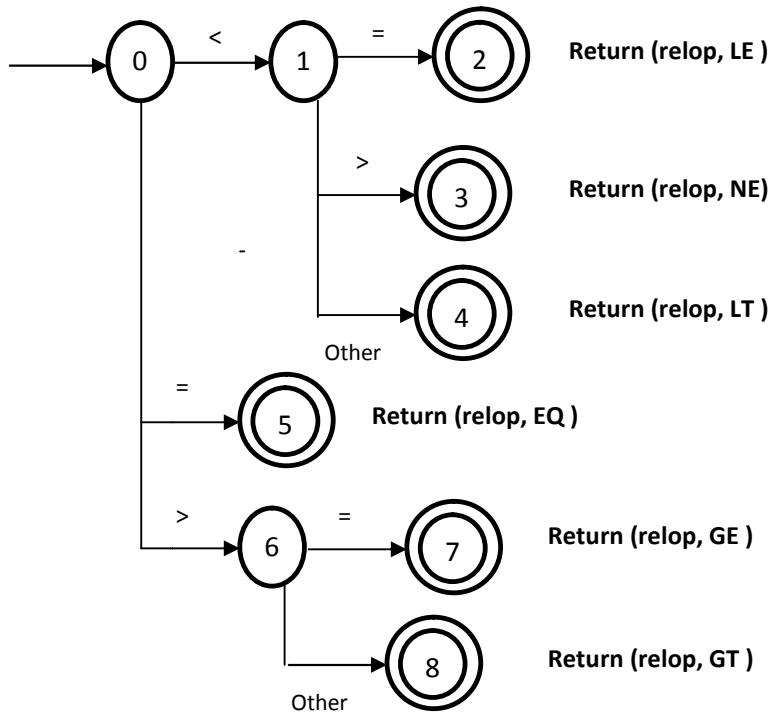


Fig. 2.5. Transition diagram for relational operator

### 8. Explain Finite automata. (NFA & DFA)

- We compile a regular expression into a recognizer by constructing a generalized transition diagram called a finite automaton.
- A finite Automata or finite state machine is a 5-tuple  $(S, \Sigma, S_0, F, \delta)$  where
  - $S$  is finite set of states
  - $\Sigma$  is finite alphabet of input symbol
  - $S_0 \in S$  (Initial state)
  - $F$  (set of accepting states)
  - $\delta$  is a transition function
- There are two types of finite automata,
  - Deterministic finite automata (DFA) have for each state (circle in the diagram) exactly one edge leaving out for each symbol.
  - Nondeterministic finite automata (NFA) are the other kind. There are no restrictions on the edges leaving a state. There can be several with the same symbol as label and some edges can be labeled with  $\epsilon$ .

### 9. Conversion from NFA to DFA using Thompson's rule.

Ex:1  $(a+b)^*abb$

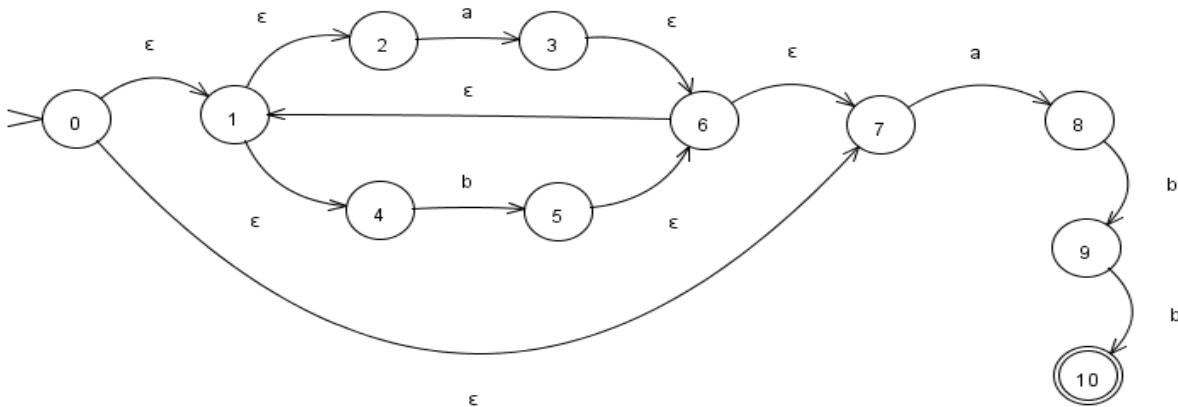


Fig. 2.6. NFA for  $(a+b)^*abb$

- $\epsilon$  – closure (0) = {0,1,2,4,7} ---- Let A
- Move(A,a) = {3,8}
  - $\epsilon$  – closure (Move(A,a)) = {1,2,3,4,6,7,8}---- Let B
  - Move(A,b) = {5}
  - $\epsilon$  – closure (Move(A,b)) = {1,2,4,5,6,7}---- Let C
- Move(B,a) = {3,8}
  - $\epsilon$  – closure (Move(B,a)) = {1,2,3,4,6,7,8}---- B
  - Move(B,b) = {5,9}
  - $\epsilon$  – closure (Move(B,b)) = {1,2,4,5,6,7,9}---- Let D
- Move(C,a) = {3,8}
  - $\epsilon$  – closure (Move(C,a)) = {1,2,3,4,6,7,8}---- B
  - Move(C,b) = {5}
  - $\epsilon$  – closure (Move(C,b)) = {1,2,4,5,6,7}---- C
- Move(D,a) = {3,8}
  - $\epsilon$  – closure (Move(D,a)) = {1,2,3,4,6,7,8}---- B
  - Move(D,b) = {5,10}
  - $\epsilon$  – closure (Move(D,b)) = {1,2,4,5,6,7,10}---- Let E
- Move(E,a) = {3,8}
  - $\epsilon$  – closure (Move(E,a)) = {1,2,3,4,6,7,8}---- B
  - Move(E,b) = {5}
  - $\epsilon$  – closure (Move(E,b)) = {1,2,4,5,6,7}---- C

States	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

Table 2.4. Transition table for  $(a+b)^*abb$

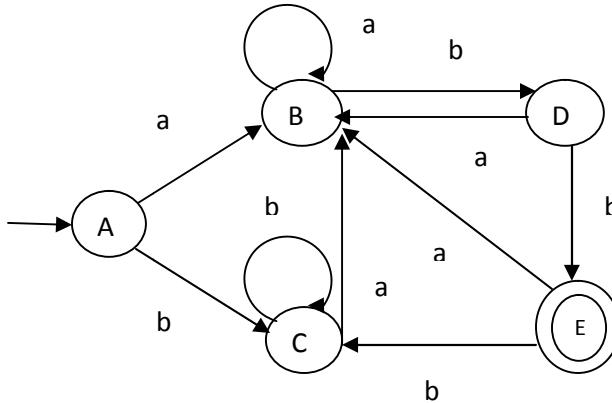


Fig.2.7. DFA for  $(a+b)^*abb$

### DFA Optimization

- Algorithm to minimize the number of states of a DFA
- 1. Construct an initial partition  $\Pi$  of the set of states with two groups: the accepting states  $F$  and the non-accepting states  $S - F$ .
- 2. Apply the repartition procedure to  $\Pi$  to construct a new partition  $\Pi_{\text{new}}$ .
- 3. If  $\Pi_{\text{new}} = \Pi$ , let  $\Pi_{\text{final}} = \Pi$  and continue with step (4). Otherwise, repeat step (2) with  $\Pi = \Pi_{\text{new}}$ .

for each group  $G$  of  $\Pi$  do begin

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

replace  $G$  in  $\Pi_{\text{new}}$  by the set of all subgroups formed

- 4. Choose one state in each group of the partition  $\Pi_{\text{final}}$  as the representative for that group. The representatives will be the states of  $M'$ . Let  $s$  be a representative state, and suppose on input  $a$  there is a transition of  $M$  from  $s$  to  $t$ . Let  $r$  be the representative of  $t$ 's group. Then  $M'$  has a transition from  $s$  to  $r$  on  $a$ . Let the start state of  $M'$  be the representative of the group containing start state  $s_0$  of  $M$ , and let the accepting states of  $M'$  be the representatives that are in  $F$ .
- 5. If  $M'$  has a dead state  $d$  (non-accepting, all transitions to self), then remove  $d$  from  $M'$ . Also remove any state not reachable from the start state.

Example: Consider transition table of above example and apply algorithm.

- Initial partition consists of two groups ( $E$ ) accepting state and non accepting states (ABCD).

- E is single state so, cannot be split further.
- For (ABCD), on input a each of these state has transition to B. but on input b, however A, B and C go to member of the group (ABCD), while D goes to E, a member of other group.
- Thus, (ABCD) split into two groups, (ABC) and (D). so, new groups are (ABC)(D) and (E).
- Apply same procedure again no splitting on input a, but (ABC) must be splitting into two group (AC) and (B), since on input b, A and C each have a transition to C, while B has transition to D. so, new groups (AC)(B)(D)(E).
- Now, no more splitting is possible.
- If we chose A as the representative for group (AC), then we obtain reduced transition table shown in table 2.5,

States	a	b
A	B	C
B	B	D
D	B	E
E	B	C

Table 2.5. Optimized Transition table for  $(a+b)^*abb$

### 10. Conversion from Regular Expression to DFA without constructing NFA.

Ex:1  $(a+b)^*abb\#$

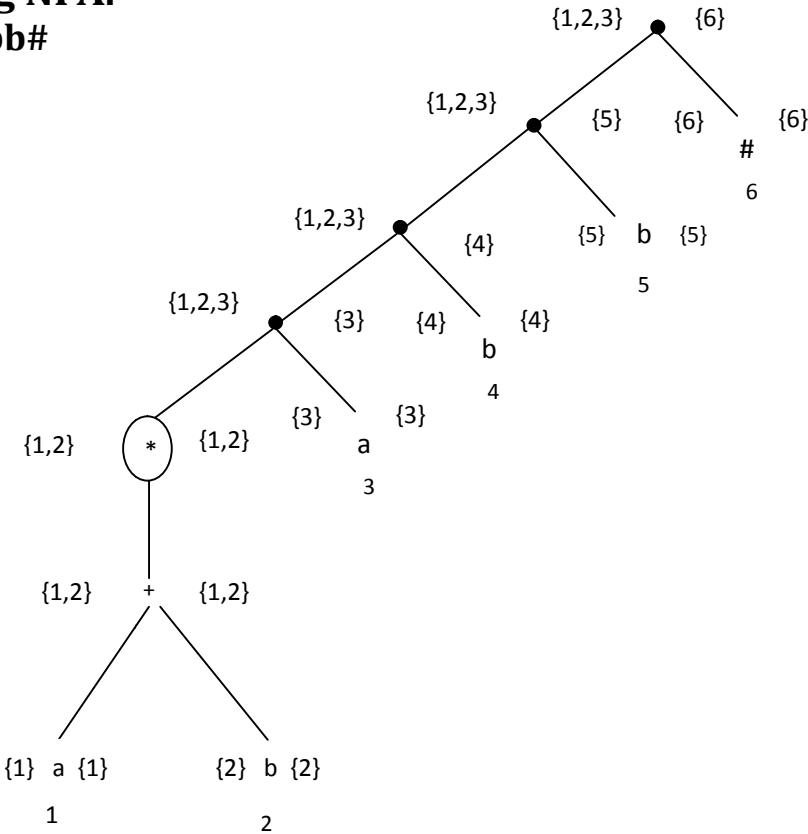


Fig.2.8. Syntax tree for  $(a+b)^*abb\#$

- To find followpos traverse concatenation and star node in depth first search order.

1. i=lastpos(c1)={5} firstpos(c2)={6} followpos(i)=firstpos(c2) followpos(5)={6}	2. i=lastpos(c1)={4} firstpos(c2)={5} followpos(i)=firstpos(c2) followpos(4)={5}	3. i=lastpos(c1)={3} firstpos(c2)={4} followpos(i)=firstpos(c2) followpos(3)={4}
4. i=lastpos(c1)={1,2} firstpos(c2)={3} followpos(i)=firstpos(c2) followpos(1)={3} followpos(2)={3}	5. i=lastpos(c1)={1,2} firstpos(c1)={1,2} followpos(i)=firstpos(c1) followpos(1)={1,2} followpos(2)={1,2}	

Position	Followpos(i)
1	{1,2,3}
2	{1,2,3}
3	{4}
4	{5}
5	{6}

**Table 2.6. follow pos table**

Construct DFA

Initial node = firstpos (root node) = {1,2,3} -- A

$$\begin{aligned}\delta(A,a) &= \text{followpos}(1) \cup \text{followpos}(3) \\ &= \{1,2,3\} \cup \{4\} \\ &= \{1,2,3,4\} \text{ --- B}\end{aligned}$$

$$\begin{aligned}\delta(A,b) &= \text{followpos}(2) \\ &= \{1,2,3\} \text{ ---- A}\end{aligned}$$

$$\begin{aligned}\delta(B,a) &= \text{followpos}(1) \cup \text{followpos}(3) \\ &= \{1,2,3\} \cup \{4\} \\ &= \{1,2,3,4\} \text{ --- B}\end{aligned}$$

$$\begin{aligned}\delta(B,b) &= \text{followpos}(2) \cup \text{followpos}(4) \\ &= \{1,2,3\} \cup \{5\} \\ &= \{1,2,3,5\} \text{ --- C}\end{aligned}$$

$$\begin{aligned}\delta(C,a) &= \text{followpos}(1) \cup \text{followpos}(3) \\ &= \{1,2,3\} \cup \{4\} \\ &= \{1,2,3,4\} \text{ --- B}\end{aligned}$$

$$\begin{aligned}\delta(C,b) &= \text{followpos}(2) \cup \text{followpos}(5) \\ &= \{1,2,3\} \cup \{6\}\end{aligned}$$

$$= \{1,2,3,6\} \text{ --- D}$$

$$\delta(D,a) = \text{followpos}(1) \cup \text{followpos}(3)$$

$$= \{1,2,3\} \cup \{4\}$$

$$= \{1,2,3,4\} \text{ --- B}$$

$$\delta(D,b) = \text{followpos}(2) = \{1,2,3\} \text{ --- A}$$

Transition Table		
States	a	b
A	B	A
B	B	C
C	B	D
D	B	A

Table 2.7. Transition table for  $(a+b)^*abb$

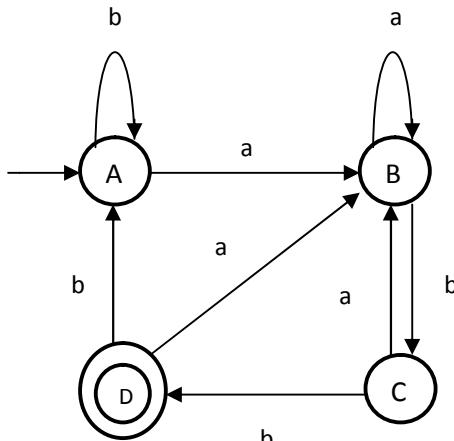
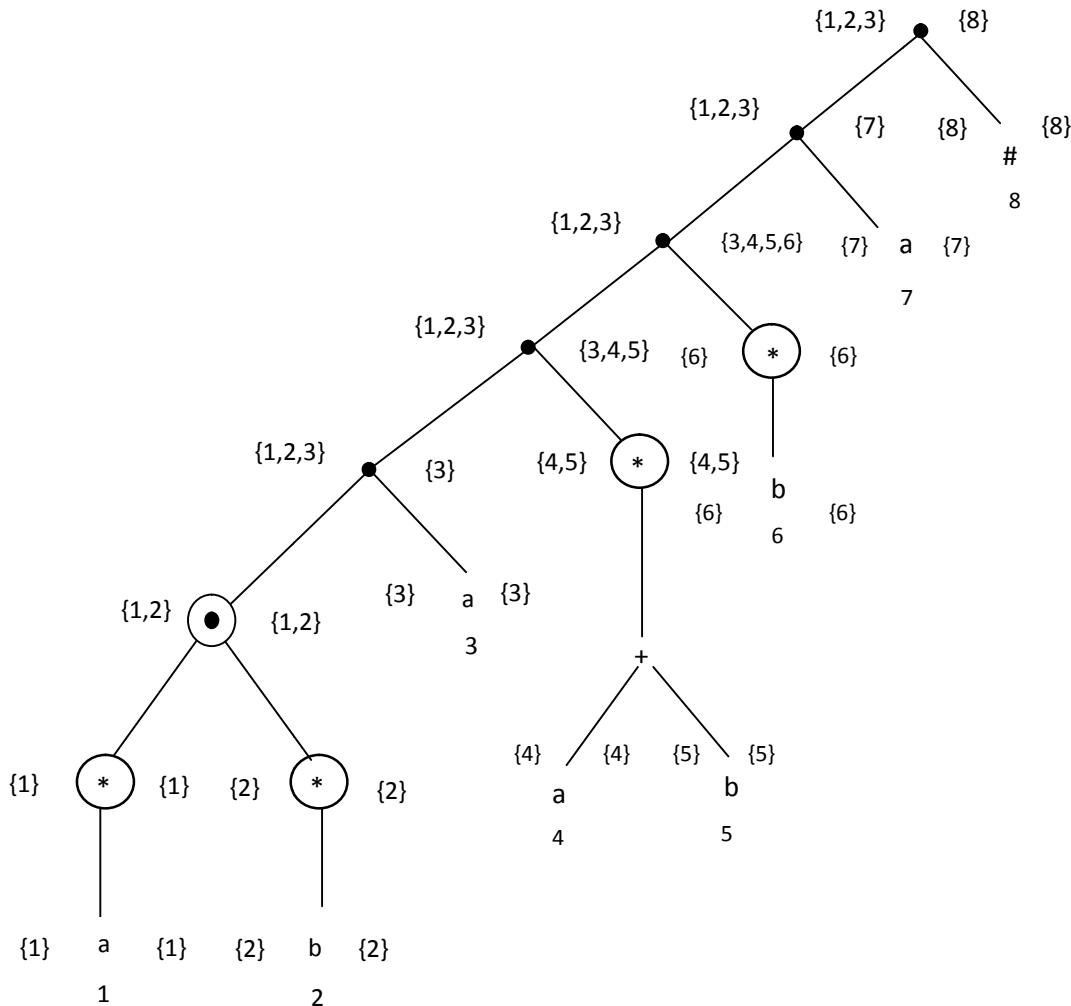


Fig.2.9. DFA for  $(a+b)^*abb$

**Ex:2  $a^*b^*a(a \setminus b)^*b^*a\#$**



**Fig.2.10. Syntax tree for  $a^*b^*a(a \setminus b)^*b^*a\#$**

- To find followpos traverse concatenation and star node in depth first search order

1. i=lastpos(c1)={7} firstpos(c2)={8} followpos(i)=firstpos(c2) followpos(7)={8}	2. i=lastpos(c1)={3,4,5,6} firstpos(c2)={7} followpos(i)=firstpos(c2) followpos(3)={7} followpos(4)={7} followpos(5)={7} followpos(6)={7}	3. i=lastpos(c1)={3,4,5} firstpos(c2)={6} followpos(i)=firstpos(c2) followpos(3)={6} followpos(4)={6} followpos(5)={6}
4. i=lastpos(c1)={3} firstpos(c2)={4,5} followpos(i)=firstpos(c2)	5. i=lastpos(c1)={1,2} firstpos(c2)={3} followpos(i)=firstpos(c2)	6. i=lastpos(c1)={1} firstpos(c2)={2} followpos(i)=firstpos(c2)

followpos(3)={4,5}	followpos(1)={3} followpos(2)={3}	followpos(1)={2}
7. i=lastpos(c1)={1} firstpos(c1)={1} followpos(i)=firstpos(c1) followpos(1)={1}	8. i=lastpos(c1)={2} firstpos(c1)={2} followpos(i)=firstpos(c1) followpos(2)={2}	9. i=lastpos(c1)={4,5} firstpos(c1)={4,5} followpos(i)=firstpos(c1) followpos(4)={4,5} followpos(5)={4,5}
9. i=lastpos(c1)={6} firstpos(c1)={6} followpos(i)=firstpos(c1) followpos(6)={6}		

n	followpos(n)
1	{1,2,3}
2	{2,3}
3	{4,5,6,7}
4	{4,5,6,7}
5	{4,5,6,7}
6	{6,7}
7	{8}

Table 2.8. follow pos table

Construct DFA

Initial node = firstpos (root node)= {1,2,3} -- A

$$\begin{aligned}\delta(A,a) &= \text{followpos}(1) \cup \text{followpos}(3) \\ &= \{1,2,3\} \cup \{4,5,6,7\} \\ &= \{1,2,3,4,5,6,7\} \text{ ---B}\end{aligned}$$

$$\begin{aligned}\delta(A,b) &= \text{followpos}(2) \\ &= \{2,3\} \text{ ---C}\end{aligned}$$

$$\begin{aligned}\delta(B,a) &= \text{followpos}(1) \cup \text{followpos}(3) \cup \text{followpos}(4) \cup \text{followpos}(7) \\ &= \{1,2,3\} \cup \{4,5,6,7\} \cup \{8\} \cup \{4,5,6,7\} \\ &= \{1,2,3,4,5,6,7,8\} \text{ ---D}\end{aligned}$$

$$\begin{aligned}\delta(B,b) &= \text{followpos}(2) \cup \text{followpos}(5) \cup \text{followpos}(6) \\ &= \{2,3\} \cup \{4,5,6,7\} \cup \{4,5,6,7\} \\ &= \{2,3,4,5,6,7\} \text{ ---E}\end{aligned}$$

$$\begin{aligned}\delta(C,a) &= \text{followpos}(3) \\ &= \{4,5,6,7\} \text{ ---F}\end{aligned}$$

$$\begin{aligned}\delta(C,b) &= \text{followpos}(2) \\ &= \{2,3\} \text{ ---C}\end{aligned}$$

$$\begin{aligned}\delta(D,a) &= \text{followpos}(1) \cup \text{followpos}(3) \cup \text{followpos}(7) \cup \text{followpos}(4) \\ &= \{1,2,3\} \cup \{4,5,6,7\} \cup \{8\} \cup \{4,5,6,7\} \\ &= \{1,2,3,4,5,6,7,8\} \text{---D}\end{aligned}$$

$$\begin{aligned}\delta(D,b) &= \text{followpos}(2) \cup \text{followpos}(5) \cup \text{followpos}(6) \\ &= \{2,3\} \cup \{4,5,6,7\} \cup \{4,5,6,7\} \\ &= \{2,3,4,5,6,7\} \text{---E}\end{aligned}$$

$$\begin{aligned}\delta(E,a) &= \text{followpos}(3) \cup \text{followpos}(4) \cup \text{followpos}(7) \\ &= \{4,5,6,7\} \cup \{4,5,6,7\} \cup \{8\} \\ &= \{4,5,6,7,8\} \text{---G}\end{aligned}$$

$$\begin{aligned}\delta(E,b) &= \text{followpos}(2) \cup \text{followpos}(5) \cup \text{followpos}(6) \\ &= \{2,3\} \cup \{4,5,6,7\} \cup \{4,5,6,7\} \\ &= \{2,3,4,5,6,7\} \text{---E}\end{aligned}$$

$$\begin{aligned}\delta(F,a) &= \text{followpos}(4) \cup \text{followpos}(7) \\ &= \{4,5,6,7\} \cup \{8\} \\ &= \{4,5,6,7,8\} \text{---G}\end{aligned}$$

$$\begin{aligned}\delta(F,b) &= \text{followpos}(5) \cup \text{followpos}(6) \\ &= \{4,5,6,7\} \cup \{4,5,6,7\} \\ &= \{4,5,6,7\} \text{---F}\end{aligned}$$

$$\begin{aligned}\delta(G,a) &= \text{followpos}(4) \cup \text{followpos}(7) \\ &= \{4,5,6,7\} \cup \{8\} \\ &= \{4,5,6,7,8\} \text{---G}\end{aligned}$$

$$\begin{aligned}\delta(G,b) &= \text{followpos}(5) \cup \text{followpos}(6) \\ &= \{4,5,6,7\} \cup \{4,5,6,7\} \\ &= \{4,5,6,7\} \text{---F}\end{aligned}$$

Transition table:

Transition table		
	a	b
A	B	C
B	D	E
C	F	C
D	D	E
E	G	E
F	G	F
G	G	F

Table 2.9. Transition table for  $a^*b^*a(a|b)^*b^*a\#$

DFA:

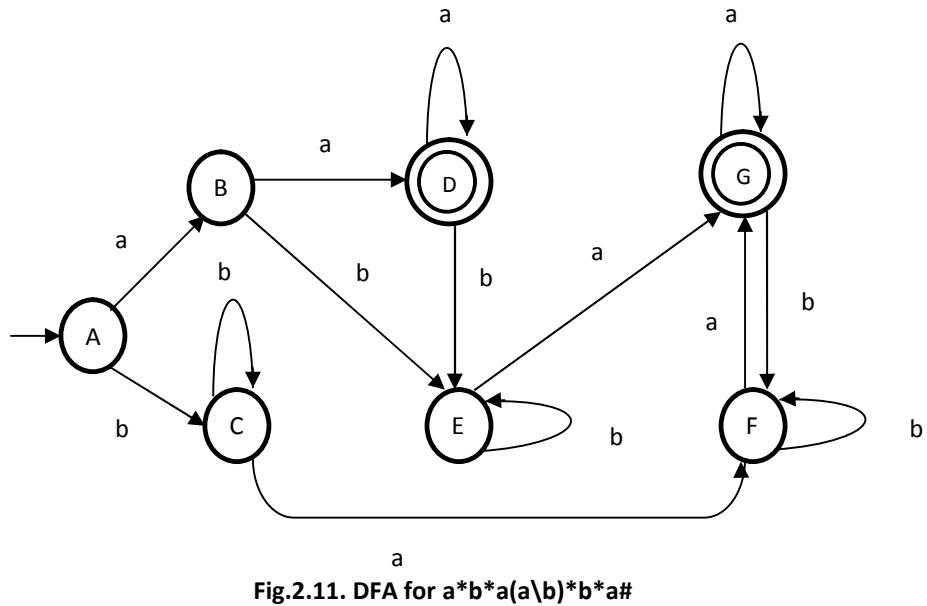


Fig.2.11. DFA for  $a^*b^*a(a \setminus b)^*b^*a\#$

## 1. Role of Parser.

- In our compiler model, the parser obtains a string of tokens from lexical analyzer, as shown in fig. 3.1.1.

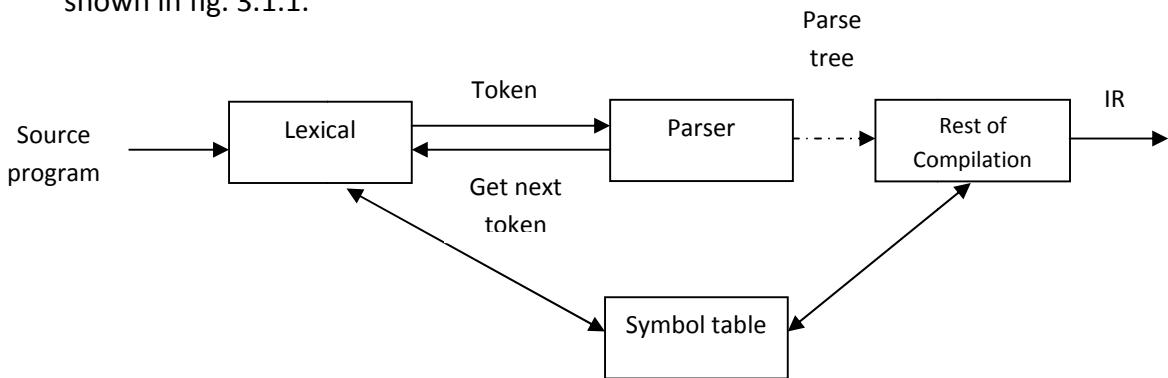


Fig.3.1.1. Position of parser in compiler model

- We expect the parser to report any syntax error. It should also recover from commonly occurring errors.
- The methods commonly used for parsing are classified as a top down or bottom up parsing.
- In top down parsing parser, build parse tree from top to bottom, while bottom up parser starts from leaves and work up to the root.
- In both the cases, the input to the parser is scanned from left to right, one symbol at a time.
- We assume the output of parser is some representation of the parse tree for the stream of tokens produced by the lexical analyzer.

## 2. Difference between syntax tree and Parse tree.

### Syntax tree v/s Parse tree

No.	Parse Tree	Syntax Tree
1	Interior nodes are non-terminals, leaves are terminals.	Interior nodes are “operators”, leaves are operands.
2	Rarely constructed as a data structure.	When representing a program in a tree structure usually use a syntax tree.
3	Represents the concrete syntax of a program.	Represents the abstract syntax of a program (the semantics).

Table 3.1.1. Difference between syntax tree & Parse tree

- Example: Consider grammar following grammar,  
 $E \rightarrow E + E$   
 $E \rightarrow E^* E$   
 $E \rightarrow Id$
- Figure 3.1.2. Shows the syntax tree and parse tree for string id + id\*id.

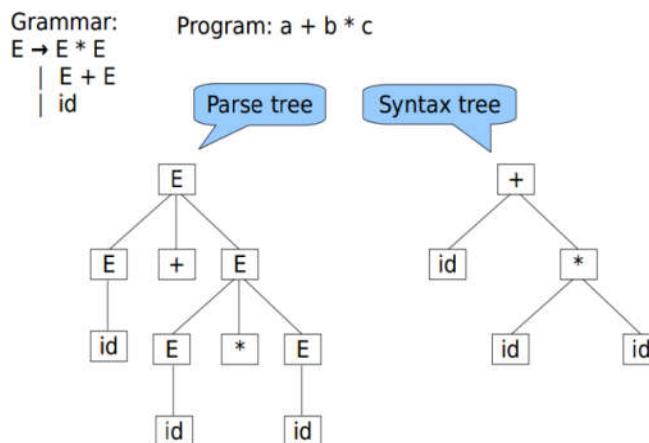


Fig.3.1.2. Syntax tree and parse tree

### 3. Types of Derivations. (Leftmost & Rightmost)

There are mainly two types of derivations,

1. Leftmost derivation
2. Rightmost derivation

Let Consider the grammar with the production  $S \rightarrow S+S \mid S-S \mid S*S \mid S/S \mid (S) \mid a$

Left Most Derivation	Right Most Derivation
A derivation of a string $W$ in a grammar $G$ is a left most derivation if at every step the left most non terminal is replaced.	A derivation of a string $W$ in a grammar $G$ is a right most derivation if at every step the right most non terminal is replaced.
Consider string $a^*a-a$ $S \rightarrow S-S$ $S \rightarrow S-S$ $S \rightarrow a^*S-S$ $S \rightarrow a^*a-S$ $S \rightarrow a^*a-a$	Consider string: $a-a/a$ $S \rightarrow S-S$ $S \rightarrow S-S$ $S \rightarrow S-a$ $S \rightarrow a/a$ $S \rightarrow a-a$
Equivalent left most derivation tree	Equivalent Right most derivation tree

```

graph TD
    S1[S] --> S2[S]
    S1 --> S3[S]
    S2 --> S4[S]
    S2 --> S5[S]
    S4 --> a1[a]
    S5 --> a2[a]
  
```

```

graph TD
    S1[S] --> S2[S]
    S1 --> S3[S]
    S3 --> S4[S]
    S3 --> S5[S]
    S4 --> a1[a]
    S5 --> a2[a]
  
```

Table 3.1.2. Difference between Left most Derivation & Right most Derivation

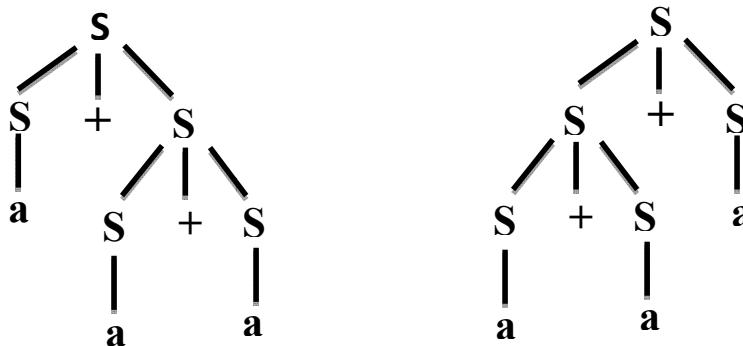
### 4. Explain Ambiguity with example.

- An ambiguous grammar is one that produces more than one leftmost or more than one rightmost derivation for the same sentence.

1) Prove that given grammar is ambiguous.  $S \rightarrow S+S / S-S / S^*S / S/S / (S)/a$  (IMP)

String : a+a+a

$S \rightarrow S+S$	$S \rightarrow S+S$
$a+S$	$S+S+S$
$a+S+S$	$a+S+S$
$a+a+S$	$a+a+S$
$a+a+a$	$a+a+a$



- Here we have two left most derivation hence, proved that above grammar is ambiguous.

2) Prove that  $S \rightarrow a \mid Sa \mid bSS \mid SSb \mid SbS$  is ambiguous

String: baaab

$S \rightarrow bSS$	$S \rightarrow SSb$
$baS$	$bSSSb$
$baSSb$	$baSSb$
$baaSb$	$baaSb$
$baaab$	$baaab$

- Here we have two left most derivation hence, proved that above grammar is ambiguous.

### 5. Elimination of left recursion.

- A grammar is said to be left recursive if it has a non terminal A such that there is a derivation  $A \rightarrow A\alpha$  for some string  $\alpha$ .
- Top down parsing methods cannot handle left recursive grammar, so a transformation that eliminates left recursion is needed.

#### Algorithm to eliminate left recursion

1. Assign an ordering  $A_1, \dots, A_n$  to the nonterminals of the grammar.

2. for  $i:=1$  to  $n$  do

begin

for  $j:=1$  to  $i-1$  do

begin

replace each production of the form  $A_i \rightarrow A_i\gamma$

by the productions  $A_i \rightarrow \delta_1\gamma | \delta_2\gamma | \dots | \delta_k\gamma$

where  $A_j \rightarrow \delta_1 | \delta_2 | \dots | \delta_k$  are all the current  $A_j$  productions;

end

eliminate the intermediate left recursion among the  $A_i$ -productions

end

- Example 1 : Consider the following grammar,

$$E \rightarrow E + T / T$$

$$T \rightarrow T^* F / F$$

$$F \rightarrow (E) / id$$

Eliminate immediate left recursion from above grammar then we obtain,

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

$$F \rightarrow (E) | id$$

- Example 2 : Consider the following grammar,

$$S \rightarrow Aa | b$$

$$A \rightarrow Ac | Sd | \epsilon$$

Here, non terminal S is left recursive because  $S \rightarrow Aa \rightarrow Sda$ , but it is not immediately left recursive.

$$S \rightarrow Aa | b$$

$$A \rightarrow Ac | Aad | bd | \epsilon$$

Now, remove left recursion

$$S \rightarrow Aa | b$$

$$A \rightarrow bdA' | A'$$

$$A \rightarrow cA' | adA' | \epsilon$$

## 6. Left factoring.

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.

**Algorithm to left factor a grammar**

Input: Grammar G

Output: An equivalent left factored grammar.

1. For each non terminal A find the longest prefix  $\alpha$  common to two or more of its alternatives.

2. If  $\alpha \neq E$ , i.e., there is a non trivial common prefix, replace all the A productions  $A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \dots | \alpha\beta_n | \gamma$  where  $\gamma$  represents all alternatives that do not begin with  $\alpha$  by

$$A \Rightarrow \alpha A' | \gamma$$

$$A' \Rightarrow \beta_1 | \beta_2 | \dots | \beta_n$$

Here  $A'$  is new non terminal. Repeatedly apply this transformation until no two alternatives for a non-terminal have a common prefix.

- EX1: Perform left factoring on following grammar,  
 $A \rightarrow xByA \mid xByAzA \mid a$   
 $B \rightarrow b$   
Left factored, the grammar becomes  
 $A \rightarrow xByAA' \mid a$   
 $A' \rightarrow zA \mid \epsilon$   
 $B \rightarrow b$
- EX2: Perform left factoring on following grammar,  
 $S \rightarrow iEtS \mid iEtSeS \mid a$   
 $E \rightarrow b$   
Left factored, the grammar becomes  
 $S \rightarrow iEtSS' \mid a$   
 $S' \rightarrow eS \mid \epsilon$   
 $E \rightarrow b$

## 7. Types of Parsing.

- Parsing or syntactic analysis is the process of analyzing a string of symbols according to the rules of a formal grammar.
- Parsing is a technique that takes input string and produces output either a parse tree if string is valid sentence of grammar, or an error message indicating that string is not a valid sentence of given grammar. Types of parsing are,
  1. **Top down parsing:** In top down parsing parser build parse tree from top to bottom.
  2. **Bottom up parsing:** While bottom up parser starts from leaves and work up to the root.

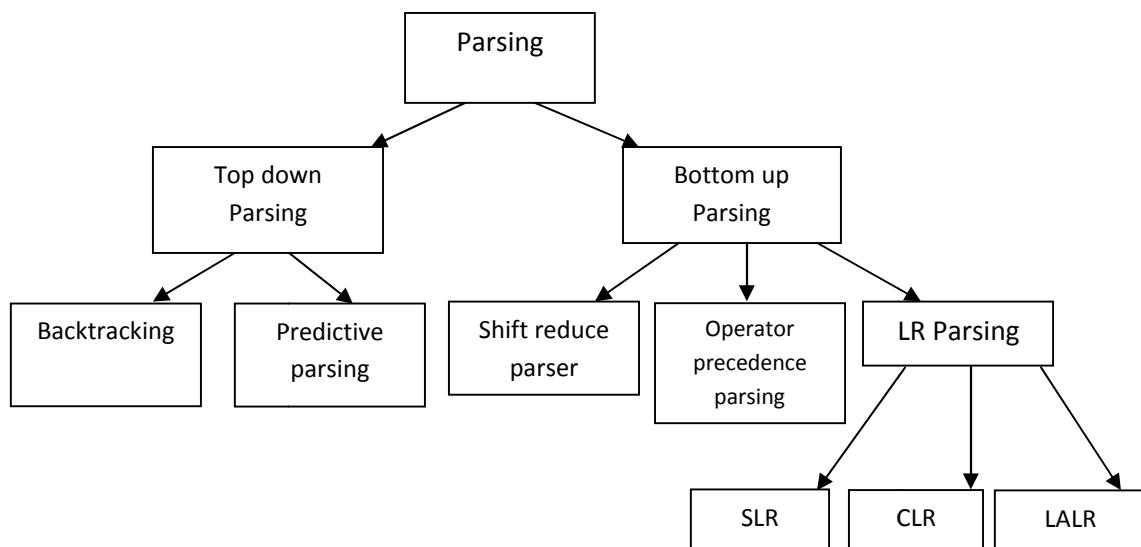


Fig.3.1.3 Parsing Techniques

### 8. Recursive Decent Parsing.

- A top down parsing that executes a set of recursive procedure to process the input without backtracking is called recursive decent parser.
- There is a procedure for each non terminal in the grammar.
- Consider RHS of any production rule as definition of the procedure.
- As it reads expected input symbol, it advances input pointer to next position.

Example:

```

 $E \rightarrow T \{+T\}^*$ 
 $T \rightarrow V \{^*V\}^*$ 
 $V \rightarrow id$ 

Procedure proc_E: (tree_root);
    var
        a, b : pointer to a tree node;
    begin
        proc_T(a);
        While (nextsymb = '+') do
            nextsymb = next source symbol;
            proc_T(b);
            a= Treebuild ('+', a, b);
            tree_root= a;
        return;
    end proc_E;
Procedure proc_T: (tree_root);
    var
        a, b : pointer to a tree node;
    begin
        proc_V(a);
        While (nextsymb = '*') do
            nextsymb = next source symbol;
            proc_V(b);
            a= Treebuild ('*', a, b);
            tree_root= a;
        return;
    end proc_T;
Procedure proc_V: (tree_root);
    var
        a : pointer to a tree node;
    begin
        If (nextsymb = 'id') then
            nextsymb = next source symbol;
            tree_root= tree_build(id, , );
        else print "Error";
    
```

```
return;
```

```
end proc_V;
```

### Advantages

- It is exceptionally simple.
- It can be constructed from recognizers simply by doing some extra work.

### Disadvantages

- It is time consuming method.
- It is difficult to provide good error messages.

## 9. Predictive parsing. OR

### LL(1) Parsing.

- This top-down parsing is non-recursive. LL (1) – the first L indicates input is scanned from left to right. The second L means it uses leftmost derivation for input string and 1 means it uses only input symbol to predict the parsing process.
- The block diagram for LL(1) parser is given below,

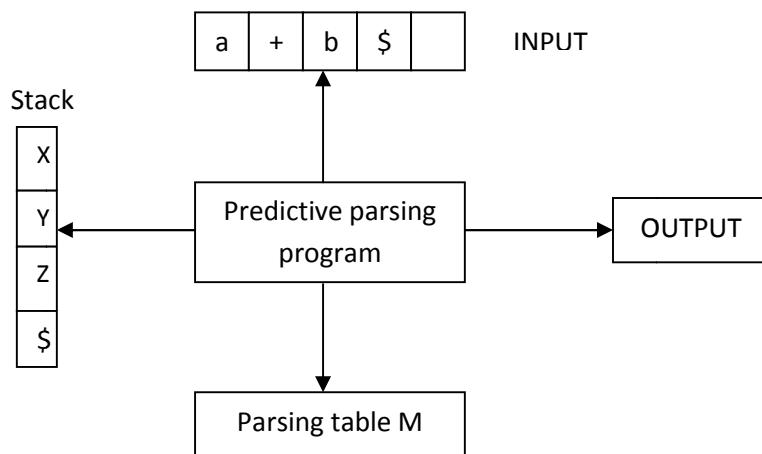


Fig.3.1.4 Model of nonrecursive predictive parser

- The data structure used by LL(1) parser are input buffer, stack and parsing table.
- The parser works as follows,
- The parsing program reads top of the stack and a current input symbol. With the help of these two symbols parsing action can be determined.
- The parser consult the table  $M[A, a]$  each time while taking the parsing actions hence this type of parsing method is also called table driven parsing method.
- The input is successfully parsed if the parser reaches the halting configuration. When the stack is empty and next token is  $\$$  then it corresponds to successful parsing.

### Steps to construct LL(1) parser

1. Remove left recursion / Perform left factoring.
2. Compute FIRST and FOLLOW of nonterminals.
3. Construct predictive parsing table.
4. Parse the input string with the help of parsing table.

**Example:**

$$E \rightarrow E + T / T$$

$$T \rightarrow T * F / F$$

$$F \rightarrow (E) / id$$

Step1: Remove left recursion

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

Step2: Compute FIRST & FOLLOW

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

Table 3.1.3 first & follow set

Step3: Predictive Parsing Table

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

Table 3.1.4 predictive parsing table

Step4: Parse the string

Stack	Input	Action
\$E	id+id*id\$	
\$E'T	id+id*id\$	$E \rightarrow TE'$
\$ E'T'F	id+id*id\$	$T \rightarrow FT'$
\$ E'T'id	id+id*id\$	$F \rightarrow id$
\$ E'T'	+id*id\$	
\$ E'	+id*id\$	$T' \rightarrow \epsilon$
\$ E'T+	+id*id\$	$E' \rightarrow +TE'$
\$ E'T	id*id\$	
\$ E'T'F	id*id\$	$T \rightarrow FT'$
\$ E'T'id	id*id\$	$F \rightarrow id$
\$ E'T'	*id\$	
\$ E'T'F*	*id\$	$T' \rightarrow *FT'$
\$ E'T'F	id\$	
\$ E'T'id	id\$	$F \rightarrow id$

\$ E'T'	\$	
\$ E'	\$	$E' \rightarrow \epsilon$
\$	\$	$E' \rightarrow \epsilon$

**Table 3.1.5. moves made by predictive parse**

### 10. Error recovery in predictive parsing.

- Panic mode error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing token appears.
- Its effectiveness depends on the choice of synchronizing set.
- Some heuristics are as follows:
  - ✓ Insert ‘synch’ in FOLLOW symbol for all non terminals. ‘synch’ indicates resume the parsing. If entry is “synch” then non terminal on the top of the stack is popped in an attempt to resume parsing.
  - ✓ If we add symbol in FIRST (A) to the synchronizing set for a non terminal A, then it may be possible to resume parsing if a symbol in FIRST(A) appears in the input.
  - ✓ If a non terminal can generate the empty string, then the production deriving the  $\epsilon$  can be used as a default.
  - ✓ If parser looks entry M[A,a] and finds that it is blank then i/p symbol a is skipped.
  - ✓ If a token on top of the stack does not match i/p symbol then we pop token from the stack.
- Consider the grammar given below:

```

E ::= TE'
E' ::= +TE' | ε
T ::= FT'
T' ::= *FT' | ε
F ::= (E)|id
    
```

- ✓ Insert ‘synch’ in FOLLOW symbol for all non terminals.

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

**Table 3.1.6. Follow set of non terminals**

NT	Input Symbol					
	id	+	*	(	)	\$
E	$E \Rightarrow TE'$			$E \Rightarrow TE'$	synch	Synch
E'		$E' \Rightarrow +TE'$			$E' \Rightarrow \epsilon$	$E' \Rightarrow \epsilon$
T	$T \Rightarrow FT'$	synch		$T \Rightarrow FT'$	Synch	Synch
T'		$T' \Rightarrow \epsilon$	$T' \Rightarrow *FT'$		$T' \Rightarrow \epsilon$	$T' \Rightarrow \epsilon$
F	$F \Rightarrow <\text{id}>$	synch	Synch	$F \Rightarrow (E)$	synch	synch

**Table 3.1.7. Synchronizing token added to parsing table**

Stack	Input	Remarks
\$E	)id*+id\$	Error, skip )
\$E	id*+id\$	
\$E' T	id*+id\$	
\$E' T' F	id*+id\$	
\$E' T' id	id*+id\$	
\$E' T'	*+id\$	
\$E' T' F*	*+id\$	
\$E' T' F	+id\$	Error, M[F,+]=synch
\$E' T'	+id\$	F has been popped.
\$E'	+id\$	
\$E' T+	+id\$	
\$E' T	id\$	
\$E' T' F	id\$	
\$E' T' id	id\$	
\$E' T'	\$	
\$E'	\$	
\$	\$	

Table 3.1.8. Parsing and error recovery moves made by predictive parser

## 11. Explain Handle and handle pruning.

**Handle:** A “handle” of a string is a substring of the string that matches the right side of a production, and whose reduction to the non terminal of the production is one step along the reverse of rightmost derivation.

**Handle pruning:** The process of discovering a handle and reducing it to appropriate Left hand side non terminal is known as handle pruning.

Right sentential form	Handle	Reducing production
id1+id2*id3	id1	$E \rightarrow id$
E+id2*id3	id2	$E \rightarrow id$
E+E*id3	id3	$E \rightarrow id$
E+E*E	E*E	$E \rightarrow E^*E$
E+E	E+E	$E \rightarrow E+E$
E		

Table 3.1.9. Handles

## 12. Shift reduce Parsing.

- The shift reduce parser performs following basic operations,
- Shift: Moving of the symbols from input buffer onto the stack, this action is called shift.
- Reduce: If handle appears on the top of the stack then reduction of it by appropriate rule is done. This action is called reduce action.
- Accept: If stack contains start symbol only and input buffer is empty at the same time then that action is called accept.

- Error: A situation in which parser cannot either shift or reduce the symbols, it cannot even perform accept action then it is called error action.

Example: Consider the following grammar,

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

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

$$F \rightarrow id$$

Perform shift reduce parsing for string  $id + id * id$ .

Stack	Input buffer	Action
\$	$id + id * id \$$	Shift
\$id	$+ id * id \$$	Reduce $F \rightarrow id$
\$F	$+ id * id \$$	Reduce $T \rightarrow F$
\$T	$+ id * id \$$	Reduce $E \rightarrow T$
\$E	$+ id * id \$$	Shift
\$E+	$id * id \$$	shift
\$E+ id	$* id \$$	Reduce $F \rightarrow id$
\$E+F	$* id \$$	Reduce $T \rightarrow F$
\$E+T	$* id \$$	Shift
\$E+T*	$id \$$	Shift
\$E+T*id	\$	Reduce $F \rightarrow id$
\$E+T*T	\$	Reduce $T \rightarrow T * F$
\$E+T	\$	Reduce $E \rightarrow E + T$
\$E	\$	Accept

Table 3.1.10. Configuration of shift reduce parser on input  $id + id * id$

### 13. Operator Precedence Parsing.

- **Operator Grammar:** A Grammar in which there is no  $\epsilon$  in RHS of any production or no adjacent non terminals is called operator precedence grammar.
- In operator precedence parsing, we define three disjoint precedence relations  $<$ ,  $>$  and  $=$  between certain pair of terminals.

Relation	Meaning
$a < b$	a “yields precedence to” b
$a = b$	a “has the same precedence as” b
$a > b$	a “takes precedence over” b

Table 3.1.11. Precedence between terminal a & b

#### Leading:-

Leading of a nonterminal is the first terminal or operator in production of that nonterminal.

#### Trailing:-

Trailing of a nonterminal is the last terminal or operator in production of that nonterminal

#### Example:

$$E \rightarrow E + T / T$$

$$T \rightarrow T^* F / F$$

$$F \rightarrow id$$

Step-1: Find leading and trailing of NT.

Leading

$$(E) = \{+, *, id\}$$

$$(T) = \{*, id\}$$

$$(F) = \{id\}$$

Trailing

$$(E) = \{+, *, id\}$$

$$(T) = \{*, id\}$$

$$(F) = \{id\}$$

Step-2: Establish Relation

1.  $a <^\cdot b$

$$Op \cdot NT \rightarrow Op <^\cdot \text{Leading}(NT)$$

$$+T \quad + <^\cdot \{*, id\}$$

$$*F \quad * <^\cdot \{id\}$$

2.  $a >^\cdot b$

$$NT \cdot Op \rightarrow \text{Traling}(NT) >^\cdot Op$$

$$E+ \quad \{+, *, id\} > +$$

$$T^* \quad \{*, id\} > *$$

3.  $\$ <^\cdot \{+, *, id\}$

4.  $\{+, *, id\} > \$$

Step-3: Creation of table

	+	*	id	\$
+	'>	'<	'<	'>
*	'>	'>	'<	'>
id	'>	'>		'>
\$	'<	'<	'<	

Table 3.1.12. precedence table

Step-4: Parsing of the string using precedence table.

We will follow following steps to parse the given string,

1. Scan the input string until first '<>' is encountered.
2. Scan backward until '<' is encountered.
3. The handle is string between '<' And '>'.

\$ < Id > + < Id > * < Id > \$	Handle id is obtained between '<' '>'. Reduce this by F->id
\$ F + < Id > * < Id > \$	Handle id is obtained between '<' '>'. Reduce this by F->id
\$ F + F * < Id > \$	Handle id is obtained between '<' '>'. Reduce this by F->id
\$ F + F * F \$	Perform appropriate reductions of all non terminals.
\$ E + T * F \$	Remove all non terminal
\$ + * \$	Place relation between operators
\$ < + < * > \$	The '*' operator is surrounded by '<' '>'. This

	indicates * becomes handle we have to reduce T*F.
\$ < + > \$	+ becomes handle. Hence reduce E+T.
\$\$	Parsing Done

Table 3.1.13. moves made by operator precedence parser

### Operator Precedence Function

Algorithm for Constructing Precedence Functions

1. Create functions  $f_a$  and  $g_a$  for each  $a$  that is terminal or \$.
  2. Partition the symbols in as many as groups possible, in such a way that  $f_a$  and  $g_b$  are in the same group if  $a = b$ .
  3. Create a directed graph whose nodes are in the groups, next for each symbols  $a$  and  $b$  do:
    - (a) if  $a < b$ , place an edge from the group of  $g_b$  to the group of  $f_a$ .
    - (b) if  $a \rightarrow b$ , place an edge from the group of  $f_a$  to the group of  $g_b$ .
  4. If the constructed graph has a cycle then no precedence functions exist. When there are no cycles collect the length of the longest paths from the groups of  $f_a$  and  $g_b$  respectively.
- Using the algorithm leads to the following graph:

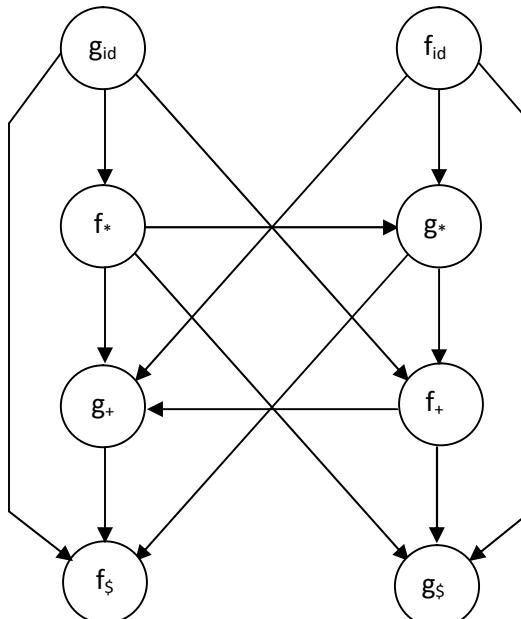


Fig. 3.1.5 Operator precedence graph

- From which we can extract the following precedence functions:

	id	+	*	\$
f	4	2	4	0
g	5	1	3	0

Table 3.1.14 precedence function

### 14. LR parsing.

- LR parsing is most efficient method of bottom up parsing which can be used to parse large class of context free grammar.
- The technique is called LR( $k$ ) parsing; the “L” is for left to right scanning of input symbol, the “R” for constructing right most derivation in reverse, and the  $k$  for the number of input symbols of lookahead that are used in making parsing decision.
- There are three types of LR parsing,
  1. SLR (Simple LR)
  2. CLR (Canonical LR)
  3. LALR (Lookahead LR)
- The schematic form of LR parser is given in figure 3.1.6.
- The structure of input buffer for storing the input string, a stack for storing a grammar symbols, output and a parsing table comprised of two parts, namely action and goto.

#### Properties of LR parser

- LR parser can be constructed to recognize most of the programming language for which CFG can be written.
- The class of grammars that can be parsed by LR parser is a superset of class of grammars that can be parsed using predictive parsers.
- LR parser works using non back tracking shift reduce technique.
- LR parser can detect a syntactic error as soon as possible.

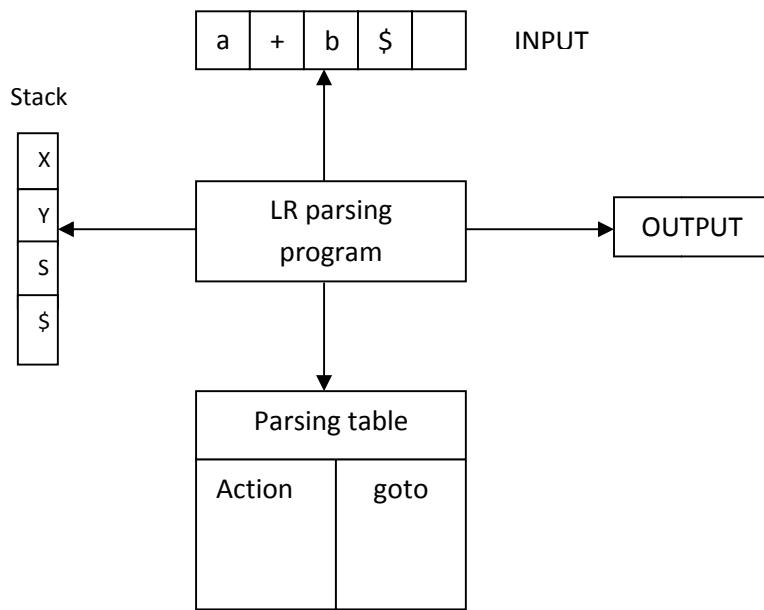


Fig.3.1.6. Model of an LR parser

### 15. Explain the following terms.

1. **Augmented grammar:** If grammar  $G$  having start symbol  $S$  then augmented grammar is the new grammar  $G'$  in which  $S'$  is a new start symbol such that  $S' \rightarrow .S$ .
2. **Kernel items:** It is a collection of items  $S' \rightarrow .S$  and all the items whose dots are not at the

left most end of the RHS of the rule.

3. **Non-Kernel items:** It is a collection of items in which dots are at the left most end of the RHS of the rule.
4. **Viable prefix:** It is a set of prefix in right sentential form of the production  $A \rightarrow \alpha$ , this set can appear on the stack during shift reduce action.

### 16. SLR Parsing.

- SLR means simple LR. A grammar for which an SLR parser can be constructed is said to be an SLR grammar.
- SLR is a type of LR parser with small parse tables and a relatively simple parser generator algorithm. It is quite efficient at finding the single correct bottom up parse in a single left to right scan over the input string, without guesswork or backtracking.
- The parsing table has two states (action, Go to).

The parsing table has four values:

- ✓ Shift S, where S is a state
- ✓ reduce by a grammar production
- ✓ accept, and
- ✓ error

Example:

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

$$T \rightarrow TF \mid F$$

$$F \rightarrow F^* \mid a \mid b$$

Augmented grammar:  $E' \rightarrow .E$

Closure(I)

$I_0 :$ $E' \rightarrow .E$ $E \rightarrow .E + T$ $E \rightarrow .T$ $T \rightarrow .TF$ $T \rightarrow .F$ $F \rightarrow .F^*$ $F \rightarrow .a$ $F \rightarrow .b$	$I_1 : \text{Go to } (I_0, E)$ $E' \rightarrow E.$ $E \rightarrow E.+T$	$I_2 : \text{Go to } (I_0, T)$ $E \rightarrow T.$ $T \rightarrow T.F$ $F \rightarrow .F^*$ $F \rightarrow .a$ $F \rightarrow .b$
$I_3 : \text{Go to } (I_0, F)$ $T \rightarrow F.$ $F \rightarrow F.^*$	$I_4 : \text{Go to } (I_0, a)$ $F \rightarrow a.$	$I_5 : \text{Go to } (I_0, b)$ $F \rightarrow b.$
$I_6 : \text{Go to } (I_1, +)$ $E \rightarrow E+.T$ $T \rightarrow .TF$ $T \rightarrow .F$ $F \rightarrow .F^*$ $F \rightarrow .F^*$ $F \rightarrow .a$	$I_7 : \text{Go to } (I_2, F)$ $T \rightarrow TF.$ $F \rightarrow F.^*$	$I_8 : \text{Go to } (I_3, ^*)$ $F \rightarrow F.^*$

$F \rightarrow .b$		
$I_9 : Go\ to\ (I_6, T)$ $E \rightarrow E + T.$ $T \rightarrow T.F$ $F \rightarrow F^*$ $F \rightarrow a$ $F \rightarrow b$		

Table 3.1.15. Canonical LR(0) collection

Follow:

Follow ( E ) : {+, \$}

Follow ( T ) : {+, a, b, \$}

Follow ( F ) : {+, \*, a, b, \$}

SLR parsing table :

state	Action					Go to		
	+	*	a	b	\$	E	T	F
0			$S_4$	$S_5$		1	2	3
1	$S_6$				Accept			
2	$R_2$		$S_4$	$S_5$	$R_2$			7
3	$R_4$	$S_8$	$R_4$	$R_4$	$R_4$			
4	$R_6$	$R_6$	$R_6$	$R_6$	$R_6$			
5	$R_6$	$R_6$	$R_6$	$R_6$	$R_6$			
6			$S_4$	$S_5$			9	3
7	$R_3$	$S_8$	$R_3$	$R_3$	$R_3$			
8	$R_5$	$R_5$	$R_5$	$R_5$	$R_5$			
9	$R_1$		$S_4$	$S_5$	$R_1$			7

Table 3.1.16. SLR Parsing table

## 17. CLR parsing.

Example :  $S \rightarrow C C$

$C \rightarrow a C \mid d$

Augmented grammar:  $S' \rightarrow .S, \$$

Closure(I)

$I_0 : S' \rightarrow .S, \$$ $S \rightarrow .CC, \$$ $C \rightarrow .a C, a \mid d$ $C \rightarrow .d, a \mid d$	$I_1 : Go\ to(I_0, S)$ $S' \rightarrow S. , \$$	$I_2 : Go\ to(I_0, C)$ $S \rightarrow C.C, \$$ $C \rightarrow .a C, \$$ $C \rightarrow .d, \$$
$I_3 : Go\ to(I_0, a)$ $C \rightarrow a.C, a \mid d$ $C \rightarrow .a C, a \mid d$	$I_4 : Go\ to(I_0, d)$ $C \rightarrow d. , a \mid d$	$I_5 : Go\ to(I_2, C)$ $S \rightarrow C.C., \$$

$C \rightarrow .d, a \mid d$		
$I_6 : Go\ to\ (I_2, a)$ $C \rightarrow a.C, \$$ $C \rightarrow .a\ C, \$$ $C \rightarrow .d, \$$	$I_7 : Go\ to\ (I_2, d)$ $C \rightarrow d., \$$	$I_8 : Go\ to\ (I_3, C)$ $C \rightarrow a\ C., a \mid d$
$I_9 : Go\ to\ (I_6, C)$ $C \rightarrow a\ C., \$$		

Table 3.1.17. Canonical LR(1) collection

Parsing table:

state	Action			Go to	
	a	d	\$	S	C
0	$S_3$	$S_4$		1	2
1			Accept		
2	$S_6$	$S_7$			5
3	$S_3$	$S_4$			8
4	$R_3$	$R_3$			
5			$R_1$		
6	$S_6$	$S_7$			9
7			$R_3$		
8	$R_2$	$R_2$			
9			$R_2$		

Table 3.1.18. CLR Parsing table

## 18. LALR Parsing.

- LALR is often used in practice because the tables obtained by it are considerably smaller than canonical LR.

**Example :**  $S \rightarrow CC$

$C \rightarrow a\ C \mid d$

Augmented grammar:  $S' \rightarrow .S, \$$

Closure( $I$ )

$I_0 : S' \rightarrow .S, \$$ $S \rightarrow .CC, \$$ $C \rightarrow .a\ C, a \mid d$ $C \rightarrow .d, a \mid d$	$I_1 : Go\ to(I_0, S)$ $S' \rightarrow S., \$$	$I_2 : Go\ to(I_0, C)$ $S \rightarrow C.C, \$$ $C \rightarrow .a\ C, \$$ $C \rightarrow .d, \$$
$I_3 : Go\ to\ (I_0, a)$ $C \rightarrow a.C, a \mid d$ $C \rightarrow .a\ C, a \mid d$ $C \rightarrow .d, a \mid d$	$I_4 : Go\ to\ (I_0, d)$ $C \rightarrow d., a \mid d$	$I_5 : Go\ to\ (I_2, C)$ $S \rightarrow C.C., \$$
$I_6 : Go\ to\ (I_2, a)$ $C \rightarrow a.C, \$$ $C \rightarrow .a\ C, \$$ $C \rightarrow .d, \$$	$I_7 : Go\ to\ (I_2, d)$ $C \rightarrow d., \$$	$I_8 : Go\ to\ (I_3, C)$ $C \rightarrow a\ C., a \mid d$

$I_9 : \text{ Go to } (I_6, C)$ $C \rightarrow a \ C \ , \$$		
---	--	--

**Table 3.1.19. Canonical LR(1) collection**

Now we will merge state 3, 6 then 4, 7 and 8, 9.

$$\begin{aligned}
I_{36} : C &\rightarrow a.C, a \mid d \mid \$ \\
&\rightarrow .a \ C, a \mid d \mid \$ \\
&\rightarrow .d, a \mid d \mid \$ \\
I_{47} : C &\rightarrow d., a \mid d \mid \$ \\
I_{89} : C &\rightarrow a.C., a \mid d \mid \$ 
\end{aligned}$$

Parsing table:

State	Action			Go to	
	a	d	\$	S	C
0	$S_{36}$	$S_{47}$		1	2
1			Accept		
2	$S_{36}$	$S_{47}$			5
36	$S_{36}$	$S_{47}$			8 9
47	$R_3$	$R_3$	$R_3$		
5			$R_1$		
89	$R_2$	$R_2$	$R_2$		

**Table 3.1.20. LALR parsing table**

### 19. Error recovery in LR parsing.

- An LR parser will detect an error when it consults the parsing action table and finds an error entry.
- Consider the grammar,  $E \rightarrow E+E \mid E^*E \mid (E) \mid id$

$I_0:$ $E' \rightarrow .E$ $E \rightarrow .E+E$ $E \rightarrow .E^*E$ $E \rightarrow .(E)$ $E \rightarrow .id$	$I_1:$ $E' \rightarrow E.$ $E \rightarrow E+E$ $E \rightarrow E.^*E$	$I_2:$ $E \rightarrow (E.)$ $E \rightarrow .E+E$ $E \rightarrow .E^*E$ $E \rightarrow .(E)$ $E \rightarrow .id$	$I_3:$ $E \rightarrow id.$	$I_4:$ $E \rightarrow E+E$ $E \rightarrow .E+E$ $E \rightarrow .E^*E$ $E \rightarrow .(E)$ $E \rightarrow .id$
$I_5:$ $E \rightarrow E^*.E$ $E \rightarrow .E+E$ $E \rightarrow .E^*E$ $E \rightarrow .(E)$ $E \rightarrow .id$	$I_6:$ $E \rightarrow (E.)$ $E \rightarrow E+E$ $E \rightarrow E.^*E$	$I_7:$ $E \rightarrow E+E$ $E \rightarrow E.+E$ $E \rightarrow E.^*E$	$I_8:$ $E \rightarrow E^*E.$ $E \rightarrow E.+E$ $E \rightarrow E.^*E$	$I_9:$ $E \rightarrow (E).$

**Table 3.1.21. Set of LR(0) items for given grammar**

- Parsing table given below shows error detection and recovery.

States	Action							goto
	id	+	*	(	)	\$	E	
0	S3	E1	E1	S2	E2	E1	1	
1	E3	S4	S5	E3	E2	Acc		
2	S3	E1	E1	S2	E2	E1	6	
3	R4	R4	R4	R4	R4	R4		
4	S3	E1	E1	S2	E2	E1	7	
5	S3	E1	E1	S2	E2	E1	8	
6	E3	S4	S5	E3	S9	E4		
7	R1	R1	S5	R1	R1	R1		
8	R2	R2	R2	R2	R2	R2		
9	R3	R3	R3	R3	R3	R3		

**Table 3.1.22. LR parsing table with error routines**

The error routines are as follow:

- E1: push an imaginary id onto the stack and cover it with state 3.  
Issue diagnostics “missing operands”. This routine is called from states 0, 2, 4 and 5, all of which expect the beginning of an operand, either an id or left parenthesis. Instead, an operator + or \*, or the end of the input found.
- E2: remove the right parenthesis from the input. Issue diagnostics “unbalanced right parenthesis”. This routine is called from states 0, 1, 2, 4, 5 on finding right parenthesis.
- E3: push + on to the stack and cover it with state 4  
Issue diagnostics “missing operator”. This routine is called from states 1 or 6 when expecting an operator and an id or right parenthesis is found.
- E4: push right parenthesis onto the stack and cover it with state 9. Issue diagnostics “missing right parenthesis”. This routine is called from states 6 when the end of the input is found. State 6 expects an operator or right parenthesis.

Stack	Input	Error message and action
0	id+)\$	
Oid3	+)\$	
OE1	+)\$	
OE1+4	)\$	
OE1+4	\$	“unbalanced right parenthesis” e2 removes right parenthesis
OE1+4id3	\$	“missing operands” e1 pushes id 3 on stack
OE1+4E7	\$	
OE1	\$	

**Table 3.1.23. Parsing and Error recovery moves made by LR parser**

### 1. Syntax Directed Definitions.

- Syntax directed definition is a generalization of context free grammar in which each grammar symbol has an associated set of attributes.
- Types of attributes are,
  1. Synthesized attribute
  2. Inherited attribute
- Difference between synthesized and inherited attribute,

No	Synthesized Attribute	Inherited attribute
1	Value of synthesized attribute at a node can be computed from the value of attributes at the children of that node in the parse tree.	Values of the inherited attribute at a node can be computed from the value of attribute at the parent and/or siblings of the node.
2	Pass the information from bottom to top in the parse tree.	Pass the information top to bottom in the parse tree or from left siblings to the right siblings

**Table 3.2.1 Difference between Synthesized and Inherited attribute**

### 2. Explain synthesized attributes with example. OR Write a syntax directed definition for desk calculator.

- Value of synthesized attribute at a node can be computed from the value of attributes at the children of that node in the parse tree.
- Syntax directed definition that uses synthesized attribute exclusively is said to be S-attribute definition.
- A parse tree for an S-attributed definition can always be annotated by evaluating the semantic rules for the attribute at the each node bottom up, from the leaves to root.
- An annotated parse tree is a parse tree showing the value of the attributes at each node. The process of computing the attribute values at the node is called annotating or decorating the parse tree.
- The syntax directed definition for simple desk calculator is given in table 3.2.2.

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

**Table 3.2.2 Syntax directed definition of a simple desk calculator**

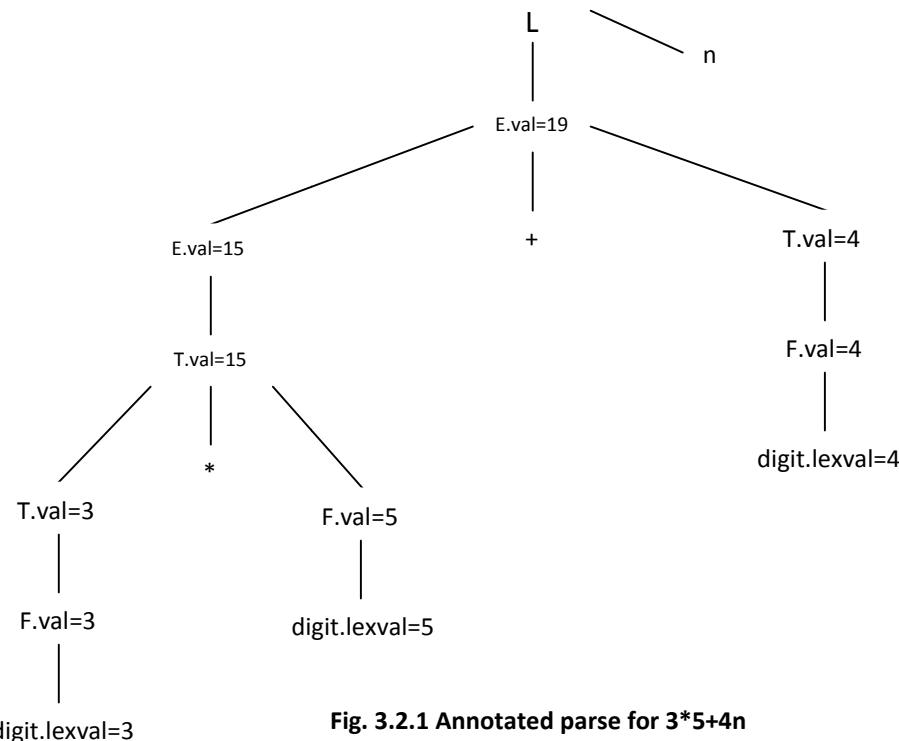


Fig. 3.2.1 Annotated parse for  $3*5+4n$

### 3. Explain Inherited Attribute.

- An inherited value at a node in a parse tree is defined in terms of attributes at the parent and/or siblings of the node.
- Convenient way for expressing the dependency of a programming language construct on the context in which it appears.
- We can use inherited attributes to keep track of whether an identifier appears on the left or right side of an assignment to decide whether the address or value of the assignment is needed.
- The inherited attribute distributes type information to the various identifiers in a declaration.

**Example:**

Production	Semantic Rules
$D \rightarrow T \ L$	$L.in = T.type$
$T \rightarrow int$	$T.type = integer$
$T \rightarrow real$	$T.type = real$
$L \rightarrow L_1, id$	$L_1.in = L.in, addtype(id.entry, L.in)$
$L \rightarrow id$	$addtype(id.entry, L.in)$

Table 3.2.3 Syntax directed definition with inherited attribute

1. Symbol T is associated with a synthesized attribute *type*.
2. Symbol L is associated with an inherited attribute *in*.

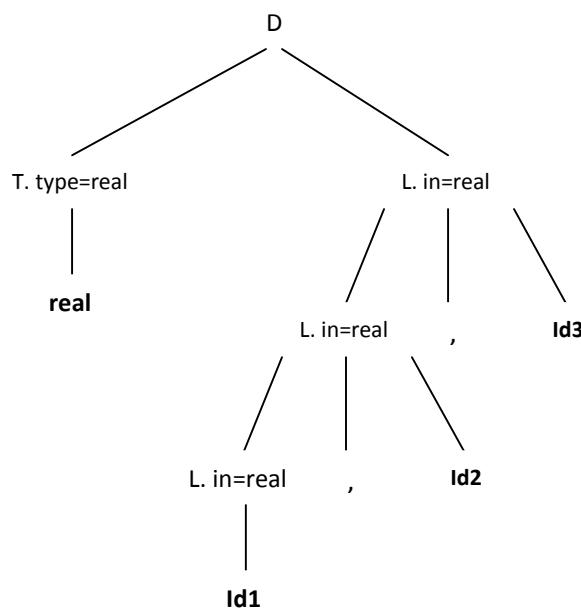


Fig. 3.2.2 Inherited attribute

#### 4. Construct a Syntax-Directed Definition that translates arithmetic expressions from infix to prefix notation.

- The grammar that contains all the syntactic rules along with the semantic rules having synthesized attribute only.
- Such a grammar for converting infix operators to prefix is given by using the 'val' as S-attribute.

Production	Semantic Rules
$L \rightarrow E$	$\text{Print}(E.\text{val})$
$E \rightarrow E+T$	$E.\text{val} = '+' E.\text{val } T.\text{val}$
$E \rightarrow E-T$	$E.\text{val} = '-' E.\text{val } T.\text{val}$
$E \rightarrow T$	$E.\text{val} = T.\text{val}$
$T \rightarrow T^*F$	$T.\text{val} = '*' T.\text{val } F.\text{val}$
$T \rightarrow T/F$	$T.\text{val} = '/' T.\text{val } F.\text{val}$
$T \rightarrow F$	$T.\text{val} = F.\text{val}$
$F \rightarrow F^P$	$F.\text{val} = '^' F.\text{val } P.\text{val}$
$F \rightarrow P$	$F.\text{val} = P.\text{val}$
$P \rightarrow (E)$	$P.\text{val} = E.\text{val}$
$P \rightarrow \text{digit}$	$P.\text{val} = \text{digit.lexval}$

Table 3.2.4 Syntax directed definition for infix to prefix notation

#### 5. Dependency graph.

- The directed graph that represents the interdependencies between synthesized and inherited attribute at nodes in the parse tree is called dependency graph.
- For the rule  $X \rightarrow YZ$  the semantic action is given by  $X.x=f(Y.y, Z.z)$  then synthesized

attribute X.x depends on attributes Y.y and Z.z.

### Algorithm to construct Dependency graph

```

for each node n in the parse tree do
    for each attribute a of the grammar symbol at n do
        Construct a node in the dependency graph for a;
    for each node n in the parse tree do
        for each semantic rule b=f(c1,c2,.....,ck)
            associated with the production used at n do
            for i=1 to k do
                construct an edge from the node for Ci to the node for b;

```

### Example:

$E \rightarrow E_1 + E_2$

$E \rightarrow E_1 * E_2$

Production	Semantic Rules
$E \rightarrow E_1 + E_2$	$E.\text{val} = E_1.\text{val} + E_2.\text{val}$
$E \rightarrow E_1 * E_2$	$E.\text{val} = E_1.\text{val} * E_2.\text{val}$

Table 3.2.5 semantic rules

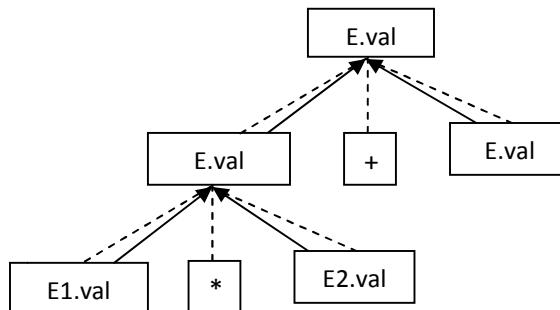


Fig. 3.2.3 Dependency Graph

- The synthesized attributes can be represented by .val.
- Hence the synthesized attributes are given by E.val, E1.val and E2.val. The dependencies among the nodes are given by solid arrow. The arrow from E1 and E2 show that value of E depends on E1 and E2.

## 6. Construction of syntax tree for expressions.

- We use the following function to create the nodes of the syntax tree for expression with binary operator. Each function returns a pointer to a newly created node.
1. Mknod(op,left,right) creates an operator node with label op and two fields containing pointers to left and right.
  2. Mkleaf(id, entry) creates an identifier node with label id and a field containing entry, a pointer to the symbol table entry for the identifier.
  3. Mkleaf(num, val) creates a number node with label num and a field containing val, the value of the number.

**Example:** construct syntax tree for a-4+c

P1:mkleaf(id, entry for a);

```

P2:mkleaf(num, 4);
P3:mknnode('-',p1,p2);
P4:mkleaf(id, entry for c);
P5:mknnode('+',p3,p4);

```

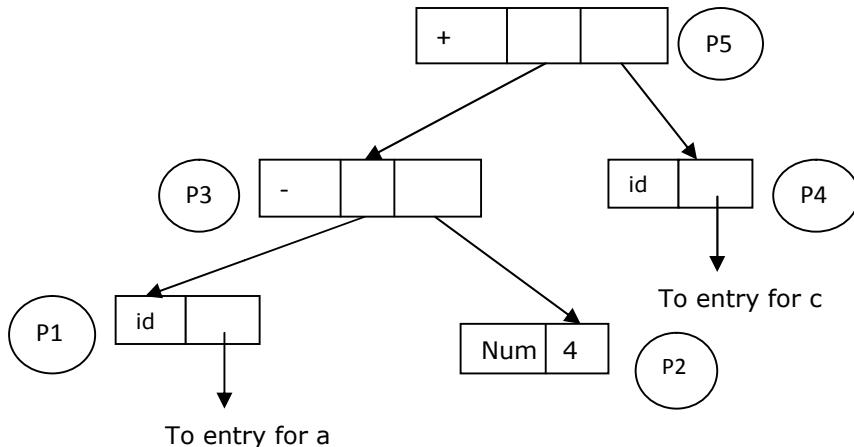


Fig.3.2.4 Syntax tree for a-4+c

## 7. L-Attributed Definitions.

- A syntax directed definition is L-attributed if each inherited attribute of  $X_j$ ,  $1 \leq j \leq n$ , on the right side of  $A \rightarrow X_1, X_2, \dots, X_{j-1}$  depends only on,
  1. The attributes of the symbols  $X_1, X_2, \dots, X_{j-1}$  to the left of  $X_j$  in the production and
  2. The inherited attribute of  $A$ .

Example:

Production	Semantic Rules
$A \rightarrow LM$	$L.i := l(A.i)$ $M.i = m(L.s)$ $A.s = f(M.s)$
$A \rightarrow QR$	$R.i = r(A.i)$ $Q.i = q(R.s)$ $A.s = f(Q.s)$

Table 3.2.6 A non L-attributed syntax directed definition

- Above syntax directed definition is not L-attributed because the inherited attribute  $Q.i$  of the grammar symbol  $Q$  depends on the attribute  $R.s$  of the grammar symbol to its right.

## 8. Syntax directed definitions & Translation schemes.

- Attributes are used to evaluate the expression along the process of parsing.
- During the process of parsing the evaluation of attribute takes place by consulting the semantic action enclosed in { }.
- This process of execution of code fragment semantic actions from the syntax-directed definition can be done by syntax-directed translation scheme.
- A translation scheme generates the output by executing the semantic actions in an

ordered manner.

- This process uses the depth first traversal.

**Example:** Consider grammar,

$$E \rightarrow TP$$

$$T \rightarrow 0|1|2|3|4|5|6|7|8|9$$

$$P \rightarrow +TP \mid \epsilon$$

The translation scheme for grammar,

Production	Semantic Actions
$E \rightarrow TP$	
$T \rightarrow 0$	{Print('0')}
$T \rightarrow 1$	{Print('1')}
$T \rightarrow 2$	{Print('2')}
$T \rightarrow 3$	{Print('3')}
$T \rightarrow 4$	{Print('4')}
$T \rightarrow 5$	{Print('5')}
$T \rightarrow 6$	{Print('6')}
$T \rightarrow 7$	{Print('7')}
$T \rightarrow 8$	{Print('8')}
$T \rightarrow 9$	{Print('9')}
$P \rightarrow +TP \mid \epsilon$	{Print('+')}P   $\epsilon$

Table 3.2.7 Translation scheme

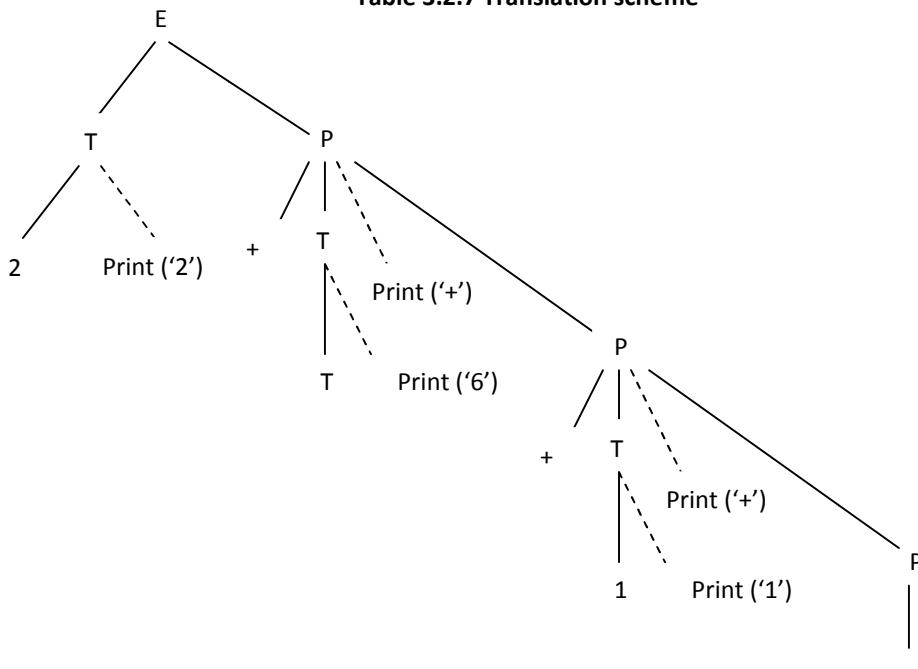


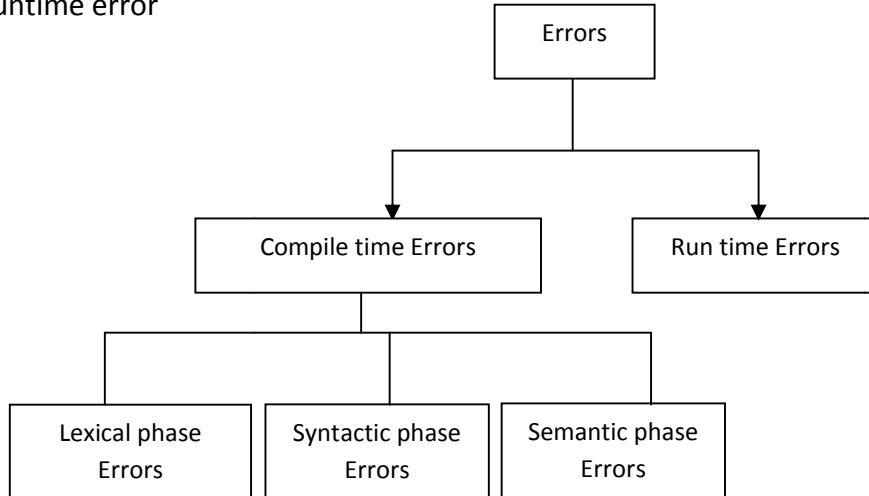
Fig. 3.2.5 Annotated parse tree

- The translation scheme proceeds using depth first traversal.
- The annotated parse shows the semantic actions associated of infix to postfix form.
- The associated semantic actions can be shown using dotted line.

### 1. Basic types of Error.

Error can be classified into mainly two categories,

1. Compile time error
2. Runtime error



**Fig.4.1. Types of Error**

#### Lexical Error

This type of errors can be detected during lexical analysis phase. Typical lexical phase errors are,

1. Spelling errors. Hence get incorrect tokens.
2. Exceeding length of identifier or numeric constants.
3. Appearance of illegal characters.

Example:

```

fi ()
{
}
  
```

- In above code 'fi' cannot be recognized as a misspelling of keyword if rather lexical analyzer will understand that it is an identifier and will return it as valid identifier. Thus misspelling causes errors in token formation.

#### Syntax error

These types of error appear during syntax analysis phase of compiler.

Typical errors are:

1. Errors in structure.
  2. Missing operators.
  3. Unbalanced parenthesis.
- The parser demands for tokens from lexical analyzer and if the tokens do not satisfy the grammatical rules of programming language then the syntactical errors get raised.

#### Semantic error

This type of error detected during semantic analysis phase.

Typical errors are:

1. Incompatible types of operands.

2. Undeclared variable.
3. Not matching of actual argument with formal argument.

## 2. Error recovery strategies. OR Ad-hoc and systematic methods.

### 1. Panic mode

- This strategy is used by most parsing methods. This is simple to implement.
- In this method on discovering error, the parser discards input symbol one at time. This process is continued until one of a designated set of synchronizing tokens is found. Synchronizing tokens are delimiters such as semicolon or end. These tokens indicate an end of input statement.
- Thus in panic mode recovery a considerable amount of input is skipped without checking it for additional errors.
- This method guarantees not to go in infinite loop.
- If there is less number of errors in the same statement then this strategy is best choice.

### 2. Phrase level recovery

- In this method, on discovering an error parser performs local correction on remaining input.
- It can replace a prefix of remaining input by some string. This actually helps parser to continue its job.
- The local correction can be replacing comma by semicolon, deletion of semicolons or inserting missing semicolon. This type of local correction is decided by compiler designer.
- While doing the replacement a care should be taken for not going in an infinite loop.
- This method is used in many error-repairing compilers.

### 3. Error production

- If we have good knowledge of common errors that might be encountered, then we can augment the grammar for the corresponding language with error productions that generate the erroneous constructs.
- If error production is used during parsing, we can generate appropriate error message to indicate the erroneous construct that has been recognized in the input.
- This method is extremely difficult to maintain, because if we change grammar then it becomes necessary to change the corresponding productions.

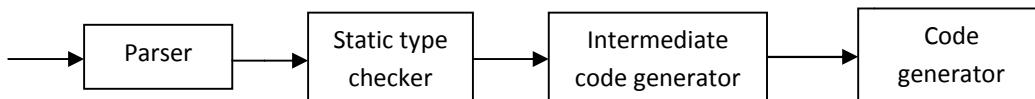
### 4. Global correction

- We often want such a compiler that makes very few changes in processing an incorrect input string.
- Given an incorrect input string  $x$  and grammar  $G$ , the algorithm will find a parse tree for a related string  $y$ , such that number of insertions, deletions and changes of token require to transform  $x$  into  $y$  is as small as possible.
- Such methods increase time and space requirements at parsing time.
- Global production is thus simply a theoretical concept.

### 1. Role of intermediate code generation. OR

**What is intermediate code? Which are the advantages of it?**

- In the analysis-synthesis model of a compiler, the front end translates a source program into an intermediate representation from which backend generates target code.
- The generation of an intermediate language leads to efficient code generation.



**Fig.5.1 Position of intermediate code generator in compiler**

- There are certain advantages of generating machine independent intermediate code,
  1. A compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
  2. A machine independent code optimizer can be applied to intermediate code in order to optimize the code generation.

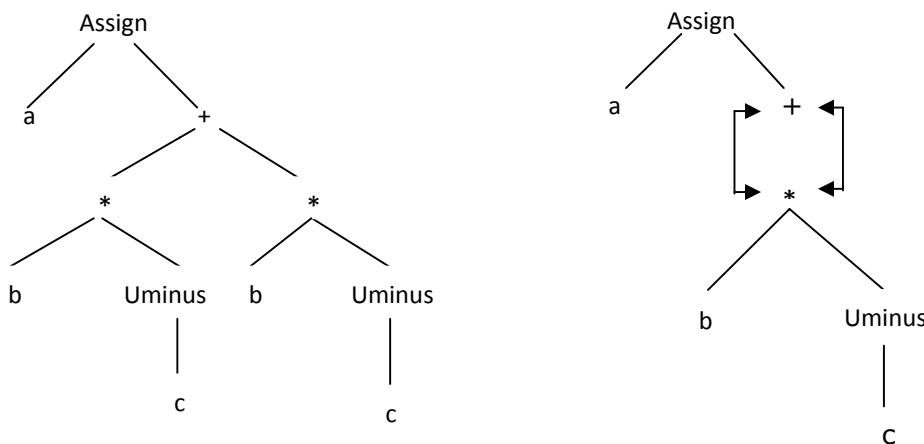
### 2. Explain different intermediate forms.

There are three types of intermediate representation,

1. Abstract syntax tree
2. Postfix notation
3. Three address code

#### Abstract syntax tree

- A syntax tree depicts the natural hierarchical structure of a source program.
- A DAG (Directed Acyclic Graph) gives the same information but in a more compact way because common sub-expressions are identified.
- A syntax tree and DAG for the assignment statement  $a = b^*-c + b^*-c$  is given in Fig. 5.2.



**Fig.5.2 Syntax tree & DAG for  $a = b^*-c + b^*-c$**

### Postfix notation

- Postfix notation is a linearization of a syntax tree.
- In postfix notation the operands occurs first and then operators are arranged.
- the postfix notation for the syntax tree in Fig. 5.2 is,  
 $a\ b\ c\ uminus\ *\ b\ c\ uminus\ *\ +\ assign$ .

### Three address code

- Three address code is a sequence of statements of the general form,  
 $a := b \ op\ c$
- Where a, b or c are the operands that can be names or constants. And op stands for any operator.
- For the expression like  $a = b + c + d$  might be translated into a sequence,
  - $t_1 = b + c$
  - $t_2 = t_1 + d$
  - $a = t_2$
- Here  $t_1$  and  $t_2$  are the temporary names generated by the compiler.
- There are at most three addresses allowed (two for operands and one for result). Hence, this representation is called three-address code.

## 3. Implementations of three address code.

- There are three types of representation used for three address code,
  1. Quadruples
  2. Triples
  3. Indirect triples
- Consider the input statement  $x := -a * b + -a * b$ .
- Three address code for above statement given in table 5.1,

$t_1 = -a$
$t_2 := t_1 * b$
$t_3 = -a$
$t_4 := t_3 * b$
$t_5 := t_2 + t_4$
$x = t_5$

Table 5.1 Three address code

### Quadruple representation

- The quadruple is a structure with at the most four fields such as op, arg1, arg2.
- The op field is used to represent the internal code for operator.
- The arg1 and arg2 represent the two operands.
- And result field is used to store the result of an expression.
- Statement with unary operators like  $x = -y$  do not use arg2.

- Conditional and unconditional jumps put the target label in result.

Number	Op	Arg1	Arg2	result
(0)	uminus	a		t <sub>1</sub>
(1)	*	t <sub>1</sub>	b	t <sub>2</sub>
(2)	uminus	a		t <sub>3</sub>
(3)	*	t <sub>3</sub>	b	t <sub>4</sub>
(4)	+	t <sub>2</sub>	t <sub>4</sub>	t <sub>5</sub>
(5)	:=	t <sub>5</sub>		x

Table 5.2 Quadruple representation

### Triples

- To avoid entering temporary names into the symbol table, we might refer a temporary value by the position of the statement that computes it.
- If we do so, three address statements can be represented by records with only three fields: op, arg1 and arg2.

Number	Op	Arg1	Arg2
(0)	uminus	a	
(1)	*	(0)	b
(2)	uminus	a	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	:=	x	(4)

Table 5.3 Triple representation

### Indirect Triples

- In the indirect triple representation the listing of triples has been done. And listing pointers are used instead of using statement.
- This implementation is called indirect triples.

Number	Op	Arg1	Arg2		Statement
(0)	uminus	a		(0)	(11)
(1)	*	(11)	b	(1)	(12)
(2)	uminus	a		(2)	(13)

(3)	*	(13)	b	(4)	(15)
(4)	+	(12)	(14)	(5)	(16)
(5)	:=	X	(15)		

Table 5.4 Indirect triple representation

### 4. Syntax directed translation mechanism.

- For obtaining the three address code the SDD translation scheme or semantic rules must be written for each source code statement.
- There are various programming constructs for which the semantic rules can be defined.
- Using these rules the corresponding intermediate code in the form of three address code can be generated.
- Various programming constructs are,
  1. Declarative statement
  2. Assignment statement
  3. Arrays
  4. Boolean expressions
  5. Control statement
  6. Switch case
  7. Procedure call

#### Declarative Statement

- In the declarative statements the data items along with their data types are declared.
- Example:

S → D	{offset=0}
D → id: T	{enter(id.name, T.type, offset); offset=offset+T.width}
T → integer	{T.type:=integer; T.width:=4}
T → real	{T.type:=real; T.width:=8}
T → array[num] of T <sub>1</sub>	{T.type:=array(num.val,T <sub>1</sub> .type) T.width:=num.val X T <sub>1</sub> .width }
T → *T <sub>1</sub>	{T.type:=pointer(T.type) T.width:=4}

Table 5.5 Syntax directed translation for Declarative statement

- Initially, the value of offset is set to zero. The computation of offset can be done by using the formula offset = offset + width.
- In the above translation scheme T.type and T.width are the synthesized attributes.
- The type indicates the data type of corresponding identifier and width is used to indicate the memory units associated with an identifier of corresponding type.
- The rule D → id: T is a declarative statement for id declaration.

- The enter function used for creating the symbol table entry for identifier along with its type and offset.
- The width of array is obtained by multiplying the width of each element by number of elements in the array.

### Assignment statement

- The assignment statement mainly deals with the expressions.
- The expressions can be of type integer, real, array and record.
- Consider the following grammar,

$$\begin{aligned} S \rightarrow & id := E \\ E \rightarrow & E_1 + E_2 \\ E \rightarrow & E_1 * E_2 \\ E \rightarrow & -E_1 \\ E \rightarrow & (E_1) \\ E \rightarrow & id \end{aligned}$$

The translation scheme of above grammar is given in table 5.6.

Production Rule	Semantic actions
$S \rightarrow id := E$	{ p=look_up(id.name); If p≠ nil then emit(p = E.place) else error; }
$E \rightarrow E_1 + E_2$	{ E.place=newtemp(); emit (E.place=E <sub>1</sub> .place '+' E <sub>2</sub> .place)}
$E \rightarrow E_1 * E_2$	{ E.place=newtemp(); emit (E.place=E <sub>1</sub> .place '*' E <sub>2</sub> .place)}
$E \rightarrow -E_1$	{ E.place=newtemp(); emit (E.place='uminus' E <sub>1</sub> .place)}
$E \rightarrow (E_1)$	{ E.place=E <sub>1</sub> .place }
$E \rightarrow id$	{ p=look_up(id.name); If p≠ nil then emit (p = E.place) else error; }

Table 5.6. Translation scheme to produce three address code for assignments

- The look\_up returns the entry for id.name in the symbol table if it exists there.
- The function emit is for appending the three address code to the output file. Otherwise an error will be reported.
- newtemp() is the function for generating new temporary variables.
- E.place is used to hold the value of E.
- Consider the assignment statement  $x:=(a+b) *(c+d)$ ,

Production Rule	Semantic action for Attribute evaluation	Output

E → id	E.place := a	
E → id	E.place := b	
E → E <sub>1</sub> + E <sub>2</sub>	E.place := t <sub>1</sub>	t <sub>1</sub> := a+b
E → id	E.place := c	
E → id	E.place := d	
E → E <sub>1</sub> + E <sub>2</sub>	E.place := t <sub>2</sub>	t <sub>2</sub> := c+d
E → E <sub>1</sub> * E <sub>2</sub>	E.place := t <sub>3</sub>	t <sub>3</sub> := (a+b)*(c+d)
S → id := E		x := t <sub>3</sub>

**Table 5.7 Three address code for Assignment statement**

### Arrays

- Array is a contiguous storage of elements.
- Elements of an array can be accessed quickly if the elements are stored in a block of consecutive locations. If the width of each array element is w, then the i<sup>th</sup> element of array A begins in location,  

$$\text{base} + (i - \text{low}) \times w$$
- Where low is the lower bound on the subscript and base is the relative address of the storage allocated for the array. That is, base is the relative address of A[low].
- The expression can be partially evaluated at compile time if it is rewritten as,  

$$i \times w + (\text{base} - \text{low} \times w)$$
- The sub expression c = base - low x w can be evaluated when the declaration of the array is seen. We assume that c is saved in the symbol table entry for A, so the relative address of A[i] is obtained by simply adding i x w to c.
- There are two representation of array,
  1. Row major representation.
  2. Column major representation.
- In the case of row-major form, the relative address of A[ i<sub>1</sub>, i<sub>2</sub>] can be calculated by the formula:  

$$\text{base} + ((i_1 - \text{low}_1) \times n_2 + i_2 - \text{low}_2) \times w$$
- where, low<sub>1</sub> and low<sub>2</sub> are the lower bounds on the values of i<sub>1</sub> and i<sub>2</sub> and n<sub>2</sub> is the number of values that i<sub>2</sub> can take. That is, if high<sub>2</sub> is the upper bound on the value of i<sub>2</sub>, then n<sub>2</sub> = high<sub>2</sub> - low<sub>2</sub> + 1.
- Assuming that i<sub>1</sub> and i<sub>2</sub> are the only values that are known at compile time, we can rewrite the above expression as  

$$((i_1 \times n_2) + i_2) \times w + (\text{base} - ((\text{low}_1 \times n_2) + \text{low}_2) \times w)$$
- Generalized formula: The expression generalizes to the following expression for the relative address of A[i<sub>1</sub>,i<sub>2</sub>,...,i<sub>k</sub>]  

$$((\dots((i_1 n_2 + i_2) n_3 + i_3) \dots) n_k + i_k) \times w + \text{base} - ((\dots((\text{low}_1 n_2 + \text{low}_2) n_3 +$$

low3) . . . )nk + lowk) x w

for all j, nj = highj – lowj + 1

- The Translation Scheme for Addressing Array Elements :

S → L := E

E → E + E

E → ( E )

E → L

L → Elist ]

L → id

Elist → Elist , E

Elist → id [ E

- The translation scheme for generating three address code is given by using appropriate semantic actions.

Production Rule	Semantic Rule
S → L := E	{ if L.offset = null then emit ( L.place ':=' E.place ); else emit ( L.place '[' L.offset ']' ':=' E.place ) }
E → E + E	{ E.place := newtemp; emit ( E.place ':=' E1.place '+' E2.place ) }
E → ( E )	{ E.place := E1.place }
E → L	{ if L.offset = null then E.place := L.place else begin E.place := newtemp; emit ( E.place ':=' L.place '[' L.offset ']' ) end }
L → Elist ]	{ L.place := newtemp; L.offset := newtemp; emit ( L.place ':=' c( Elist.array )); emit ( L.offset ':=' Elist.place '*' width ( Elist.array )) }
L → id	{ L.place := id.place; L.offset := null }
Elist → Elist , E	{ t := newtemp; dim := Elist1.ndim + 1; emit ( t ':=' Elist1.place '*' limit ( Elist1.array, dim )); emit ( t ':=' t '+' E.place ); Elist.array := Elist1.array; Elist.place := t; Elist.ndim := dim }
Elist → id [ E	{ Elist.array := id.place;

	Elist.place := E.place; Elist.ndim := 1 }
--	--

Table 5.8 Syntax directed translation scheme to generate three address code for Array

- Annotated parse tree for  $x=A[i, j]$  is given in figure 5.3.

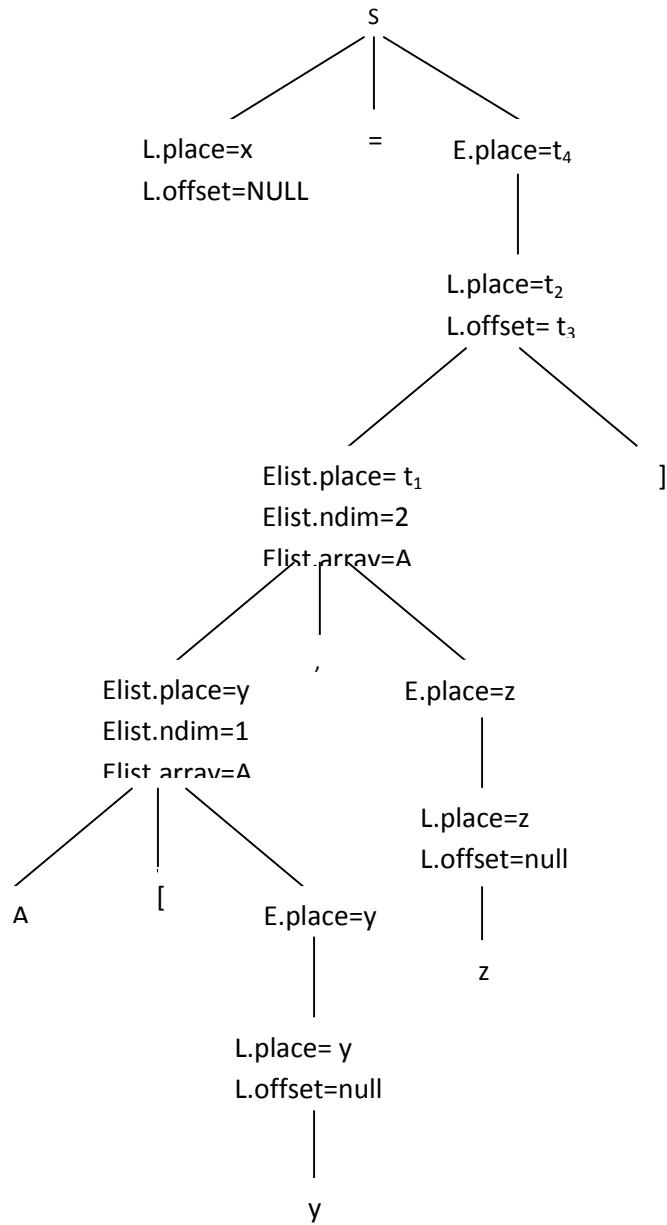


Fig 5.3. Annotated parse tree for  $x:=A[y, z]$

### Boolean expressions

- Normally there are two types of Boolean expressions used,
  - For computing the logical values.
  - In conditional expressions using if-then-else or while-do.
- Consider the Boolean expression generated by following grammar :  

$$E \rightarrow E \text{ OR } E$$

$E \rightarrow E \text{ AND } E$   
 $E \rightarrow \text{NOT } E$   
 $E \rightarrow (E)$   
 $E \rightarrow \text{id relop id}$   
 $E \rightarrow \text{TRUE}$   
 $E \rightarrow \text{FALSE}$

- The relop is denoted by  $\leq, \geq, <, >$ . The OR and AND are left associate.
- The highest precedence is to NOT then AND and lastly OR.

$E \rightarrow E_1 \text{ OR } E_2$	{E.place:=newtemp() Emit (E.place ':=E1.place "OR' E2.place)}
$E \rightarrow E_1 \text{ AND } E_2$	{E.place:=newtemp() Emit (E.place ':=E1.place "AND' E2.place)}
$E \rightarrow \text{NOT } E_1$	{E.place:=newtemp() Emit (E.place ':="NOT' E1.place)}
$E \rightarrow (E_1)$	{E.place := E1.place }
$E \rightarrow \text{id}_1 \text{ relop } \text{id}_2$	{E.place := newtemp() Emit ('if id.place relop.op id2.place 'goto' next_state +3); Emit (E.place':='0'); Emit ('goto' next state +2); Emit (E.place := '1')}
$E \rightarrow \text{TRUE}$	{E.place:=newtemp() Emit (E.place ':=' '1')}
$E \rightarrow \text{FALSE}$	{E.place:=newtemp() Emit (E.place ':=' '0')}

**Table 5.9 Syntax directed translation scheme to generate three address code for Boolean expression**

- The function Emit generates the three address code and newtemp () is for generation of temporary variables.
- For the semantic action for the rule  $E \rightarrow \text{id}_1 \text{ relop } \text{id}_2$  contains next\_state which gives the index of next three address statement in the output sequence.
- Let us take an example and generate the three address code using above translation scheme:

$p > q \text{ AND } r < s \text{ OR } u > v$   
100: if p > q goto 103  
101: t1:=0  
102: goto 104  
103: t1:=1  
104: if r < s goto 107  
105: t2:=0  
106: goto 108  
107: t2=1  
108:if u>v goto 111

```

109:t3=0
110:goto 112
111:t3=1
112:t4=t1 AND t2
113:t5=t4 OR t3

```

### Control statement

- The control statements are if-then-else and while-do.
- The grammar and translation scheme for such statements is given in table 5.10.
- $S \rightarrow \text{if } E \text{ then } S_1 \mid \text{if } E \text{ then } S_1 \text{ else } S_2 \mid \text{while } E \text{ do } S_1$

S->if E then S1	{E.true=new_label() E.False=new_label() S1.next=S.next S2.next=S.next S.code=E.code  gen_code(E.true':')  S1.code}
S->if E then S1 else S2	{E.true=new_label() E.False=new_label() S1.next=S.next S2.next=S.next S.code=E.code  gen_code(E.true':')  S1.code   gen_code('goto',S.next)   gen_code(E.false':')  S2.code}
S->while E do S1	{S.begin=new_label() E.True=new_label() E.False=S.next S1.next=S.begin S.code=gen_code(S.begin':')  E.code   gen_code(E.true':')   S1.code  gen_code('goto',S.begin)}

Table 5.10 Syntax directed translation scheme to generate three address code for Control statement

- Consider the statement: if  $a < b$  then  $a = a + 5$  else  $a = a + 7$
- Three address code for above statement using semantic rule is,  
100: if  $a < b$  goto L1  
101: goto 103  
102: L1:  $a = a + 5$   
103:  $a = a + 7$

### Switch case

- Consider the following switch statement;  
**switch** E

```

begin
  case V1: S1
  case V2: S2
  ....
  case Vn-1: Sn-1
  default: Sn
end

```

- Syntax directed translation scheme to translate this case statement into intermediate code is given in table 5.11.

	Code to evaluate E into t
	goto test
L <sub>1</sub> :	Code for S <sub>1</sub>
	goto next
L <sub>2</sub>	Code for S <sub>2</sub>
	goto next
	.....
L <sub>n-1</sub>	Code for S <sub>n-1</sub>
	goto next
L <sub>n</sub>	Code for S <sub>n</sub>
	goto next
test:	If t=V <sub>1</sub> goto L <sub>1</sub>
	If t=V <sub>1</sub> goto L <sub>1</sub>
	If t=V <sub>1</sub> goto L <sub>1</sub>
	goto L <sub>n</sub>
next:	

Table 5.11 Syntax directed translation scheme to generate three address code for switch case

- When we see the keyword switch, we generate two new labels test and next and a new temporary t.
- After processing E, we generate the jump goto test.
- We process each statement case V<sub>i</sub> : S<sub>i</sub> by emitting the newly created label L<sub>i</sub>, followed by code for S<sub>i</sub>, followed by the jump goto next.
- When the keyword end terminating the body of switch is found, we are ready to generate the code for n-way branch.
- Reading the pointer value pairs on the case stack from the bottom to top , we can generate a sequence of three address code of the form,

```

case V1 L1
case V2 L2
.....
case Vn-1 Ln-1
case t Ln
label next

```

- Where t is the name holding the value of selector expression E, and L<sub>n</sub> is the label for default statement.
- The case V<sub>i</sub> L<sub>i</sub> three address statement is a synonym for if t= V<sub>i</sub> goto L<sub>i</sub>.

### Procedure call

- Procedure or function is an important programming construct which is used to obtain the modularity in the user program.
- Consider a grammar for a simple procedure call,  
 $S \rightarrow \text{call id (L)}$   
 $L \rightarrow L, E$   
 $L \rightarrow E$
- Here S denotes the statement and L denotes the list of parameters.
- And E denotes the expression.
- The translation scheme can be as given below,

Production rule	Semantic Action
$S \rightarrow \text{call id (L)}$	{ for each item p in queue do append('param' p); append('call' id.place);}
$L \rightarrow L, E$	{ insert E.place in the queue }
$L \rightarrow E$	{ initialize the queue and insert E.place in the queue }

Table 5.12 Syntax directed translation scheme to generate three address code for procedure call

- The data structure queue is used to hold the various parameters of the procedure.
- The keyword param is used to denote list of parameters passed to the procedure.
- The call to the procedure is given by 'call id' where id denotes the name of procedure.
- E.place gives the value of parameter which is inserted in the queue.
- For L → E the queue gets empty and a single pointer to the symbol table is obtained. This pointer denotes the value of E.

### 1. Source language issue.

#### 1. Procedure call

- A procedure definition is a declaration that associates an identifier with a statement.
- The identifier is the procedure name and the statement is the procedure body.
- For example, the following is the definition of procedure named readarray:

```

Procedure readarray
Var i: integer;
Begin
    For i=1 to 9 do real(a[i])
End;
```

- When a procedure name appears within an executable statement, the procedure is said to be called at that point.

#### 2. Activation tree

- An activation tree is used to depict the way control enters and leaves activations. In an activation tree,
  - Each node represents an activation of a procedure.
  - The root represents the activation of the main program.
  - The node for a is the parent of the node b if and only if control flows from activation a to b.
  - The node for a is to the left of the node for b if and only if the lifetime of a occurs before the lifetime of b.

#### 3. Control stack

- A control stack is used to keep track of live procedure activations.
- The idea is to push the node for activation onto the control stack as the activation begins and to pop the node when the activation ends.
- The contents of the control stack are related to paths to the root of the activation tree.
- When node n is at the top of the stack, the stack contains the nodes along the path from n to the root.

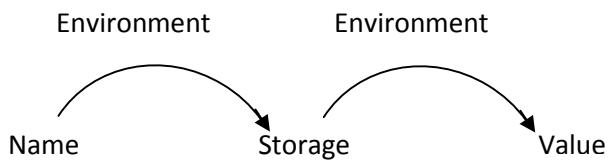
#### 4. The scope of declaration

- A declaration is a syntactic construct that associates information with a name.
- Declaration may be explicit, such as:  

```
var i: integer;
```
- Or they may be implicit. Example, any variable name starting with i is assumed to denote an integer.
- The portion of the program to which a declaration applies is called the scope of that declaration.

#### 5. Bindings of names

- Even if each time name is declared once in a program, the same name may denote different data objects at run time.
- “Data object” corresponds to a storage location that holds values.
- The term environment refers to a function that maps a name to a storage location.
- The term state refers to a function that maps a storage location to the value held there.



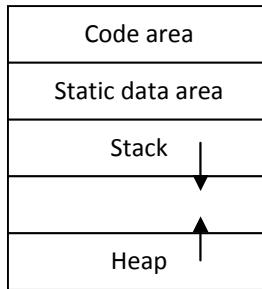
**Fig 6.1 Two stage mapping from name to value**

- When an environment associates storage location s with a name x, we say that x is bound to s.
- This association is referred as a binding of x.

## 2. Storage organization.

### 1. Subdivision of Run-Time memory

- The compiler demands for a block of memory to operating system. The compiler utilizes this block of memory executing the compiled program. This block of memory is called **run time storage**.
- The run time storage is subdivided to hold code and data such as, the generated target code and Data objects.
- The size of generated code is fixed. Hence the target code occupies the determined area of the memory. Compiler places the target code at end of the memory.
- The amount of memory required by the data objects is known at the compiled time and hence data objects also can be placed at the statically determined area of the memory.



**Fig 6.2 Typical subdivision of run time memory into code and data areas**

- Stack is used to manage the active procedure. Managing of active procedures means when a call occurs then execution of activation is interrupted and information about status of the stack is saved on the stack. When the control returns from the call this suspended activation resumed after storing the values of relevant registers.
- Heap area is the area of run time storage in which the other information is stored. For example memory for some data items is allocated under the program control. Memory required for these data items is obtained from this heap area. Memory for some activation is also allocated from heap area.

### 2. Activation Records (Most IMP)

Various fields of activation record are as follows:

1. Temporary values: The temporary variables are needed during the evaluation of expressions. Such variables are stored in the temporary field of activation record.
2. Local variables: The local data is a data that is local to the execution procedure is stored in this field of activation record.

Return value
Actual parameter
Control link
Access link
Saved M/c status
Local variables
Temporaries

**Fig 6.3 Activation Record**

3. Saved machine registers: This field holds the information regarding the status of machine just before the procedure is called. This field contains the registers and program counter.
4. Control link: This field is optional. It points to the activation record of the calling procedure. This link is also called dynamic link.
5. Access link: This field is also optional. It refers to the non local data in other activation record. This field is also called static link field.
6. Actual parameters: This field holds the information about the actual parameters. These actual parameters are passed to the called procedure.
7. Return values: This field is used to store the result of a function call.

### 3. Compile time layout of local data

- Suppose run-time storage comes in block of contiguous bytes, where byte is the smallest unit of addressable memory.
- The amount of storage needed for a name is determined from its type.
- Storage for an aggregate, such as an array or record, must be large enough to hold all its components.
- The field of local data is laid out as the declarations in a procedure are examined at compile time.
- Variable length data has been kept outside this field.
- We keep a count of the memory locations that have been allocated for previous declarations.
- From the count we determine a relative address of the storage for a local with respect to some position such as the beginning of the activation record.
- The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.

### 3. Difference between Static v/s Dynamic memory allocation

No.	Static Memory Allocation	Dynamic Memory Allocation
1	Memory is allocated before the execution of the program begins.	Memory is allocated during the execution of the program.
2	No memory allocation or de-allocation actions are performed during	Memory Bindings are established and destroyed during the execution.

	execution.	
3	Variables remain permanently allocated.	Allocated only when program unit is active.
4	Implemented using stacks and heaps.	Implemented using data segments.
5	Pointer is needed to accessing variables.	No need of dynamically allocated pointers.
6	Faster execution than dynamic.	Slower execution than static.
7	More memory space required.	Less memory space required.

**Table 6.1 Difference between Static and Dynamic memory allocation**

## 4. Storage allocation strategies.

The different storage allocation strategies are;

1. Static allocation: lays out storage for all data objects at compile time.
2. Stack allocation: manages the run-time storage as a stack.
3. Heap allocation: allocates and de-allocates storage as needed at run time from a data area known as heap.

### Static allocation

- In static allocation, names are bound to storage as the program is compiled, so there is no need for a run-time support package.
- Since the bindings do not change at run-time, every time a procedure is activated, its names are bounded to the same storage location.
- Therefore values of local names are retained across activations of a procedure. That is, when control returns to a procedure the value of the local are the same as they were when control left the last time.

### Stack allocation

- All compilers for languages that use procedures, functions or methods as units of user define actions manage at least part of their run-time memory as a stack.
- Each time a procedure is called, space for its local variables is pushed onto a stack, and when the procedure terminates, the space is popped off the stack.

### Calling Sequences: (How is task divided between calling & called program for stack updating?)

- Procedures called are implemented in what is called as calling sequence, which consist of code that allocates an activation record on the stack and enters information into its fields.
- A return sequence is similar to code to restore the state of machine so the calling procedure can continue its execution after the call.
- The code is calling sequence of often divided between the calling procedure (caller) and procedure is calls (callee).
- When designing calling sequences and the layout of activation record, the following principles are helpful:
  1. Value communicated between caller and callee are generally placed at the

beginning of the callee's activation record, so they are as close as possible to the caller's activation record.

2. Fixed length items are generally placed in the middle. Such items typically include the control link, the access link, and the machine status field.
  3. Items whose size may not be known early enough are placed at the end of the activation record.
  4. We must locate the top of the stack pointer judiciously. A common approach is to have it point to the end of fixed length fields in the activation is to have it point to the end of fixed length fields in the activation record. Fixed length data can then be accessed by fixed offsets, known to the intermediate code generator, relative to the top of the stack pointer.
- The calling sequence and its division between caller and callee are as follows:
    1. The caller evaluates the actual parameters.
    2. The caller stores a return address and the old value of *top\_sp* into the callee's activation record. The caller then increments the *top\_sp* to the respective positions.
    3. The callee saves the register values and other status information.
    4. The callee initializes its local data and begins execution.

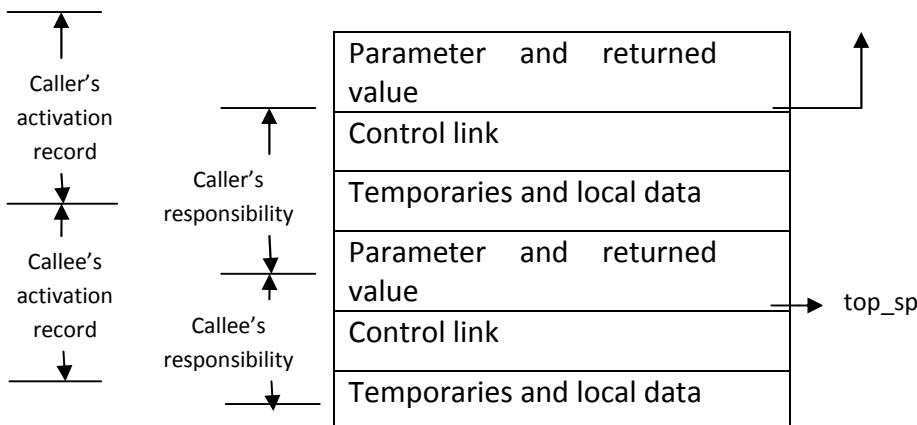


Fig. 6.4 Division of task between caller and callee

- A suitable, corresponding return sequence is:
  1. The callee places the return value next to the parameters.
  2. Using the information in the machine status field, the callee restores *top\_sp* and other registers, and then branches to the return address that the caller placed in the status field.
  3. Although *top\_sp* has been decremented, the caller knows where the return value is, relative to the current value of *top\_sp*; the caller therefore may use that value.

### Variable length data on stack

- The run time memory management system must deal frequently with the allocation of objects, the sizes of which are not known at the compile time, but which are local to a procedure and thus may be allocated on the stack.

- The same scheme works for objects of any type if they are local to the procedure called have a size that depends on the parameter of the call.

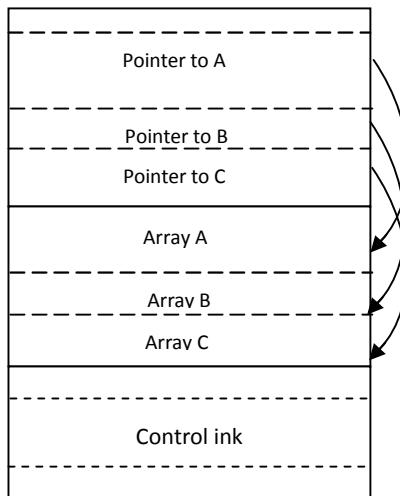


Fig 6.5 Access to dynamically allocated arrays

### Dangling Reference

- Whenever storage can be allocated, the problem of dangling reference arises. The dangling reference occurs when there is a reference of storage that has been allocated.
- It is a logical error to use dangling reference, since, the value of de-allocated storage is undefined according to the semantics of most languages.
- Whenever storage can be allocated, the problem of dangling reference arises. The dangling reference occurs when there is a reference of storage that has been allocated.

### Heap allocation

- Stack allocation strategy cannot be used if either of the following is possible:
  - The value of local names must be retained when activation ends.
  - A called activation outlives the caller.
- Heap allocation parcels out pieces of contiguous storage, as needed for activation record or other objects.
- Pieces may be de-allocated in any order, so over the time the heap will consist of alternate areas that are free and in use.
- The record for an activation of procedure r is retained when the activation ends.
- Therefore, the record for new activation q(1, 9) cannot follow that for s physically.
- If the retained activation record for r is de-allocated, there will be free space in the heap between the activation records for s and q.

Position in the activation tree	Activation records in the heap	Remarks									
 r      q(1,9)	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>S</td></tr> <tr><td>Control link</td></tr> <tr><td> </td></tr> <tr><td>r</td></tr> <tr><td>Control link</td></tr> <tr><td> </td></tr> <tr><td>q(1,9)</td></tr> <tr><td>Control link</td></tr> <tr><td> </td></tr> </table>	S	Control link		r	Control link		q(1,9)	Control link		Retained activation record for r
S											
Control link											
r											
Control link											
q(1,9)											
Control link											

Fig 6.6 Records for live activations need not be adjacent in a heap

## 5. Parameter passing methods.

- There are two types of parameters, Formal parameters & Actual parameters.
- And based on these parameters there are various parameter passing methods, the common methods are,

### 1. Call by value:

- This is the simplest method of parameter passing.
- The actual parameters are evaluated and their r-values are passed to caller procedure.
- The operations on formal parameters do not change the values of a parameter.
- Example: Languages like C, C++ use actual parameter passing method.

### 2. Call by reference :

- This method is also called as call by address or call by location.
- The L-value, the address of actual parameter is passed to the called routines activation record.

<u>Call by value</u>	<u>Call by reference</u>
<pre>void main() {     int x, y;     printf("Enter the value of X &amp; Y:");     scanf("%d%d", &amp;x, &amp;y);     swap(x, y);     printf("\n Values inside the main function");     printf("\n x=%d, y=%d", x, y);     getch(); }  void swap(int x,int y) {</pre>	<pre>void swap(int *, int *); void main() {     int x,y;     printf("Enter the value of X &amp; Y:");     scanf("%d%d", &amp;x, &amp;y);     swap(&amp;x, &amp;y);     printf("\n Value inside the main function");     printf("\n x=%d y=%d", x, y); }  void swap(int *x, int *y) {     int temp;</pre>

<pre> int temp; temp=x; x=y; y=temp; printf("\n Values inside the swap function"); printf("\n x=%d y=%d", x, y); } </pre>	<pre> temp=*x; *x=*y; *y=temp; printf("\n Value inside the swap function"); printf("\n x=%d y=%d", x, y); } </pre>
---	--

**Table 6.2 Code for call by value and call by reference**

### 3. Copy restore:

- This method is a hybrid between call by value and call by reference. This method is also known as copy-in-copy-out or values result.
- The calling procedure calculates the value of actual parameter and it then copied to activation record for the called procedure.
- During execution of called procedure, the actual parameters value is not affected.
- If the actual parameter has L-value then at return the value of formal parameter is copied to actual parameter.

### 4. Call by name:

- This is less popular method of parameter passing.
- Procedure is treated like macro. The procedure body is substituted for call in caller with actual parameters substituted for formals.
- The actual parameters can be surrounded by parenthesis to preserve their integrity.
- The local names of called procedure and names of calling procedure are distinct.

## 6. Block Structure and Non Block Structure Storage Allocation

- Storage allocation can be done for two types of data variables.
  1. Local data
  2. Non local data
- The local data can be handled using activation record whereas non local data can be handled using scope information.
- The block structured storage allocation can done using static scope or lexical scope and the non block structured storage allocation done using dynamic scope.

### 1. Local Data

- The local data can be accessed with the help of activation record.
- The offset relative to base pointer of an activation record points to local data variables within a record, Hence
- Reference to any variable  $x$  in procedure = Base pointer pointing to start of procedure + Offset of variable  $x$  from base pointer.

### 2. Access to non local names

- A procedure may sometimes refer to variables which are not local to it. Such variables are called as non local variables. For the non local names there are two types of rules that can be defined: static and dynamic.

### Static scope rule

- The static scope rule is also called as lexical scope. In this type the scope is determined by examining the program text. PASCAL, C and ADA are the languages use the static scope rule. These languages are also called as block structured language.

### Dynamic scope rule

- For non block structured languages this dynamic scope allocation rules are used.
- The dynamic scope rule determines the scope of declaration of the names at run time by considering the current activation.
- LISP and SNOBOL are the languages which use dynamic scope rule.

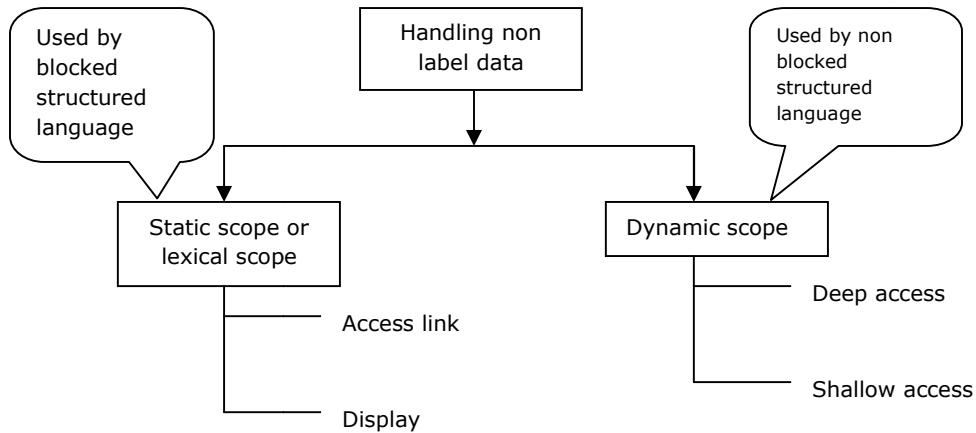


Fig 6.7 Access to non local data

## 7. What is symbol table? How characters of a name (identifiers) are stored in symbol table?

- Definition: Symbol table is a data structure used by compiler to keep track of semantics of a variable. That means symbol table stores the information about scope and binding information about names.
- Symbol table is built in lexical and syntax analysis phases.

### Symbol table entries

- The items to be stored into symbol table are:
  - Variable names
  - Constants
  - Procedure names
  - Function names
  - Literal constants and strings
  - Compiler generated temporaries
  - Labels in source language
- Compiler use following types of information from symbol table:
  - Data type
  - Name
  - Declaring procedure
  - Offset in storage

- 5) If structure or record then pointer to structure table
- 6) For parameters, whether parameter passing is by value or reference?
- 7) Number and type of arguments passed to the function
- 8) Base address

### How to store names in symbol table? (IMP)

There are two types of representation:

#### 1. Fixed length name

- 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.

Name										Attribute	
c	a	l	c	u	l	a	t	e			
s	u	m									
a											
b											

Fig. 6.8 Fixed length name

#### 2. Variable length name

- The amount of space required by string is used to store the names. The name can be stored with the help of starting index and length of each name.

Name		Attribute	
Starting index	Length		
0	10		
10	4		
14	2		
16	2		

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
C	a	l	c	u	l	a	t	e	\$	s	u	m	\$	a	\$	b	\$

Fig. 6.9 Variable length name

## 8. Explain data structures for a symbol table.

### 1. List Data structure

- The amount of space required by string is used to store the names. The name can be stored with the help of starting index and length of each name.
- Linear list is a simplest kind of mechanism to implement the symbol table.
- In this method an array is used to store names and associated information.
- New names can be added in the order as they arrive.
- The pointer 'available' is maintained at the end of all stored records. The list data structure

using array is given below:

Name 1	Info 1
Name 2	Info 2
Name 3	Info 3
Name n	Info n

Fig. 6.10 List data structure

- 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 should not be already there. If it is there another error occurs i.e. "Multiple defined Name".
- The advantage of list organization is that it takes minimum amount of space.

### 2. Self organizing list

- 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".
- This symbol table implementation is 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.

First →

Name 1	Info 1	
Name 2	Info 2	
Name 3	Info 3	
Name 4	Info 4	

Fig. 6.11 Self organizing list

- A pointer "First" is maintained to point to first record of the symbol table.
- The reference to these names can be Name 3, Name 1, Name 4, and Name 2.
- When the name is referenced or created it is moved to the front of the list.
- The most frequently referred names will tend to be front of the list. Hence time to most referred names will be the least.

### 3. Binary tree

- The most frequently referred names will tend to be front of the list. Hence time to most referred names will be the least.
- When the organization symbol table is by means of binary tree, the node structure will as follows:
- The left child field stores the address of previous symbol.
- Right child field stores the address of next symbol. The symbol field is used to store the

name of the symbols.

- Information field is used to give information about the symbol.
- Binary tree structure is basically a binary search tree in which the value of left is always less than the value of parent node. Similarly the value of right node is always more or greater than the parent node.

#### 4. Hash table

- Hashing is an important technique used to search the records of symbol table. This method is superior to list organization.
- In hashing scheme two tables are maintained-a hash table and symbol table.
- The hash table consists of k entries from 0,1 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  $h(\text{name})$  will result any integer between 0 to k-1. We can search any name by position= $h(\text{name})$ .
- Using this position we can obtain the exact locations of name in symbol table.
- Hash function should result in uniform distribution of names in symbol.
- Hash function should be such that there will be minimum number of collision. Collision is such situation where hash function results in same location for storing the names.
- Collision resolution techniques are open addressing, chaining, rehashing.
- Advantage of hashing is quick search is possible and the disadvantage is that hashing is complicated to implement. Some extra space is required. Obtaining scope of variables is very difficult to implement.

## 9. Dynamic Storage Allocation Techniques

There are two techniques used in dynamic memory allocation and those are -

- Explicit allocation
- Implicit allocation

#### 1. Explicit Allocation

- The explicit allocation can be done for fixed size and variable sized blocks.

##### **Explicit Allocation for Fixed Size Blocks**

- This is the simplest technique of explicit allocation in which the size of the block for which memory is allocated is fixed.
- In this technique a free list is used. Free list is a set of free blocks. This observed when we want to allocate memory. If some memory is de-allocated then the free list gets appended.
- The blocks are linked to each other in a list structure. The memory allocation can be done by pointing previous node to the newly allocated block. Memory de-allocation can be done by de-referencing the previous link.
- The pointer which points to first block of memory is called Available.
- This memory allocation and de-allocation is done using heap memory.

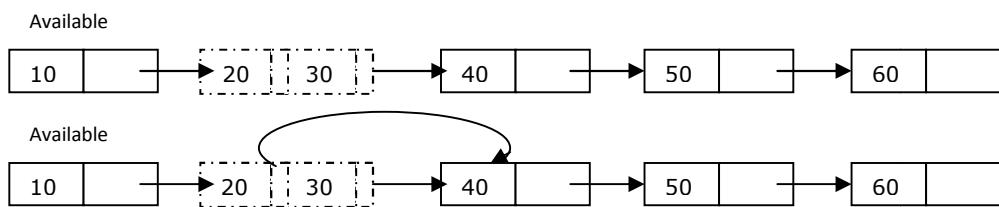


Fig. 6.12 De-allocate block '30'

- The explicit allocation consists of taking a block off the list and de-allocation consist of putting the block back on the list.
  - The advantage of this technique is that there is no space overhead.
- Explicit Allocation of Variable Sized Blocks**
- Due to frequent memory allocation and de-allocation the heap memory becomes fragmented. That means heap may consist of some blocks that are free and some that are allocated.

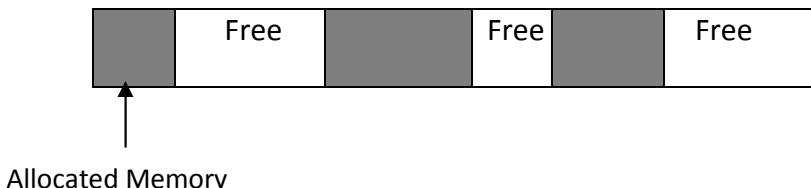


Fig. 6.13 Heap Memory

- In Fig. a fragmented heap memory is shown. Suppose a list of 7 blocks gets allocated and second, fourth and sixth block is de-allocated then fragmentation occurs.
- Thus we get variable sized blocks that are available free. For allocating variable sized blocks some strategies such as first fit, worst fit and best fit are used.
- Sometimes all the free blocks are collected together to form a large free block. This ultimately avoids the problem of fragmentation.

### 2. Implicit Allocation

- The implicit allocation is performed using user program and runtime packages.
- The run time package is required to know when the **storage block** is not in use.
- The format of **storage block** is as shown in Fig 6.14.

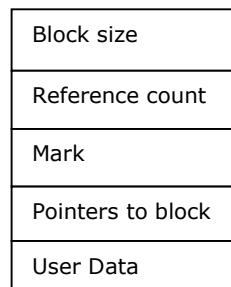


Fig. 6.14 Block format

- There are two approaches used for implicit allocation.

#### Reference count:

- Reference count is a special counter used during implicit memory allocation. If any block is

referred by some another block then its reference count incremented by one. That also means if the reference count of particular block drops down to 0 then, that means that block is not referenced one and hence it can be de-allocated. Reference counts are best used when pointers between blocks never appear in cycle.

### **Marking techniques:**

- This is an alternative approach to determine whether the block is in use or not. In this method, the user program is suspended temporarily and **frozen pointers** are used to mark the blocks that are in use. Sometime bitmaps are used. These pointers are then placed in the heap memory. Again we go through hear memory and mark those blocks which are unused.
- There is one more technique called **compaction** in which all the used blocks are moved at the one end of heap memory, so that all the free blocks are available in one large free block.

### 1. Explain code optimization technique.

#### 1. Common sub expressions elimination

- Compile time evaluation means shifting of computations from run time to compile time.
- There are two methods used to obtain the compile time evaluation.

##### Folding

- In the folding technique the computation of constant is done at compile time instead of run time.

Example : length =  $(22/7)*d$

- Here folding is implied by performing the computation of  $22/7$  at compile time.

##### Constant propagation

- In this technique the value of variable is replaced and computation of an expression is done at compilation time.

Example:

pi = 3.14; r = 5;

Area = pi \* r \* r;

- Here at the compilation time the value of pi is replaced by 3.14 and r by 5 then computation of  $3.14 * 5 * 5$  is done during compilation.

#### 2. Common sub expressions elimination

- The common sub expression is an expression appearing repeatedly in the program which is computed previously.
- If the operands of this sub expression do not get changed at all then result of such sub expression is used instead of re-computing it each time.

Example:

t1 := 4 * i	t1=4*i
t2 := a[t1]	t2=a[t1]
t3 := 4 * j	→ t3=4*j
t4 := 4 * i	t5=n
t5:= n	t6=b[t1]+t5
t6 := b[t4]+t5	

- The common sub expression  $t4:=4*i$  is eliminated as its computation is already in  $t1$  and value of  $i$  is not been changed from definition to use.

#### 3. Variable propagation

- Variable propagation means use of one variable instead of another.

Example:

x = pi;  
area = x \* r \* r;

- The optimization using variable propagation can be done as follows, area = pi \* r \* r;
- Here the variable x is eliminated. Here the necessary condition is that a variable must be assigned to another variable or some constant.

#### 4. Code movement

- There are two basic goals of code movement:

- I. To reduce the size of the code.
- II. To reduce the frequency of execution of code.

Example:

```

for(i=0;i<=10;i++)
{
    x=y*5;
    k=(y*5)+50;
}
temp=y*5
for(i=0;i<=10;i++)
{
    x=temp;
    k=(temp) + 50;
}

```

### Loop invariant computation

- Loop invariant optimization can be obtained by moving some amount of code outside the loop and placing it just before entering in the loop.
- This method is also called code motion.

Example:

```

While(i<=max-1)
{
    sum=sum+a[i];
}
N=max-1;
While(i<=N)
{
    sum=sum+a[i];
}

```

### 5. Strength reduction

- Strength of certain operators is higher than others.
- For instance strength of \* is higher than +.
- In this technique the higher strength operators can be replaced by lower strength operators.

• Example:

```

for(i=1;i<=50;i++)
{
    count = i*7;
}

```

- Here we get the count values as 7, 14, 21 and so on up to less than 50.

- This code can be replaced by using strength reduction as follows

```

temp=7;
for(i=1;i<=50;i++)
{
    count = temp;
    temp = temp+7;
}

```

### 6. Dead code elimination

- A variable is said to be **live** in a program if the value contained into it is subsequently used.
- On the other hand, the variable is said to be **dead** at a point in a program if the value contained into it is never been used. The code containing such a variable supposed to be a dead code. And an optimization can be performed by eliminating such a dead

code.

Example:

```
i=0;
if(i==1)
{
    a=x+5;
}
```

- If statement is a dead code as this condition will never get satisfied hence, statement can be eliminated and optimization can be done.

## 2. Explain Peephole optimization.

**Definition:** Peephole optimization is a simple and effective technique for locally improving target code. This technique is applied to improve the performance of the target program by examining the short sequence of target instructions (called the peephole) and replacing these instructions by shorter or faster sequence whenever possible. Peephole is a small, moving window on the target program.

### Characteristics of Peephole Optimization

- The peephole optimization can be applied on the target code using following characteristic.

#### 1. Redundant instruction elimination.

- Especially the redundant loads and stores can be eliminated in this type of transformations.

Example:

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

- We can eliminate the second instruction since x is in already R0. But if MOV x, R0 is a label statement then we cannot remove it.

#### 2. Unreachable code.

- Especially the redundant loads and stores can be eliminated in this type of transformations.
- An unlabeled instruction immediately following an unconditional jump may be removed.
- This operation can be repeated to eliminate the sequence of instructions.

Example:

```
#define debug 0
If(debug) {
    Print debugging information
}
```

In the intermediate representation the if- statement may be translated as:

```
If debug=1 goto L1
goto L2
L1: print debugging information
L2:
```

- One obvious peephole optimization is to eliminate jumps over jumps. Thus no matter what the value of debug, can be replaced by:

*If debug≠1 goto L2  
Print debugging information*

*L2:*

- Now, since debug is set to 0 at the beginning of the program, constant propagation should replace by

*If 0≠1 goto L2  
Print debugging information*

*L2:*

- As the argument of the first statement evaluates to a constant true, it can be replaced by goto L2.
- Then all the statements that print debugging aids are manifestly unreachable and can be eliminated one at a time.

### 3. Flow of control optimization.

- The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations.
- We can replace the jump sequence.

*Goto L1*

.....

*L1: goto L2*

*By the sequence*

*Goto L2*

.....

*L1: goto L2*

- If there are no jumps to L1 then it may be possible to eliminate the statement L1: goto L2 provided it is preceded by an unconditional jump. Similarly, the sequence

*If a<b goto L1*

.....

*L1: goto L2*

*Can be replaced by*

*If a<b goto L2*

.....

*L1: goto L2*

### 4. Algebraic simplification.

- Peephole optimization is an effective technique for algebraic simplification.
- The statements such as  $x = x + 0$  or  $x := x * 1$  can be eliminated by peephole optimization.

### 5. Reduction in strength

- Certain machine instructions are cheaper than the other.
- In order to improve performance of the intermediate code we can replace these instructions by equivalent cheaper instruction.
- For example,  $x^2$  is cheaper than  $x * x$ . Similarly addition and subtraction is cheaper than

multiplication and division. So we can add effectively equivalent addition and subtraction for multiplication and division.

### 6. Machine idioms

- The target instructions have equivalent machine instructions for performing some operations.
- Hence we can replace these target instructions by equivalent machine instructions in order to improve the efficiency.
- Example: Some machines have auto-increment or auto-decrement addressing modes. These modes can be used in code for statement like  $i=i+1$ .

## 3. Loops in flow graphs.

### 1. Dominators

- In a flow graph, a node  $d$  dominates  $n$  if every path to node  $n$  from initial node goes through  $d$  only. This can be denoted as ' $d$  dom  $n$ '.
- Every initial node dominates all the remaining nodes in the flow graph. Similarly every node dominates itself.

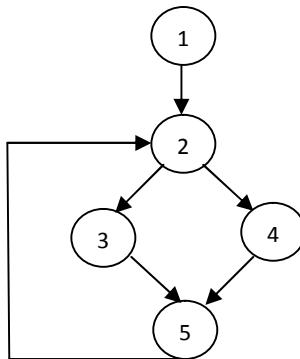


Fig.7.1. Dominators

- Node 1 is initial node and it dominates every node as it is initial node.
- Node 2 dominates 3, 4 and 5.
- Node 3 dominates itself similarly node 4 dominates itself.

### 2. Natural loops

- Loop in a flow graph can be denoted by  $n \rightarrow d$  such that  $d$  dom  $n$ . This edge is called back edges and for a loop there can be more than one back edge. If there is  $p \rightarrow q$  then  $q$  is a head and  $p$  is a tail. And head dominates tail.

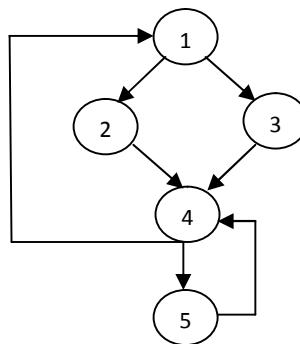


Fig.7.2. Natural loops

- The loop in above graph can be denoted by  $4 \rightarrow 1$  i.e. 1 dom 4. Similarly  $5 \rightarrow 4$  i.e. 4 dom 5.
- The natural loop can be defined by a back edge  $n \rightarrow d$  such there exist a collection of all the node that can reach to  $n$  without going through  $d$  and at the same time  $d$  also can be added to this collection.

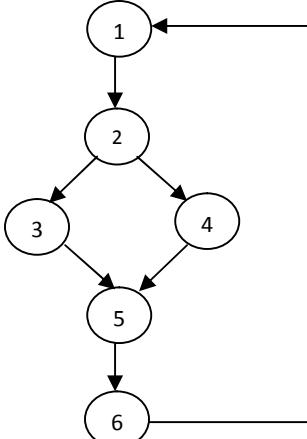


Fig.7.3. Natural loop

- $6 \rightarrow 1$  is a natural loop because we can reach to all the remaining nodes from 6.

### 3. Inner loops

- The inner loop is a loop that contains no other loop.
- Here the inner loop is  $4 \rightarrow 2$  that mean edge given by 2-3-4.

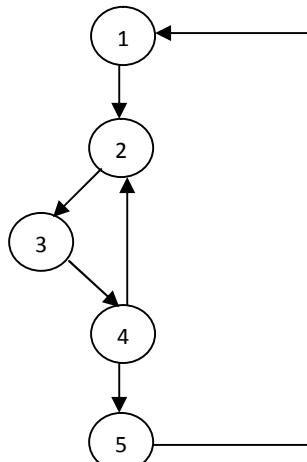


Fig.7.4. Inner loop

### 4. Pre-header

- The pre-header is a new block created such that successor of this block is the block. All the computations that can be made before the header block can be made the pre-header block.

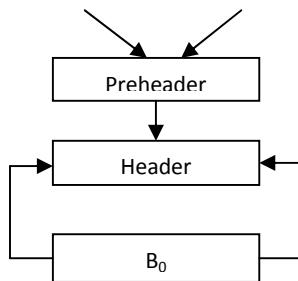


Fig.7.5. Preheader

### 5. Reducible flow graph

- The reducible graph is a flow graph in which there are two types of edges forward edges and backward edges. These edges have following properties,
  - The forward edge forms an acyclic graph.
  - The back edges are such edges whose head dominates their tail.

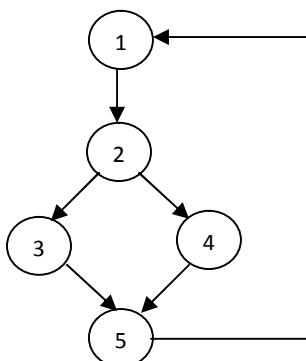


Fig.7.6. Reducible flow graph

Can be reduced as

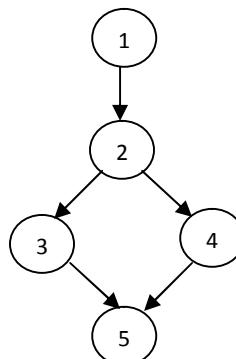


Fig.7.7. Reduced flow graph

- The above flow graph is reducible. We can reduce this graph by removing the edge from 3 to 2. Similarly by removing the back edge from 5 to 1. We can reduce above flow graph and the resultant graph is a cyclic graph.

### 6. Non-reducible flow graph

- A non reducible flow graph is a flow graph in which:
  - There are no back edges.
  - Forward edges may produce cycle in the graph.

Example: Following flow graph is non-reducible.

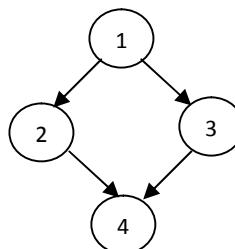


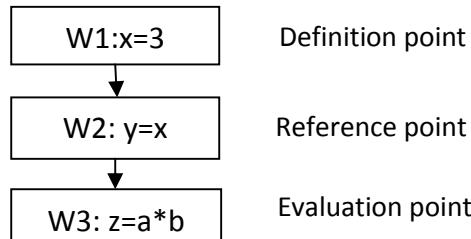
Fig.7.8. Non-Reducible flow graph

### 4. Global data flow analysis.

- Data flow equations are the equations representing the expressions that are appearing in the flow graph.
- Data flow information can be collected by setting up and solving systems of equations that relate information at various points in a program.
- The data flow equation written in a form of equation such that,
$$\text{out}[S] = \text{gen}[S] \cup (\text{in}[S] - \text{kill}[S])$$
- And can be read as “the information at the end of a statement is either generated within a statement, or enters at the beginning and is not killed as control flows through the statement”.
- The details of how dataflow equations are set up and solved depend on three factors.
  - I. The notions of generating and killing depend on the desired information, i.e., on the data flow analysis problem to be solved. Moreover, for some problems, instead of proceeding along with flow of control and defining  $\text{out}[s]$  in terms of  $\text{in}[s]$ , we need to proceed backwards and define  $\text{in}[s]$  in terms of  $\text{out}[s]$ .
  - II. Since data flows along control paths, data-flow analysis is affected by the constructs in a program. In fact, when we write  $\text{out}[s]$  we implicitly assume that there is unique end point where control leaves the statement; in general, equations are set up at the level of basic blocks rather than statements, because blocks do have unique end points.
  - III. There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and even assignments to array variables.

### 5. Data Flow Properties.

- A program point containing the definition is called **definition point**.
- A program point at which a reference to a data item is made is called **reference point**.
- A program point at which some evaluating expression is given is called **evaluation point**.



### 1. Available expression

- An expression  $x+y$  is available at a program point w if and only if along all paths are reaching to w.
  - I. The expression  $x+y$  is said to be available at its evaluation point.
  - II. The expression  $x+y$  is said to be available if no definition of any operand of the expression (here either x or y) follows its last evaluation along the path. In other word, if neither of the two operands get modified before their use.

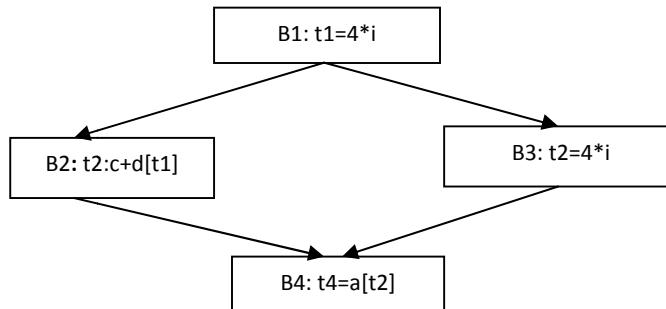


Fig.7.9. Available expression

- Expression  $4 * i$  is the available expression for  $B_2$ ,  $B_3$  and  $B_4$  because this expression has not been changed by any of the block before appearing in  $B_4$ .

### 2. Reaching definition

- A definition D reaches at the point P if there is a path from D to P if there is a path from D to P along which D is not killed.
- A definition D of variable x is killed when there is a redefinition of x.

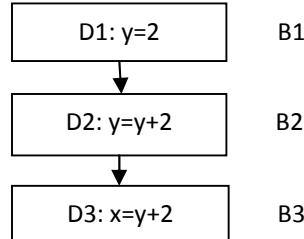


Fig.7.10. Reaching definition

- The definition D1 is reaching definition for block B2, but the definition D1 is not reaching definition for block B3, because it is killed by definition D2 in block B2.

### 3. Live variable

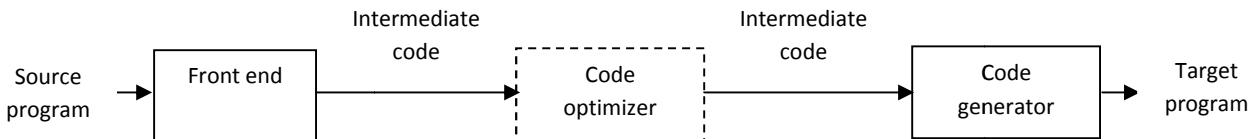
- A live variable x is live at point p if there is a path from p to the exit, along which the value of x is used before it is redefined. Otherwise the variable is said to be dead at the point.

### 4. Busy expression

- An expression e is said to be busy expression along some path  $p_i..p_j$  if and only if an evaluation of e exists along some path  $p_i..p_j$  and no definition of any operand exist before its evaluation along the path.

### 1. Role of code generator.

- The final phase of compilation process is code generation.
- It takes an intermediate representation of the source program as input and produces an equivalent target program as output.



**Fig 8.1 Position of code generator in compilation**

- Target code should have following property,
  1. Correctness
  2. High quality
  3. Efficient use of resources of target code
  4. Quick code generation

### 2. Issues in the design of code generation.

Issues in design of code generator are:

#### 1. Input to the Code Generator

- Input to the code generator consists of the intermediate representation of the source program.
- There are several types for the intermediate language, such as postfix notation, quadruples, and syntax trees or DAGs.
- The detection of semantic error should be done before submitting the input to the code generator.
- The code generation phase requires complete error free intermediate code as an input.

#### 2. Target program

- The output of the code generator is the target program. The output may take on a variety of forms; absolute machine language, relocatable machine language, or assembly language.
- Producing an absolute machine language program as output has the advantage that it can be placed in a location in memory and immediately executed.
- Producing a relocatable machine language program as output is that the subroutine can be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader.
- Producing an assembly language program as output makes the process of code generation somewhat easier .We can generate symbolic instructions and use the macro facilities of the assembler to help generate code.

#### 3. Memory management

- Mapping names in the source program to addresses of data objects in run time memory is done cooperatively by the front end and the code generator.
- We assume that a name in a three-address statement refers to a symbol table entry for the name.

- From the symbol table information, a relative address can be determined for the name in a data area.

### 4. Instruction selection

- If we do not care about the efficiency of the target program, instruction selection is straightforward. It requires special handling. For example, the sequence of statements

$a := b + c$

$d := a + e$

would be translated into

MOV b, R0

ADD c, R0

MOV R0, a

MOV a, R0

ADD e, R0

MOV R0, d

- Here the fourth statement is redundant, so we can eliminate that statement.

### 5. Register allocation

- If the instruction contains register operands then such a use becomes shorter and faster than that of using in memory.
- The use of registers is often subdivided into two sub problems:
- During register allocation, we select the set of variables that will reside in registers at a point in the program.
- During a subsequent register assignment phase, we pick the specific register that a variable will reside in.
- Finding an optimal assignment of registers to variables is difficult, even with single register value.
- Mathematically the problem is NP-complete.

### 6. Choice of evaluation

- The order in which computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others. Picking a best order is another difficult, NP-complete problem.

### 7. Approaches to code generation

- The most important criterion for a code generator is that it produces correct code.
- Correctness takes on special significance because of the number of special cases that code generator must face.
- Given the premium on correctness, designing a code generator so it can be easily implemented, tested, and maintained is an important design goal.

## 3. The target machine and instruction cost.

- Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator.
- We will assume our target computer models a three-address machine with load and store operations, computation operations, jump operations, and conditional jumps. The

underlying computer is a byte-addressable machine with  $n$  general-purpose registers,  $R_0, R_1, \dots, R_n$

- The two address instruction of the form  $op \ source, destination$
- It has following opcodes,
  - MOV (move source to destination)
  - ADD (add source to destination)
  - SUB (subtract source to destination)
- The address modes together with the assembly language forms and associated cost as follows:

Mode	Form	Address	Extra cost
Absolute	M	M	1
Register	R	R	0
Indexed	k(R)	$k + \text{contents}(R)$	1
Indirect register	*R	$\text{contents}(R)$	0
Indirect indexed	*k(R)	$\text{contents}(k + \text{contents}(R))$	1

Table 8.1 Addressing modes

### Instruction cost:

- The instruction cost can be computed as one plus cost associated with the source and destination addressing modes given by “extra cost”.
- Calculate cost for following:
  - MOV B,R0
  - ADD C,R0
  - MOV R0,A

Instruction cost,

$$\text{MOV B,R0} \rightarrow \text{cost} = 1+1+0=2$$

$$\text{ADD C,R0} \rightarrow \text{cost} = 1+1+0=2$$

$$\text{MOV R0,A} \rightarrow \text{cost} = 1+0+1=2$$

Total cost=6

## 4. Basic Blocks.

- A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.
- The following sequence of three-address statements forms a basic block:

t1 := a\*a

t2 := a\*b

t3 := 2\*t2

t4 := t1+t3

t5 := b\*b

t6 := t4+t5

- Some terminology used in basic blocks are given below:
- A three-address statement  $x:=y+z$  is said to **define**  $x$  and to **use**  $y$  or  $z$ . A name in a basic block is said to be *live* at a given point if its value is used after that point in the program, perhaps in another basic block.
- The following algorithm can be used to partition a sequence of three-address statements into basic blocks.

### Algorithm: Partition into basic blocks.

Input: A sequence of three-address statements.

Output: A list of basic blocks with each three-address statement in exactly one block.

Method:

1. We first determine the set of **leaders**, for that we use the following rules:
  - i) The first statement is a leader.
  - ii) Any statement that is the target of a conditional or unconditional goto is a leader.
  - iii) Any statement that immediately follows a goto or conditional goto statement is a leader.
2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

Example: Program to compute dot product

```

begin
    prod := 0;
    i := 1;
    do
        prod := prod + a[t1] * b[t2];
        i := i+1;
    while i<= 20
    end

```

Three address code for the above program,

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

- Let us apply an algorithm to the three-address code to determine its basic blocks.
- Statement (1) is a leader by rule (I) and statement (3) is a leader by rule (II), since the last statement can jump to it.

- Therefore, statements (1) and (2) form a basic block.
- The remainder of the program beginning with statement (3) forms a second basic block.

### 5. Transformations on basic block

- A number of transformations can be applied to a basic block without changing the set of expressions computed by the block.
- Many of these transformations are useful for improving the quality of the code.
- There are two important classes of local transformations that can be applied to basic block. These are,
  1. Structure preserving transformation.
  2. Algebraic transformation.

#### 1. Structure Preserving Transformations

The primary structure-preserving transformations on basic blocks are,

##### A. Common sub-expression elimination.

- Consider the basic block,  
 $a := b + c$   
 $b := a - d$   
 $c := b + c$   
 $d := a - d$
- The second and fourth statements compute the same expression, hence this basic block may be transformed into the equivalent block  
 $a := b + c$   
 $b := a - d$   
 $c := b + c$   
 $d := b$
- Although the 1<sup>st</sup> and 3<sup>rd</sup> statements in both cases appear to have the same expression on the right, the second statement redefines b. Therefore, the value of b in the 3<sup>rd</sup> statement is different from the value of b in the 1<sup>st</sup>, and the 1<sup>st</sup> and 3<sup>rd</sup> statements do not compute the same expression.

##### B. Dead-code elimination.

- Suppose x is dead, that is, never subsequently used, at the point where the statement  $x := y + z$  appears in a basic block. Then this statement may be safely removed without changing the value of the basic block.

##### C. Renaming of temporary variables.

- Suppose we have a statement  $t := b + c$ , where t is a temporary. If we change this statement to  $u := b + c$ , where u is a new temporary variable, and change all uses of this instance of t to u, then the value of the basic block is not changed.
- In fact, we can always transform a basic block into an equivalent block in which each statement that defines a temporary defines a new temporary. We call such a basic block a *normal-form* block.

##### D. Interchange of two independent adjacent statements.

- Suppose we have a block with the two adjacent statements,

$t1 := b+c$   
 $t2 := x+y$

- Then we can interchange the two statements without affecting the value of the block if and only if neither  $x$  nor  $y$  is  $t1$  and neither  $b$  nor  $c$  is  $t2$ . A normal-form basic block permits all statement interchanges that are possible.

### 2. Algebraic transformation

- Countless algebraic transformation can be used to change the set of expressions computed by the basic block into an algebraically equivalent set.
- The useful ones are those that simplify expressions or replace expensive operations by cheaper one.
- Example:  $x=x+0$  or  $x=x+1$  can be eliminated.

### 3. Flow graph

- A graph representation of three-address statements, called a **flow graph**, is useful for understanding code-generation algorithms.
- Nodes in the flow graph represent computations, and the edges represent the flow of control.
- Example of flow graph for following three address code,

(1)  $\text{prod}=0$   
(2)  $i=1$   
(3)  $t1 := 4*i$   
(4)  $t2 := a [ t1 ]$   
(5)  $t3 := 4*i$   
(6)  $t4 := b [ t3 ]$   
(7)  $t5 := t2*t4$   
(8)  $t6 := \text{prod} + t5$   
(9)  $\text{prod} := t6$   
(10)  $t7 := i+1$   
(11)  $i := t7$   
(12) if  $i \leq 20$  goto (3)

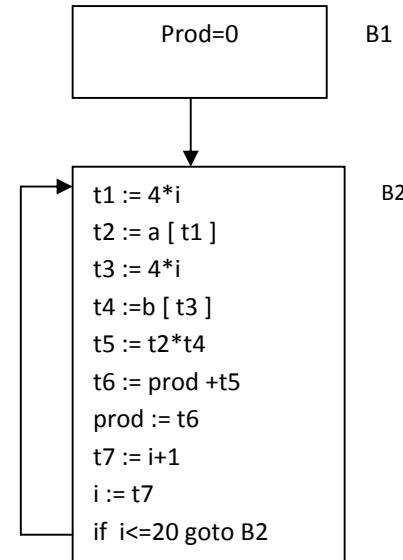


Fig 8.2 flow graph

## 6. Next-Use information.

- The next-use information is a collection of all the names that are useful for next subsequent statement in a block. The use of a name is defined as follows,
- Consider a statement,

$x := i$   
 $j := x \text{ op } y$

- That means the statement  $j$  uses value of  $x$ .
- The next-use information can be collected by making the backward scan of the programming code in that specific block.

### Storage for Temporary Names

- For the distinct names each time a temporary is needed. And each time a space gets allocated for each temporary. To have optimization in the process of code generation we pack two temporaries into the same location if they are not live simultaneously.
- Consider three address code as,

t1 := a * a	t1 := a * a
t2 := a * b	t2 := a * b
t3 := 4 * t2	t2 := 4 * t2
t4 := t1+t3	t1 := t1+t2
t5 := b * b	t2 := b * b
t6 := t4+t5	t1 := t1+t2



## 7. Register and address descriptors.

- The code generator algorithm uses descriptors to keep track of register contents and addresses for names.
- Address descriptor** stores the location where the current value of the name can be found at run time. The information about locations can be stored in the symbol table and is used to access the variables.
- Register descriptor** is used to keep track of what is currently in each register. The register descriptor shows that initially all the registers are empty. As the generation for the block progresses the registers will hold the values of computation.

## 8. Register allocation and assignment.

- Efficient utilization of registers is important in generating good code.
- There are four strategies for deciding what values in a program should reside in a registers and which register each value should reside. Strategies are,

### 1. Global register allocation

- Following are the strategies adopted while doing the global register allocation.
- The global register allocation has a strategy of storing the most frequently used variables in fixed registers throughout the loop.
- Another strategy is to assign some fixed number of global registers to hold the most active values in each inner loop.
- The registers are not already allocated may be used to hold values local to one block.
- In certain languages like C or Bliss programmer can do the register allocation by using register declaration to keep certain values in register for the duration of the procedure.

### 2. Usage count

- The usage count is the count for the use of some variable x in some register used in any basic block.

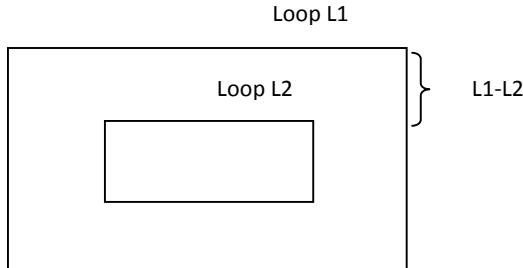
- The usage count gives the idea about how many units of cost can be saved by selecting a specific variable for global register allocation.
- The approximate formula for usage count for the Loop L in some basic block B can be given as,

$$\sum_{\text{block } B \text{ in } L} (\text{use}(x, B) + 2 * \text{live}(x, B))$$

- Where  $\text{use}(x, B)$  is number of times x used in block B prior to any definition of x
- $\text{live}(x, B) = 1$  if x is live on exit from B; otherwise  $\text{live}(x) = 0$ .

### 3. Register assignment for outer loop

- Consider that there are two loops L1 is outer loop and L2 is an inner loop, and allocation of variable a is to be done to some register. The approximate scenario is as given below,



**Fig 8.3 Loop representation**

Following criteria should be adopted for register assignment for outer loop,

- If a is allocated in loop L2 then it should not be allocated in L1 - L2.
- If a is allocated in L1 and it is not allocated in L2 then store a on entrance to L2 and load a while leaving L2.
- If a is allocated in L2 and not in L1 then load a on entrance of L2 and store a on exit from L2.

### 4. Register allocation for graph coloring

The graph coloring works in two passes. The working is as given below,

- In the first pass the specific machine instruction is selected for register allocation. For each variable a symbolic register is allocated.
- In the second pass the register inference graph is prepared. In register inference graph each node is a symbolic registers and an edge connects two nodes where one is live at a point where other is defined.
- Then a graph coloring technique is applied for this register inference graph using k-color. The k-colors can be assumed to be number of assignable registers. In graph coloring technique no two adjacent nodes can have same color. Hence in register inference graph using such graph coloring principle each node (actually a variable) is assigned the symbolic registers so that no two symbolic registers can interfere with each other with assigned physical registers.

## 9. DAG representation of basic blocks.

- The directed acyclic graph is used to apply transformations on the basic block.

- A DAG gives a picture of how the value computed by each statement in a basic block used in a subsequent statements of the block.
- To apply the transformations on basic block a DAG is constructed from three address statement.
- A DAG can be constructed for the following type of labels on nodes,
  1. Leaf nodes are labeled by identifiers or variable names or constants. Generally leaves represent r-values.
  2. Interior nodes store operator values.
  3. Nodes are also optionally given a sequence of identifiers for label.
- The DAG and flow graphs are two different pictorial representations. Each node of the flow graph can be represented by DAG because each node of the flow graph is a basic block.

Example:

```
sum = 0;
for (i=0;i<= 10;i++)
    sum = sum+a[t1];
```

Solution :

The three address code for above code is

1. sum := 0
2. i := 0
3. t1 := 4\*i
4. t2 := a[t1]
5. t3 := sum + t2
6. sum := t3
7. t4 := i+1;
8. i := t4
9. if i <= 10 goto (3)

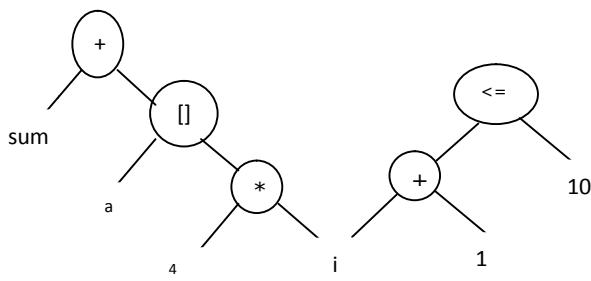


Fig 8.4 DAG for block B2

### Algorithm for Construction of DAG

- We assume the three address statement could of following types,
  - Case (i)  $x := y \text{ op } z$
  - Case (ii)  $x := \text{op } y$
  - Case (iii)  $x := y$

- With the help of following steps the DAG can be constructed.
  - Step 1: If y is undefined then create node(y). Similarly if z is undefined create a node(z)
  - Step 2: For the case(i) create a node(op) whose left child is node(y) and node(z) will be the right child. Also check for any common sub expressions. For the case(ii) determine whether is a node labeled op,such node will have a child node(y). In case(iii) node n will be node(y).
  - Step 3: Delete x from list of identifiers for node(x). Append x to the list of attached identifiers for node n found in 2.

### Applications of DAG

The DAGs are used in,

1. Determining the common sub-expressions.
2. Determining which names are used inside the block and computed outside the block.
3. Determining which statements of the block could have their computed value outside the block.
4. Simplifying the list of quadruples by eliminating the common sub-expressions and not performing the assignment of the form  $x:=y$  unless and until it is a must.

## 10. Generating code from DAGs.

- Methods generating code from DAG as shown in Figure 8.5.

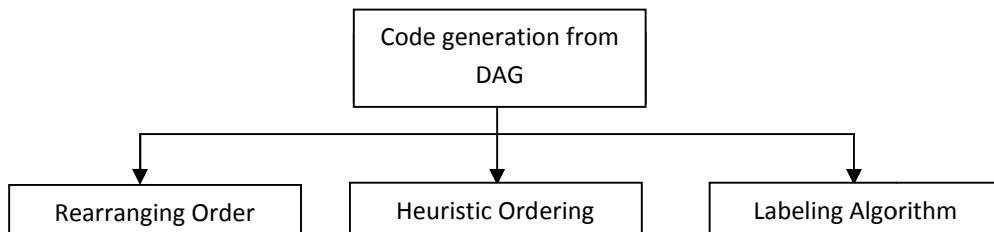


Fig. 8.5. Methods to generate code from DAG

### 1. Rearranging Order

- The order of three address code affects the cost of the object code being generated. In the senses that by changing the order in which computations are done we can obtain the object code with minimum cost.
- Consider the following code,

```

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

- The code can be generated by translating the three address code line by line.

```

MOV a, R0
ADD b, R0
MOV c, R1
ADD d, R1
MOV R0, t1
MOV e, R0
  
```

```
SUB R1, R0
MOV t1, R1
SUB R0, R1
MOV R1, t4
```

- Now if we change the sequence of the above three address code.

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

- Then we can get improved code as

```
MOV c, R0
ADD d, R0
MOV e, R1
SUB R0, R1
MOV a, R0
ADD b, R0
SUB R1, R0
MOV R0, t4
```

### 2. Heuristic ordering

- The heuristic ordering algorithm is as follows:

1. Obtain all the interior nodes. Consider these interior nodes as unlisted nodes.
2. while( unlisted interior nodes remain)
3.     {
4.         pick up an unlisted node n, whose parents have been listed
5.         list n;
6.         while(the leftmost child m of n has no unlisted parent AND is not leaf
7.             {
8.                 List m;
9.                 n=m;
10.             }
11.     }

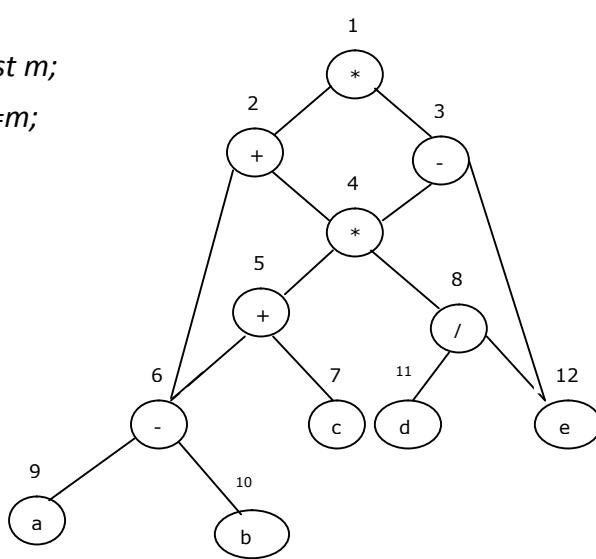


Fig 8.6 A DAG

- The DAG is first numbered from top to bottom and from left to right. Then consider the unlisted interior nodes 1 2 3 4 5 6 8.
- Initially the only node with unlisted parent is 1. ( Set n=1 by line 4 of algorithm)
- Now left argument of 1 is 2 and parent of 2 is 1 which is listed. Hence list 2. Set n=2 by line 7) of algorithm

1 2 3 4 5 6 8

- Now we will find the leftmost node of 2 and that is 6. But 6 has unlisted parent 5. Hence we cannot select 6.
- We therefore can switch to 3. The parent of 3 is 1 which is listed one. Hence list 3 set n=3

1 2 3 4 5 6 8

- The left of 3 is 4. As parent of 4 is 3 and that is listed hence list 4. Left of 4 is 5 which has listed parent (i.e. 4) hence list 5. Similarly list 6

1 2 3 4 5 6 8

- As now only 8 is remaining from the unlisted interior nodes we will list it.
- Hence the resulting list is 1 2 3 4 5 6 8.
- Then the order of computation is decided by reversing this list.
- We get the order of evaluation as 8 6 5 4 3 2 1.
- That also means that we have to perform the computations at these nodes in the given order,

T8=d/e

T6=a-b

T5=t6+c

T4=t5\*t8

T3=t4-e

T2=t6+t4

T1=t2\*t3

### 3. Labeling algorithm

- The labeling algorithm generates the optimal code for given expression in which minimum registers are required.
- Using labeling algorithm the labeling can be done to tree by visiting nodes in bottom up order.
- By this all the child nodes will be labeled its parent nodes.
- For computing the label at node n with the label L1 to left child and label L2 to right child as,

Label (n) = max(L1,L2) if L1 not equal to L2

Label(n) = L1+1 if L1=L2

- We start in bottom-up fashion and label left leaf as 1 and right leaf as 0.

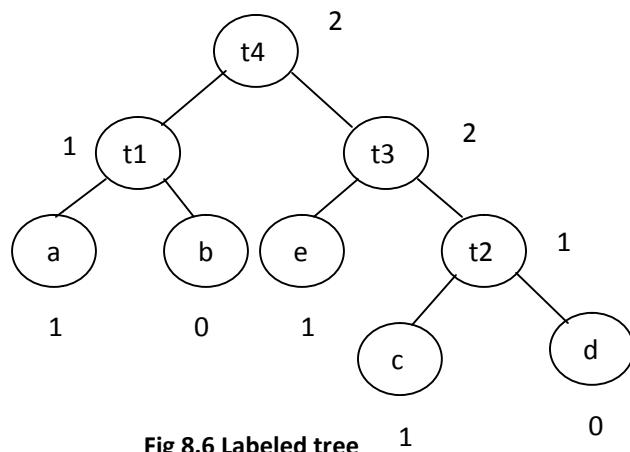


Fig 8.6 Labeled tree

# Unit – 1

# Overview of the Compiler

## &

## it's Structure



**Prof. Dixita B. Kagathara**  
Computer Engineering Department  
Darshan Institute of Engineering & Technology, Rajkot  
✉ dixita.kagathara@darshan.ac.in  
📞 +91 - 97277 47317 (CE Department)

# Topics to be covered

- Language Processor
- Translator
- Analysis synthesis model of compilation
- Phases of compiler
- Grouping of the Phases
- Difference between compiler & interpreter
- Context of compiler (Cousins of compiler)
- Pass structure
- Types of compiler
- The Science of building a compiler

# What do you see?

```
10101001 00000000 10101001 00000000  
10101001 00000000 10101001 00000000  
10000101 00000000 110101001 00000000  
10101001 00000010 10101001 00000010  
10000101 00000010 10101001 00000010  
10100000 00000000 10101001 00000010  
10101001 00000000 110101001 00000010  
10010001 00000001 10000101 00000000  
10000101 00000001 10101001 00000010  
10101001 00000010 10101001 00000010  
10101001 00000010 10101001 00000010
```

**Binary  
program**

**What does it  
mean????**

# What do you see?



# Semantic gap

```
10101001 00000000 10101001 00000000  
10101001 00000000 10101001 00000000  
10000101 00000000 110101001 00000000  
10101001 00000010 10101001 00000010  
10000101 00000010 10101001 00000010  
10100000 00000000 10101001 00000010  
10101001 00000000 110101001 00000010  
10010001 00000001 10000101 00000000
```

**Actual Data**



**Human perception**

# Language processor

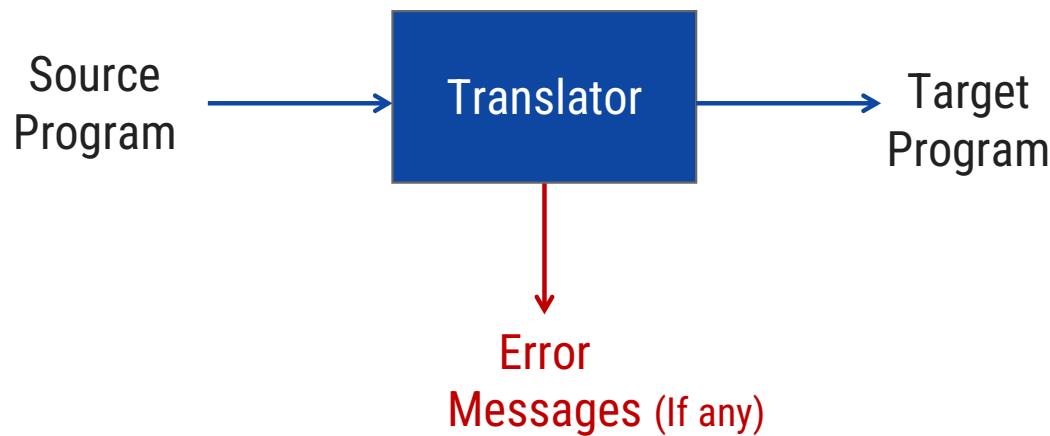
- ▶ Language processor is a software which **bridges semantic gap**.
- ▶ A **language processor** is a software program designed or used to perform tasks such as processing program code to machine code.



# Translator

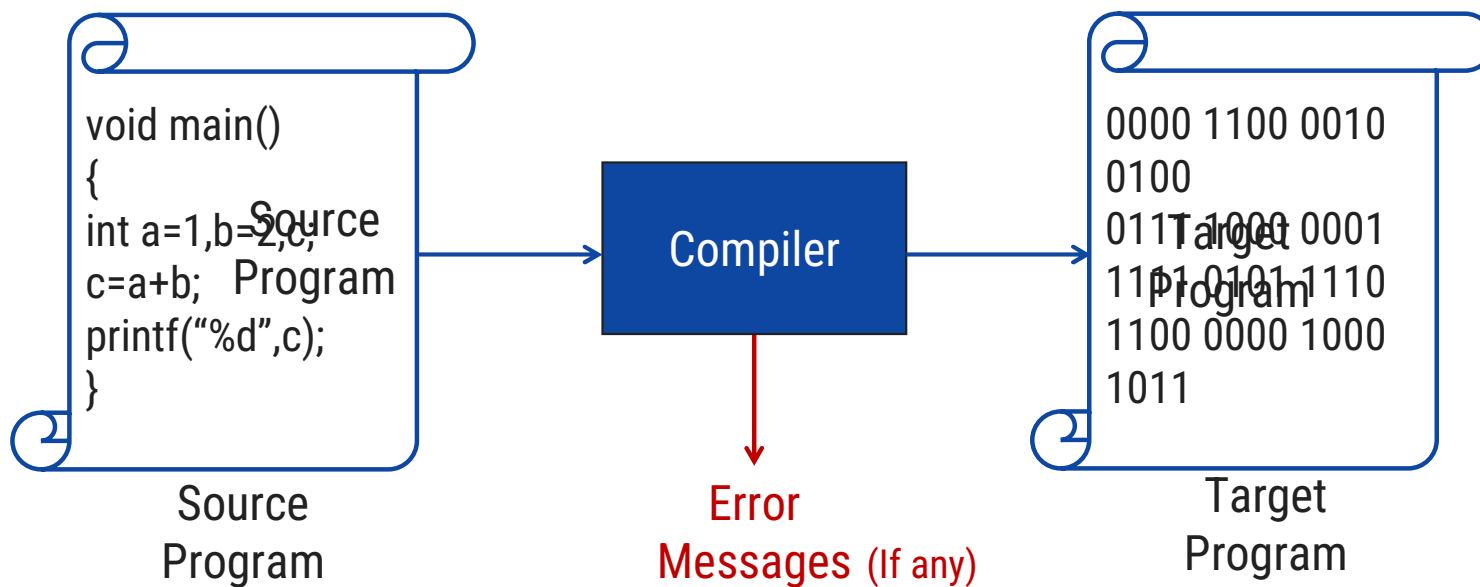
# Translator

- ▶ A translator is a program that **takes one form of program as input** and **converts it into another form**.
- ▶ Types of translators are:
  1. Compiler
  2. Interpreter
  3. Assembler



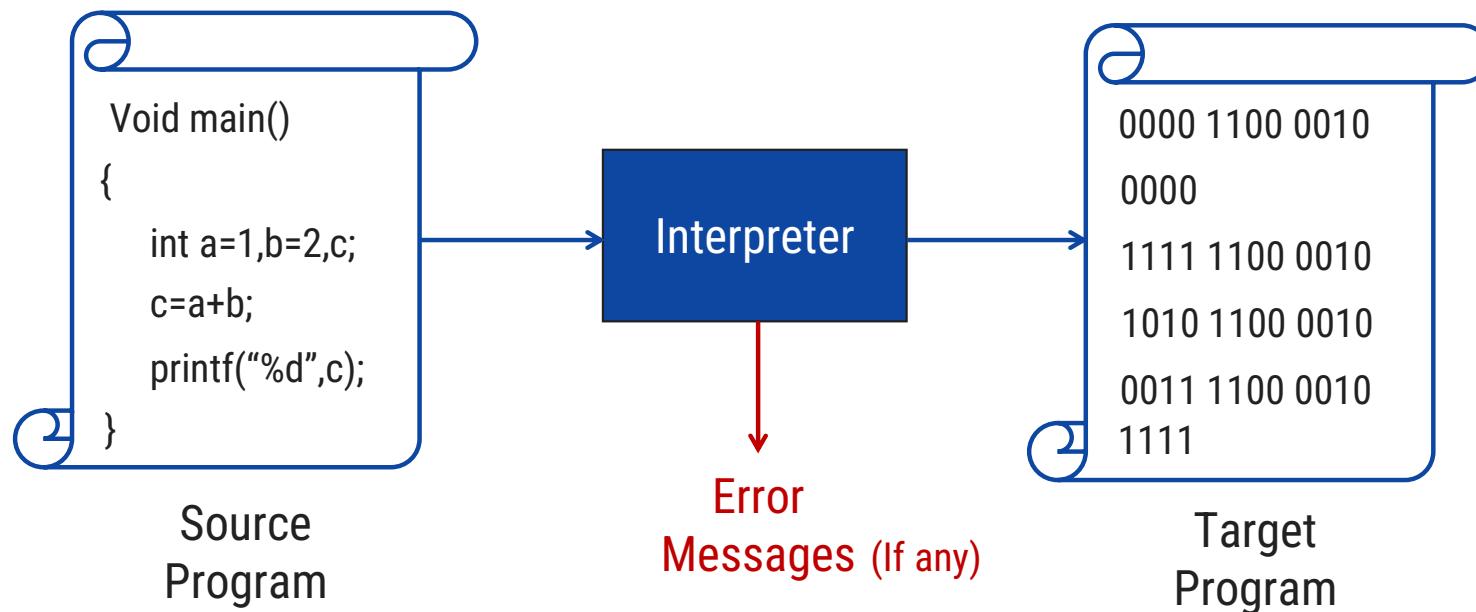
# Compiler

- ▶ A compiler is a program that reads a program written in source language and translates it into an equivalent program in target language.



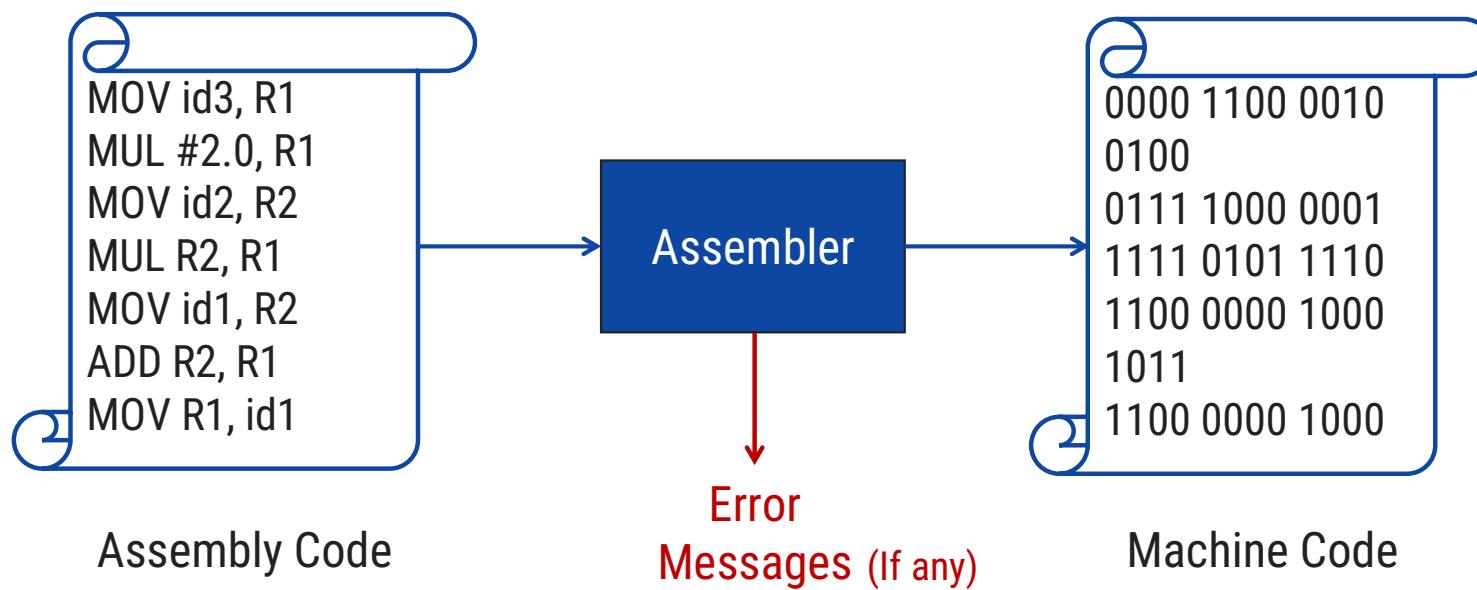
# Interpreter

- Interpreter is also program that reads a program written in source language and translates it into an equivalent program in target language line by line.



# Assembler

- Assembler is a translator which takes the assembly code as an input and generates the machine code as an output.

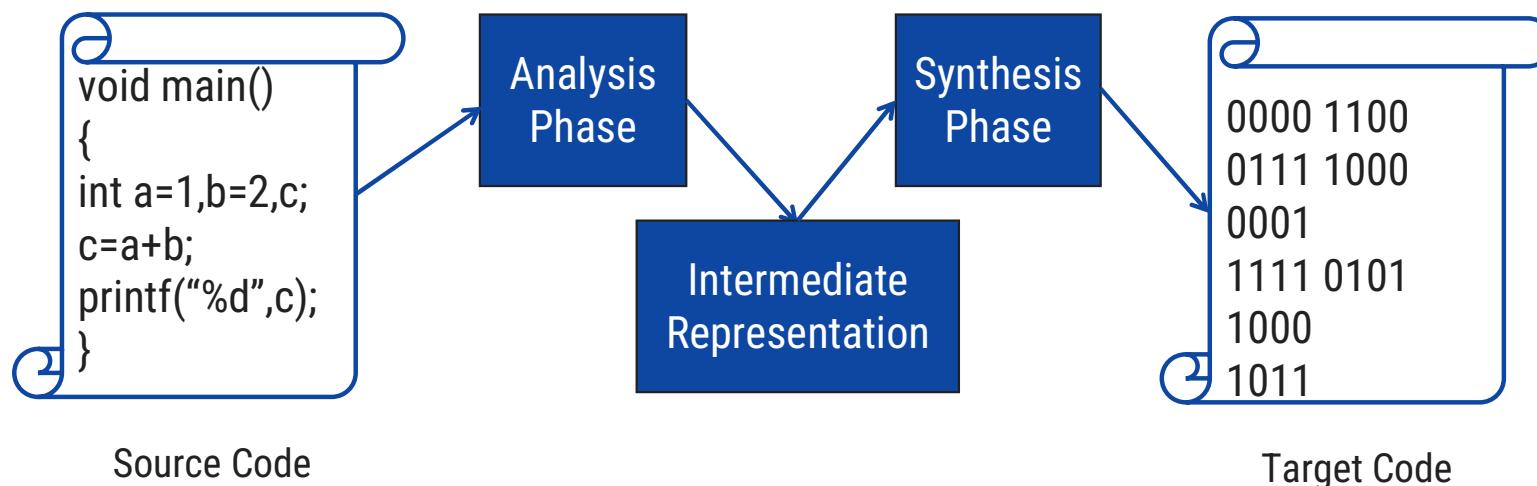


# Analysis Synthesis model of compilation

# Analysis synthesis model of compilation

- ▶ There are two parts of compilation.

1. Analysis Phase
2. Synthesis Phase



# Analysis phase & Synthesis phase

## Analysis Phase

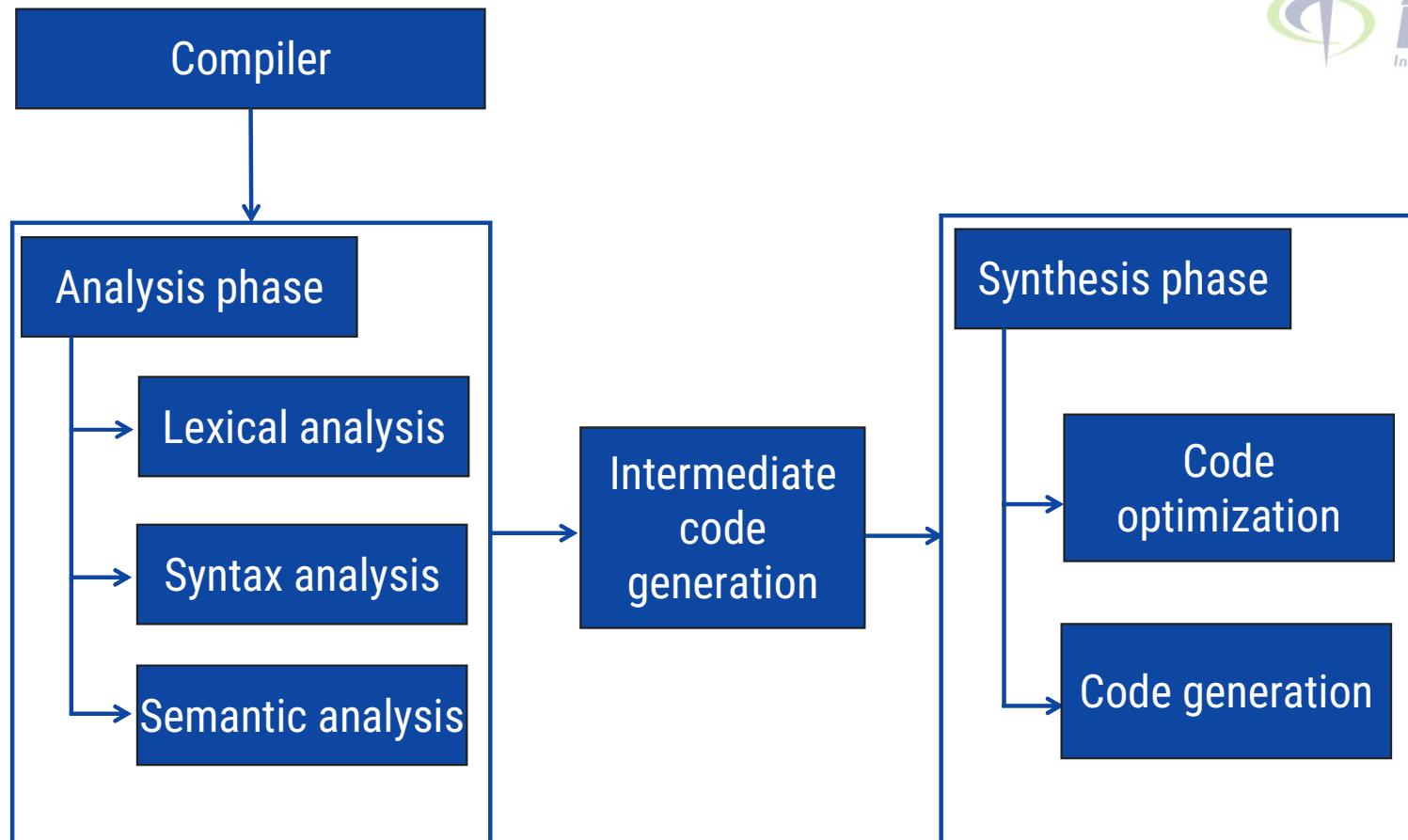
- ▶ Analysis part **breaks up the source program into constituent pieces** and creates an intermediate representation of the source program.
- ▶ Analysis phase consists of three sub phases:
  1. Lexical analysis
  2. Syntax analysis
  3. Semantic analysis

## Synthesis Phase

- ▶ The synthesis part constructs the desired target program from the intermediate representation.
- ▶ Synthesis phase consist of the following sub phases:
  1. Code optimization
  2. Code generation

# Phases of compiler

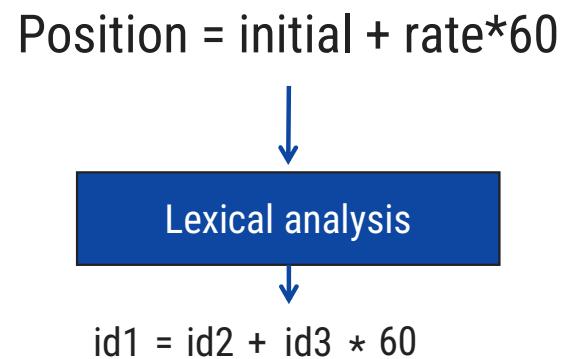
# Phases of compiler



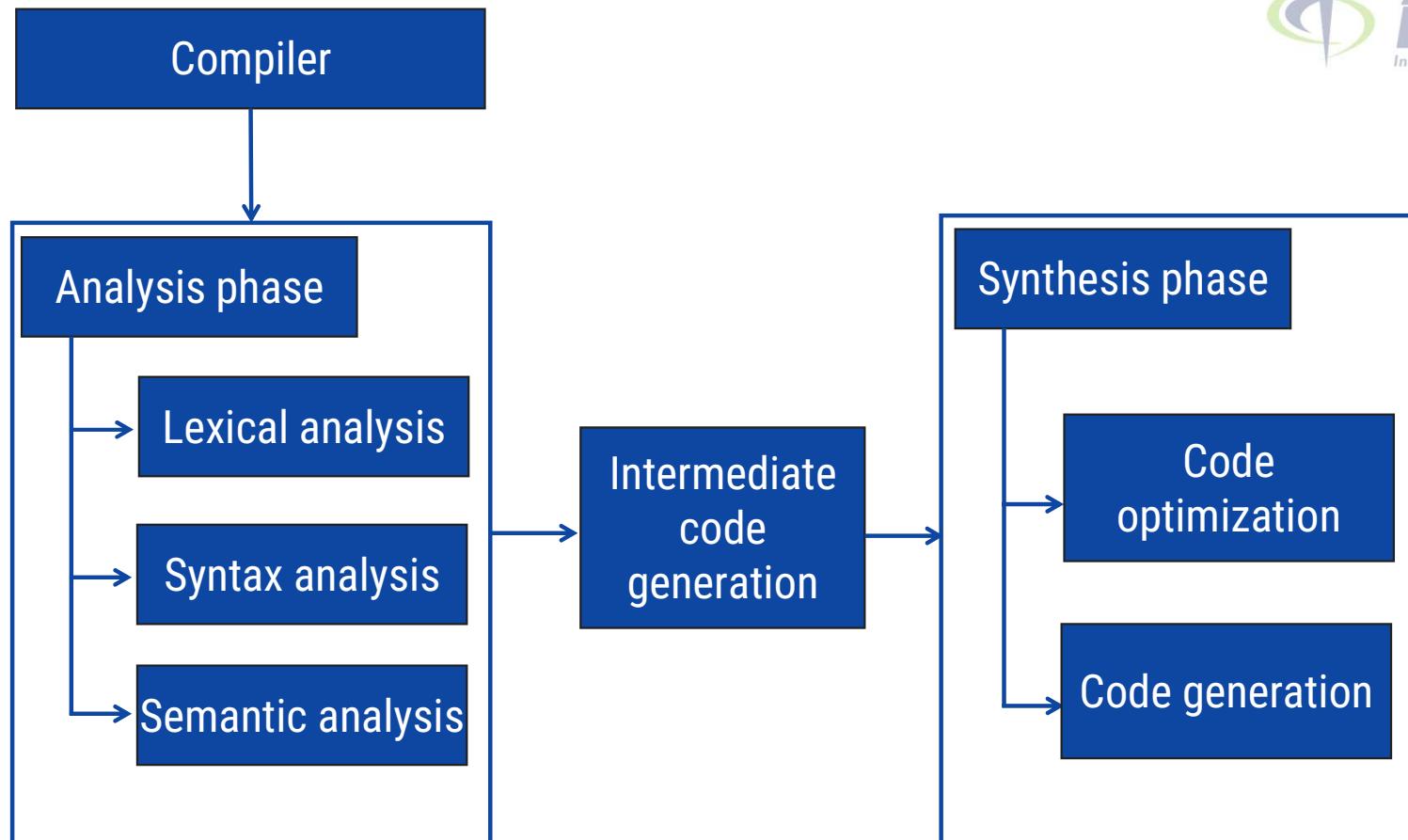
# Lexical analysis

- ▶ Lexical Analysis is also called **linear analysis** or **scanning**.
- ▶ Lexical Analyzer divides the given source statement into the **tokens**.
- ▶ Ex: **Position = initial + rate \* 60** would be grouped into the following tokens:

Position (identifier)  
= (Assignment symbol)  
initial (identifier)  
+ (Plus symbol)  
rate (identifier)  
\* (Multiplication symbol)  
60 (Number)



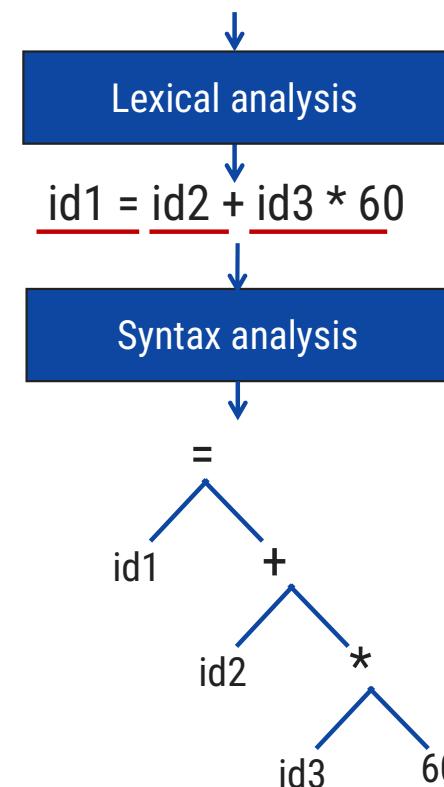
# Phases of compiler



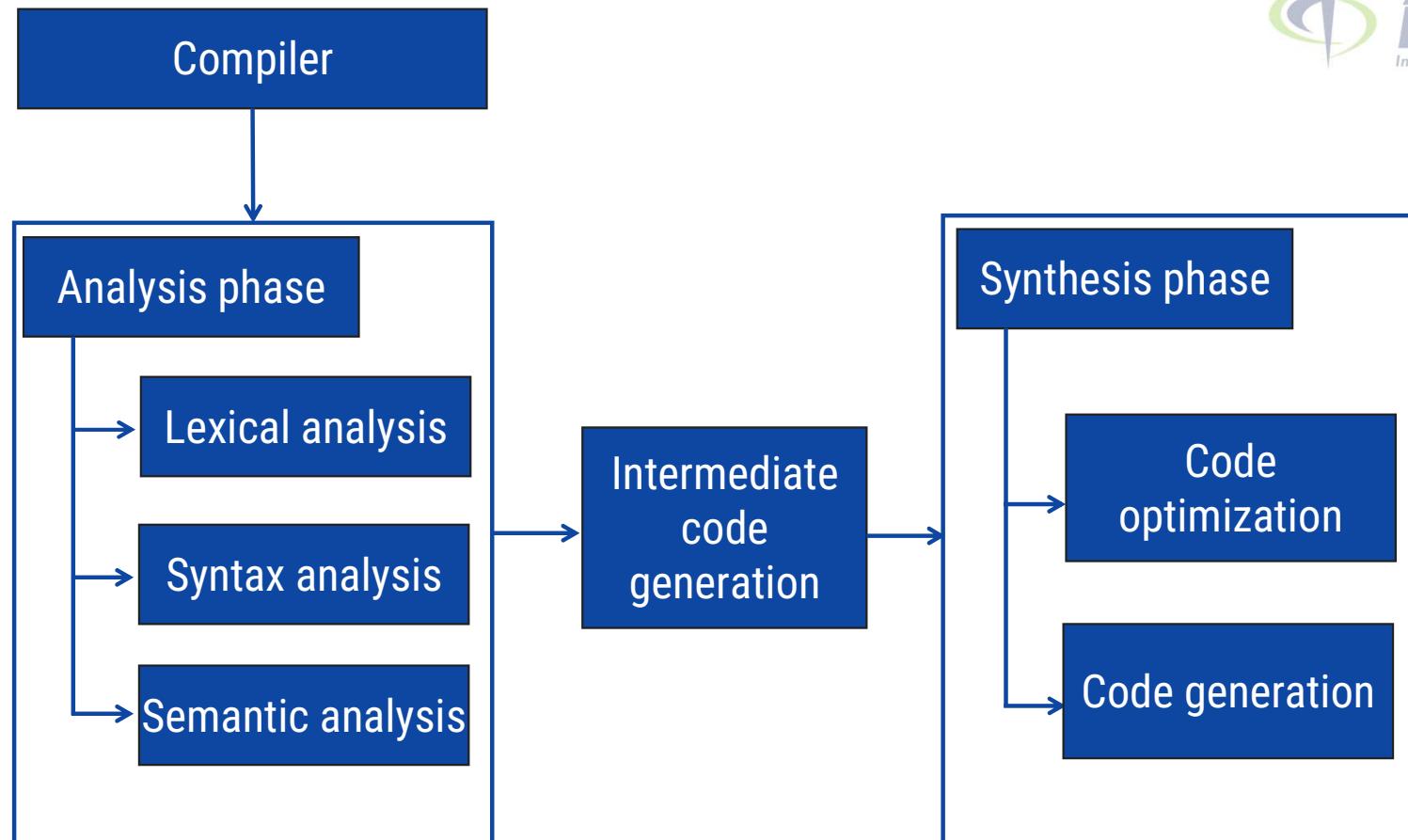
# Syntax analysis

- ▶ Syntax Analysis is also called **Parsing** or **Hierarchical Analysis**.
- ▶ The syntax analyzer checks each line of the code and spots every tiny mistake.
- ▶ If code is error free then syntax analyzer generates the tree.

Position = initial + rate\*60



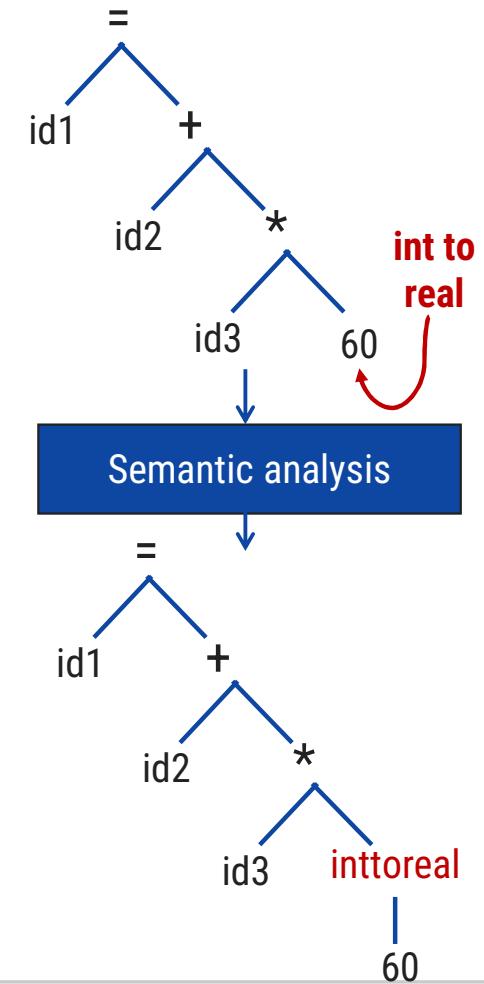
# Phases of compiler



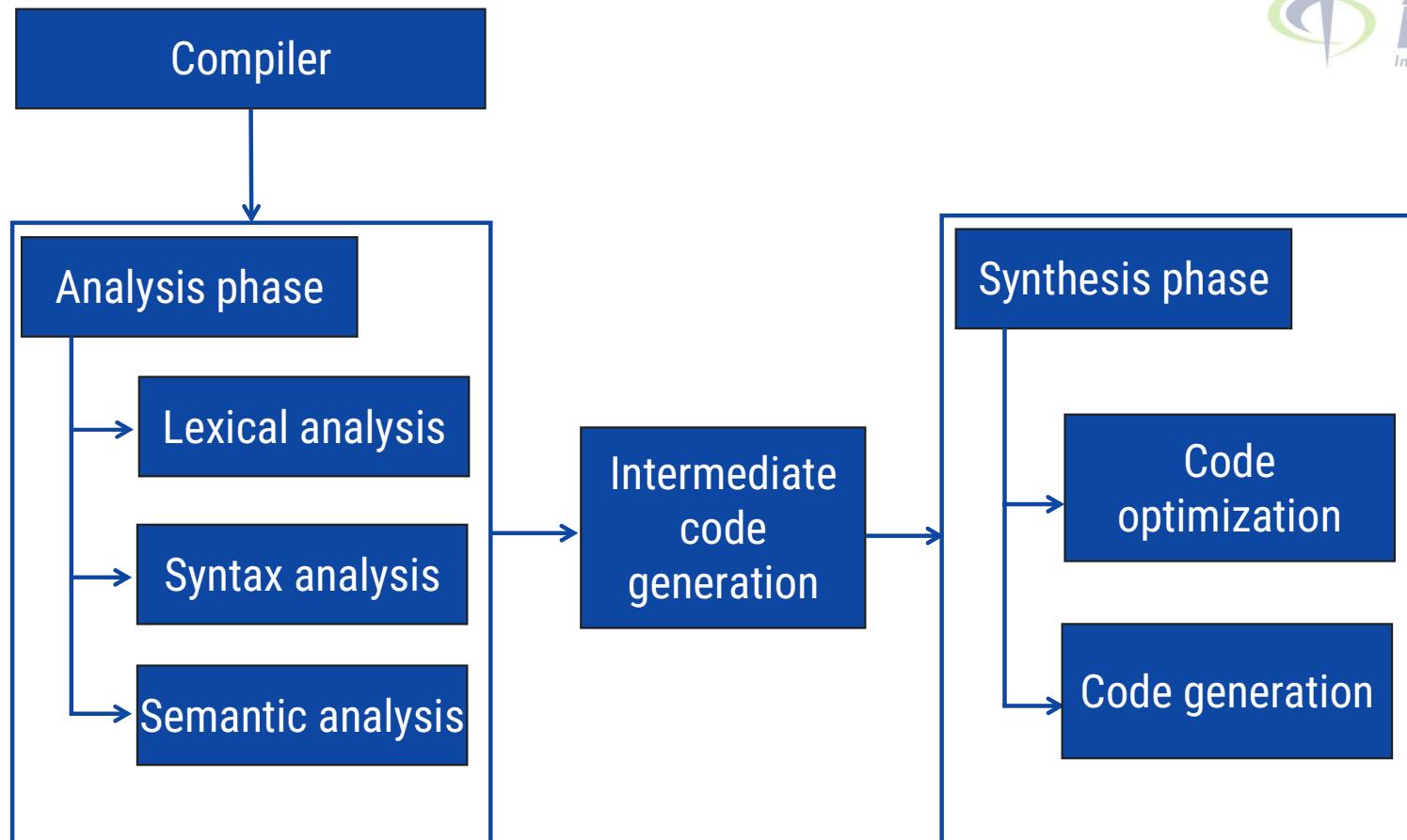
# Semantic analysis

- ▶ Semantic analyzer determines the **meaning of a source string**.
- ▶ It performs following operations:
  1. matching of parenthesis in the expression.
  2. Matching of if..else statement.
  3. Performing arithmetic operation that are type compatible.
  4. Checking the scope of operation.

\*Note: Consider id1, id2 and id3 are real

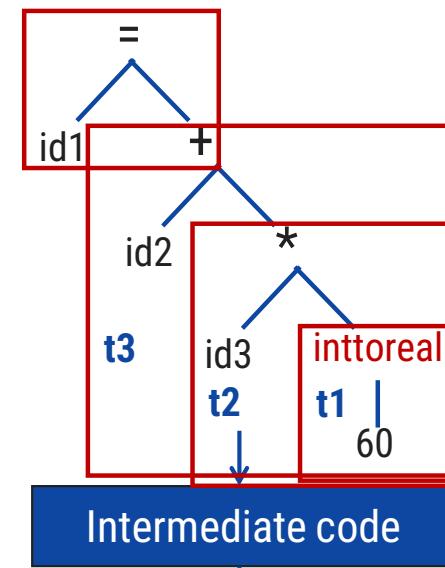


# Phases of compiler



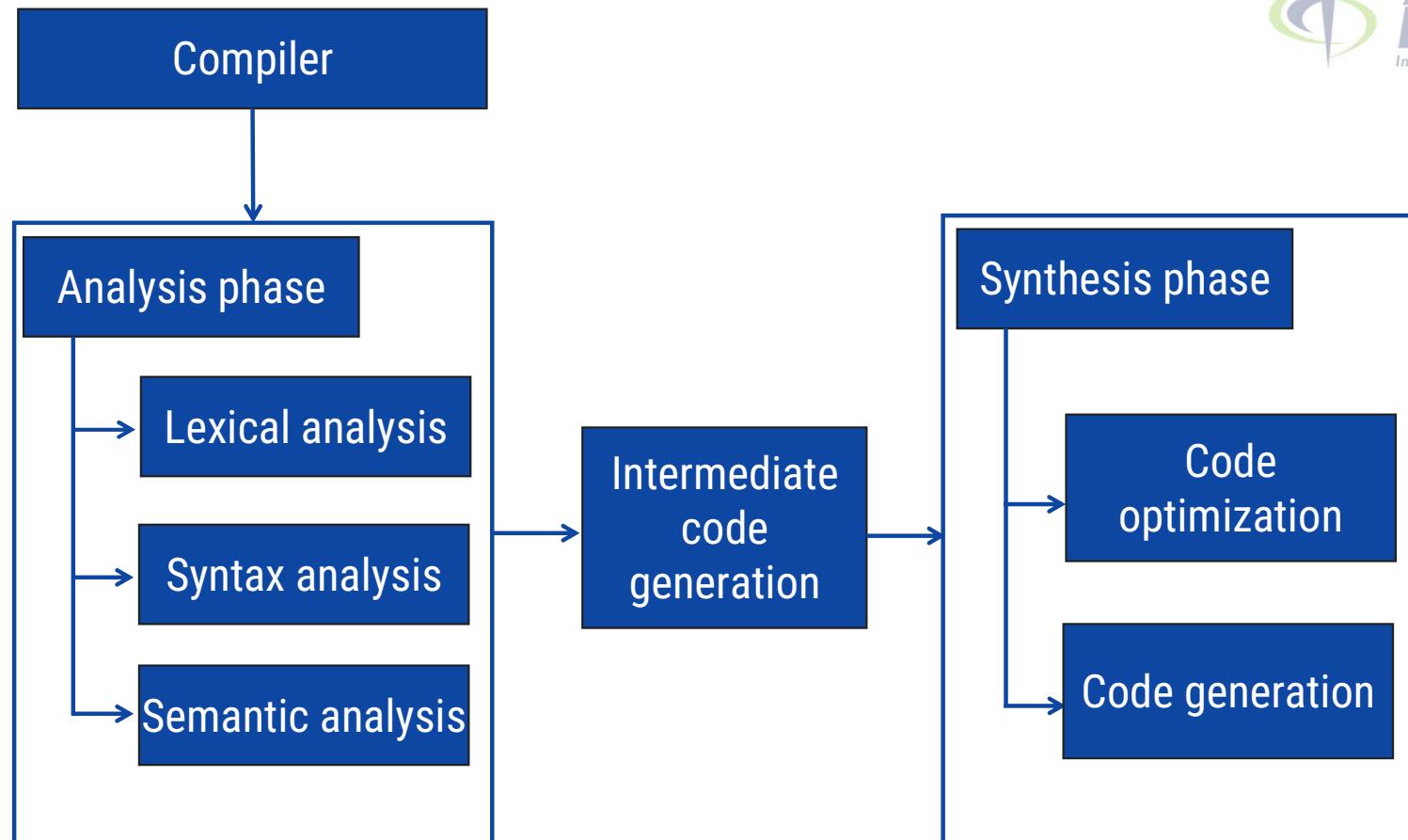
# Intermediate code generator

- ▶ Two important properties of intermediate code :
  1. It should be **easy to produce**.
  2. **Easy to translate** into target program.
- ▶ Intermediate form can be represented using "**three address code**".
- ▶ Three address code consist of a sequence of instruction, each of which has **at most three** operands.



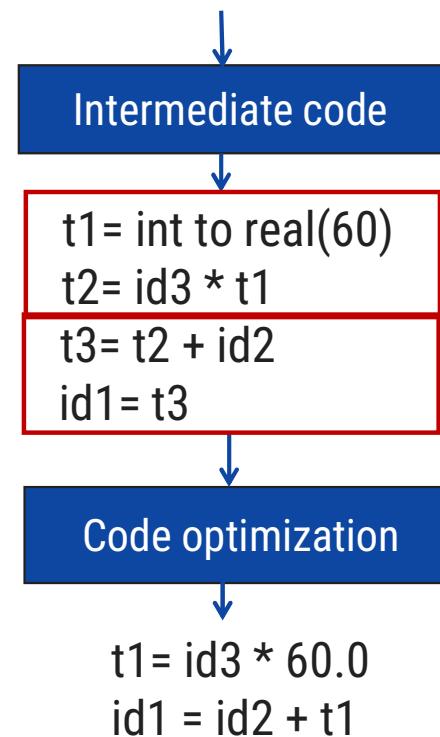
$t1 = \text{int to real}(60)$   
 $t2 = id3 * t1$   
 $t3 = t2 + id2$   
 $id1 = t3$

# Phases of compiler

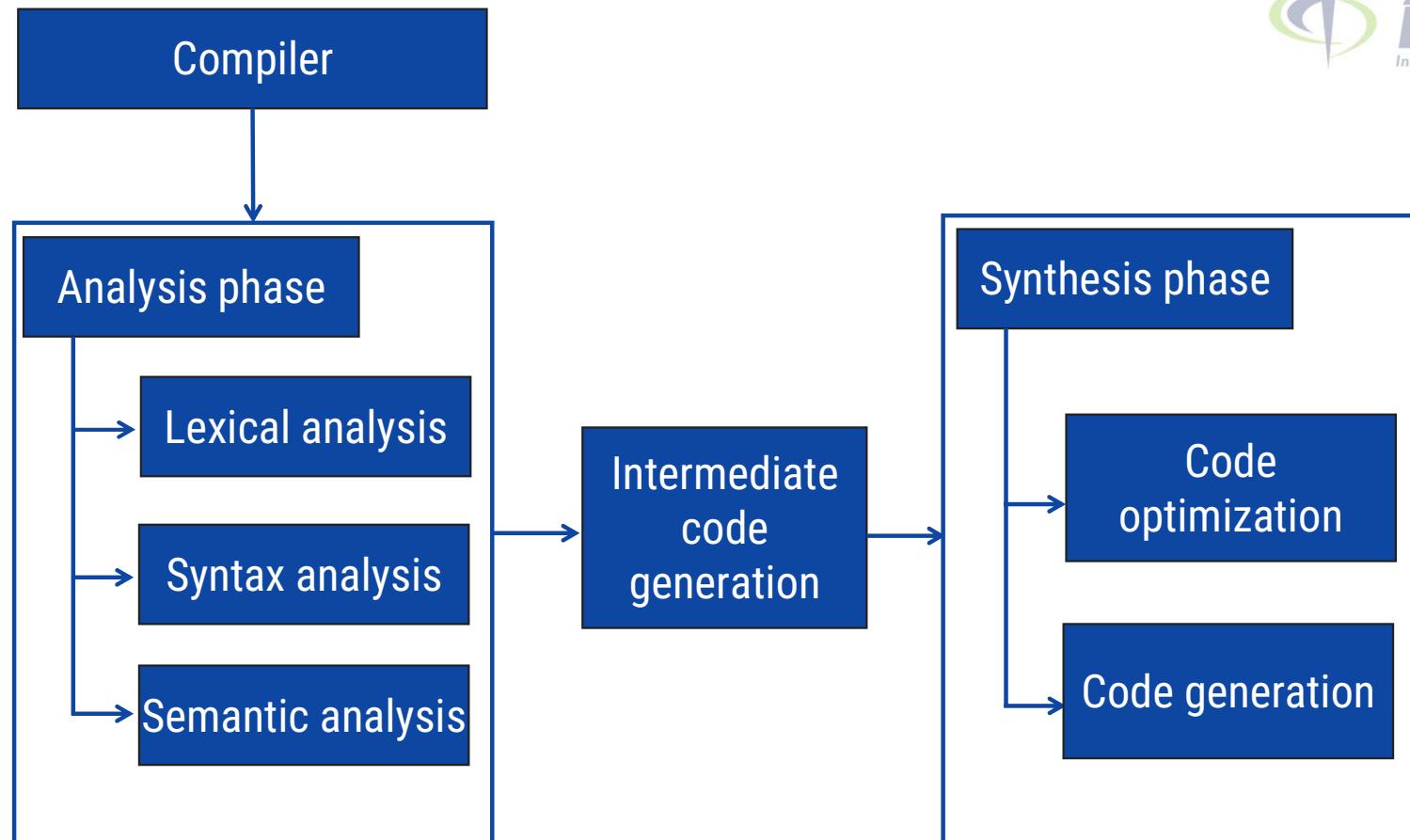


# Code optimization

- ▶ It **improves** the intermediate code.
- ▶ This is necessary to have a **faster execution** of code or **less consumption of memory**.

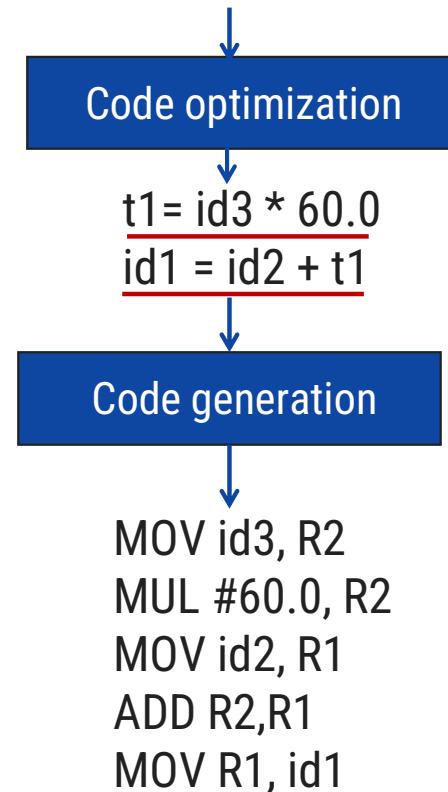


# Phases of compiler



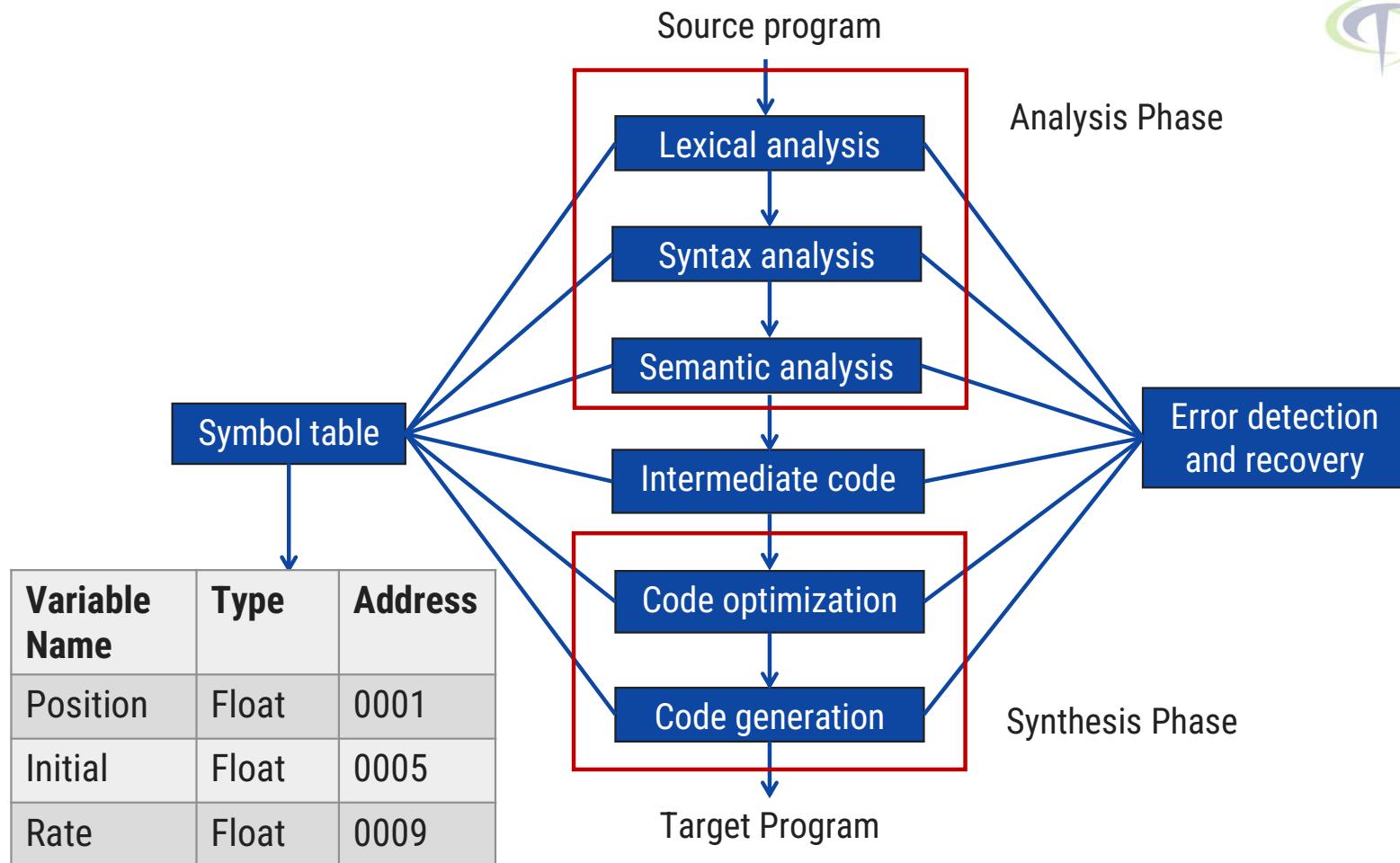
# Code generation

- ▶ The intermediate code instructions are **translated into sequence of machine instruction.**



**Id3→R2  
Id2→R1**

# Phases of compiler



# Exercise

► Write output of all the phases of compiler for following statements:

1.  $x = b - c * 2$
2.  $I = p * n * r / 100$

# Grouping of Phases

# Front end & back end (Grouping of phases)

## Front end

- ▶ Depends primarily on source language and largely independent of the target machine.
- ▶ It includes following phases:
  1. Lexical analysis
  2. Syntax analysis
  3. Semantic analysis
  4. Intermediate code generation
  5. Creation of symbol table & Error handling

## Back end

- ▶ Depends on target machine and do not depends on source program.
- ▶ It includes following phases:
  1. Code optimization
  2. Code generation phase
  3. Error handling and symbol table operation

# Difference between compiler & interpreter

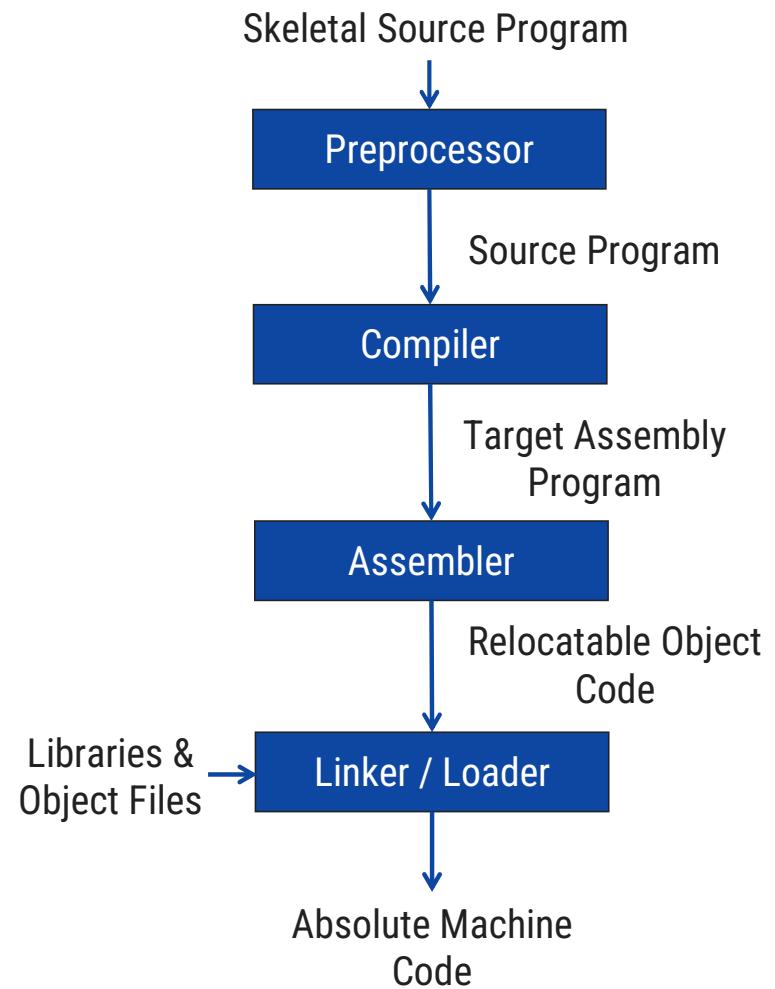
Compiler	Interpreter
Scans the <b>entire program</b> and translates it as a whole into machine code.	It translates program's <b>one statement at a time</b> .
It <b>generates</b> intermediate code.	It <b>does not generate</b> intermediate code.
An error is displayed after <b>entire program</b> is checked.	An error is displayed for <b>every instruction</b> interpreted if any.
Memory requirement is <b>more</b> .	Memory requirement is <b>less</b> .
Example: C compiler	Example: Basic, Python, Ruby

# **Context of Compiler (Cousins of compiler)**

# Context of compiler (Cousins of compiler)

- ▶ In addition to compiler, many other system programs are required to generate absolute machine code.
- ▶ These system programs are:

- Preprocessor
- Assembler
- Linker
- Loader

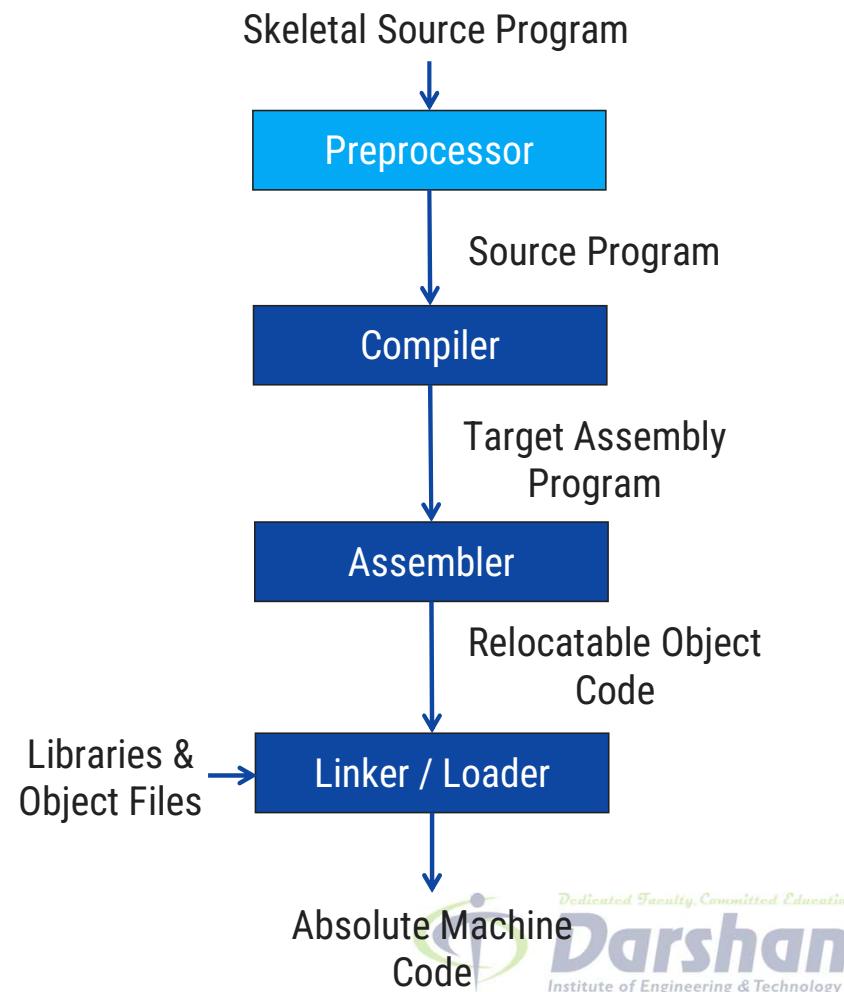


# Context of compiler (Cousins of compiler)

## Preprocessor

### ► Some of the task performed by preprocessor:

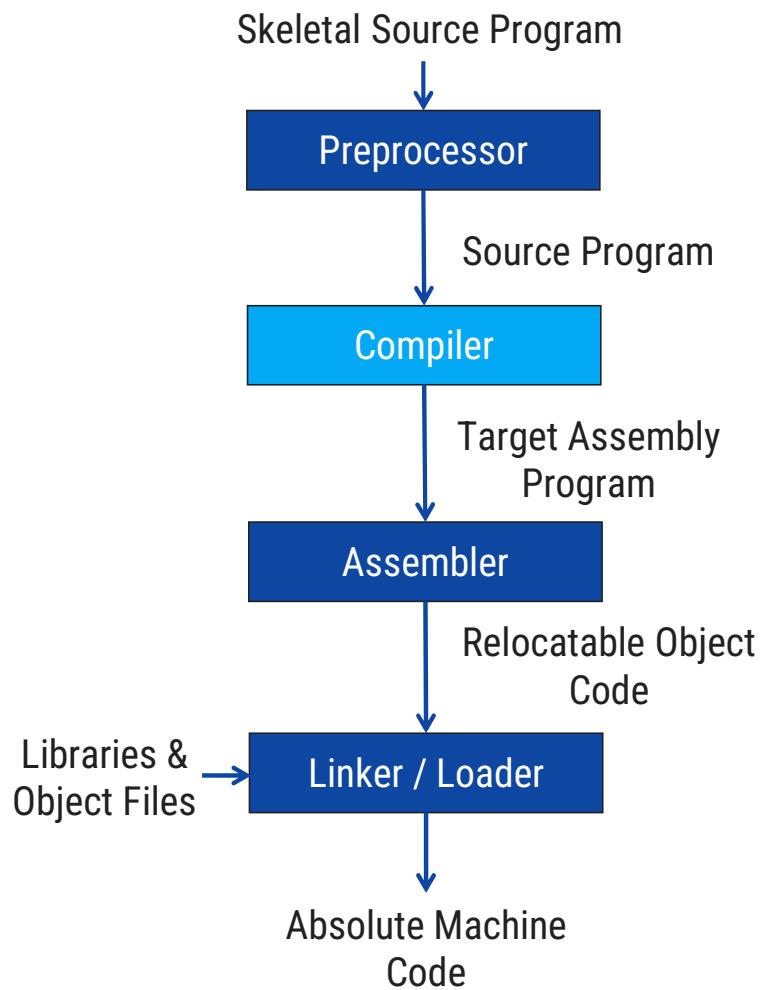
1. **Macro processing:** Allows user to define macros. Ex: `#define PI 3.14159265358979323846`
2. **File inclusion:** A preprocessor may include the header file into the program. Ex: `#include<stdio.h>`
3. **Rational preprocessor:** It provides built in macro for construct like `while` statement or `if` statement.
4. **Language extensions:** Add capabilities to the language by using built-in macros.
  - Ex: the language equal is a database query language embedded in C. Statement beginning with `##` are taken by preprocessor to be database access statement unrelated to C and translated into procedure call on routines that perform the database access.



# Context of compiler (Cousins of compiler)

## Compiler

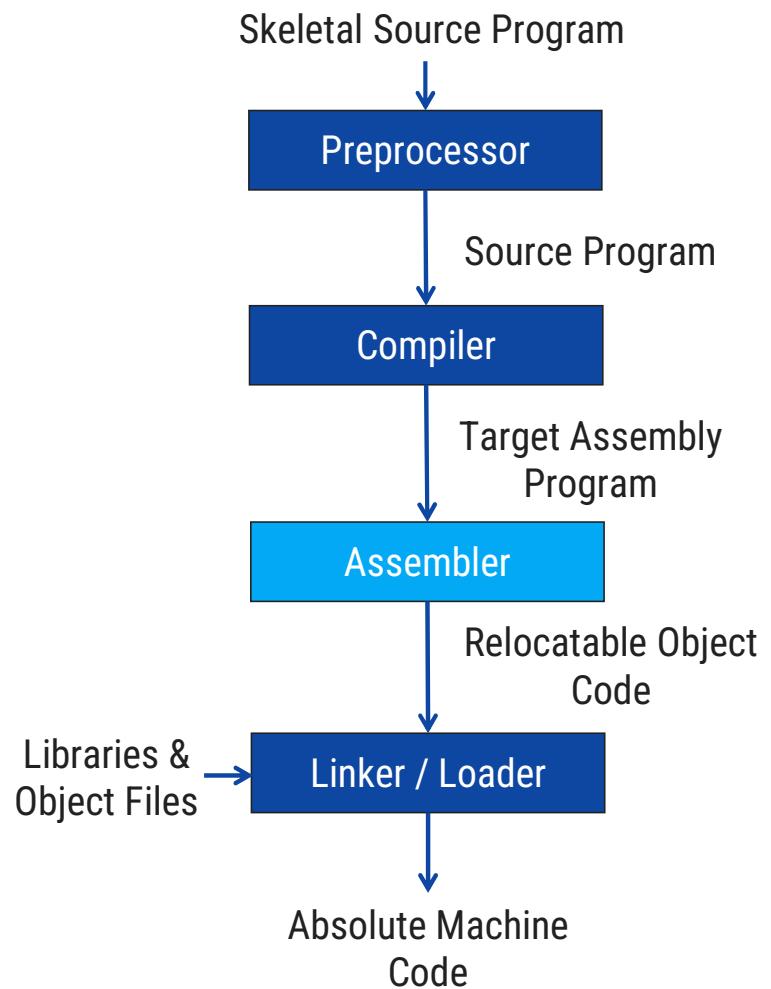
- ▶ A compiler is a program that reads a program written in source language and translates it into an equivalent program in target language.



# Context of compiler (Cousins of compiler)

## Assembler

- Assembler is a translator which takes the assembly program (mnemonic) as an input and generates the machine code as an output.



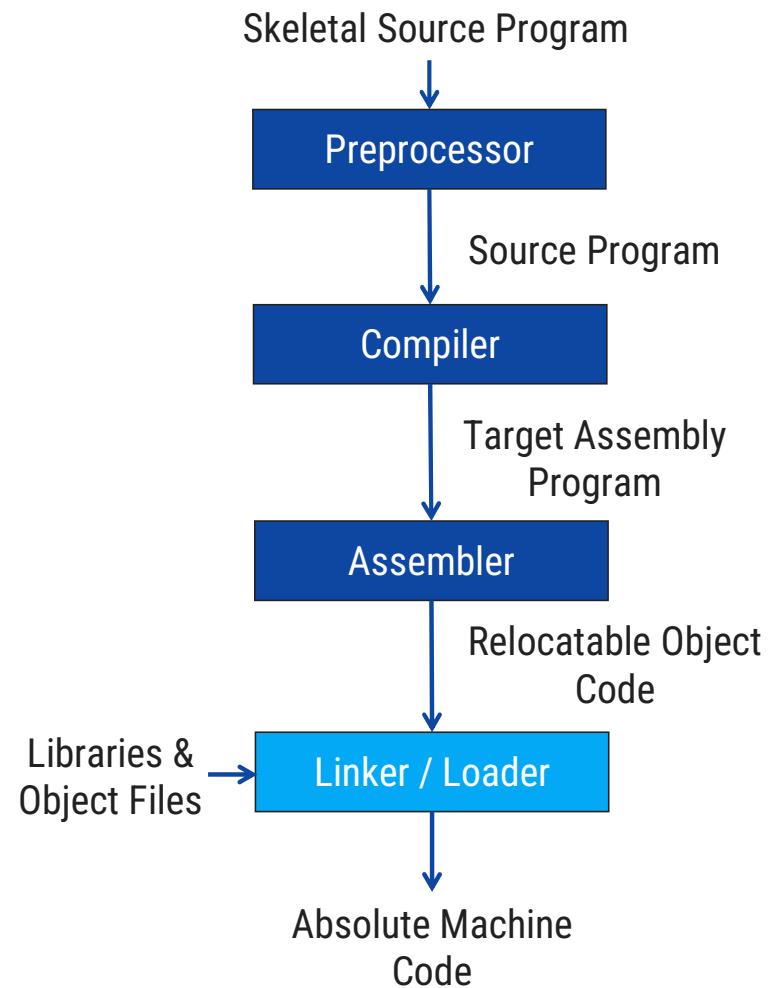
# Context of compiler (Cousins of compiler)

## Linker

- ▶ Linker makes a single program from a several files of relocatable machine code.
- ▶ These files may have been the result of several different compilation, and one or more library files.

## Loader

- ▶ The process of loading consists of:
  - Taking relocatable machine code
  - Altering the relocatable address
  - Placing the altered instructions and data in memory at the proper location.



# Pass structure

# Pass structure

- ▶ One complete scan of a source program is called pass.
- ▶ Pass includes **reading an input file** and **writing to the output** file.
- ▶ In a single pass compiler analysis of source statement is immediately followed by synthesis of equivalent target statement.
- ▶ While in a two pass compiler intermediate code is generated between analysis and synthesis phase.
- ▶ It is difficult to compile the source program into single pass due to: **forward reference**

# Pass structure

**Forward reference:** A forward reference of a program entity is a **reference to the entity which precedes its definition** in the program.

- ▶ This problem can be solved by postponing the generation of target code until more information concerning the entity becomes available.
- ▶ It leads to multi pass model of compilation.

## Pass I:

---

- ▶ Perform analysis of the source program and note relevant information.

## Pass II:

---

- ▶ In Pass II: Generate target code using information noted in pass I.

# Effect of reducing the number of passes

- ▶ It is desirable to have a few passes, because **it takes time to read and write** intermediate file.
- ▶ If we group **several phases into one pass** then **memory requirement** may be **large**.

# Types of compiler

# Types of compiler

## 1. One pass compiler

→ It is a type of compiler that compiles whole process in one-pass.

## 2. Two pass compiler

→ It is a type of compiler that compiles whole process in two-pass.  
→ It generates intermediate code.

## 3. Incremental compiler

→ The compiler which compiles only the changed line from the source code and update the object code.

## 4. Native code compiler

→ The compiler used to compile a source code for a same type of platform only.

## 5. Cross compiler

→ The compiler used to compile a source code for a different kinds platform.

# Science of building Compilers

# Science of building Compilers

- ▶ The main job of compiler is to accept the **source program** and **convert** it into suitable **target program**.
- ▶ A compiler must accept all source programs that conform to the specification of the language; the set of source programs is infinite and any program can be very large, consisting of possibly millions of lines of code.
- ▶ Compiler study mainly focused on study of **how to design the correct mathematical model** and **choose correct algorithm**.
- ▶ In compiler design, term “Code Optimization” indicates the attempts made by a compiler to produce a code which is more efficient than a previous code.
- ▶ The code should be faster than any other code that performs the same task.
- ▶ The objectives to be fulfilled by the compiler optimization include:
  1. The meaning of the compiled program must be preserved.
  2. Optimization should improve programs performance.
  3. Time required for compilation should be reasonable.

# Science of building Compilers

- ▶ Just theory is not sufficient to build a compiler, People involved in the design of compiler should be able to formulate the right problem to solve.
- ▶ In, order to do this the first step in through understanding of the behavior of programs.

# References

## Books:

### 1. **Compilers Principles, Techniques and Tools, PEARSON Education (Second Edition)**

Authors: Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

### 2. **Compiler Design, PEARSON (for Gujarat Technological University)**

Authors: Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman

# Thank You



## Unit - 2

# Lexical Analyzer



**Prof. Dixita B. Kagathara**

Computer Engineering Department

Darshan Institute of Engineering & Technology, Rajkot

✉ dixita.kagathara@darshan.ac.in

📞 +91 - 97277 47317 (CE Department)

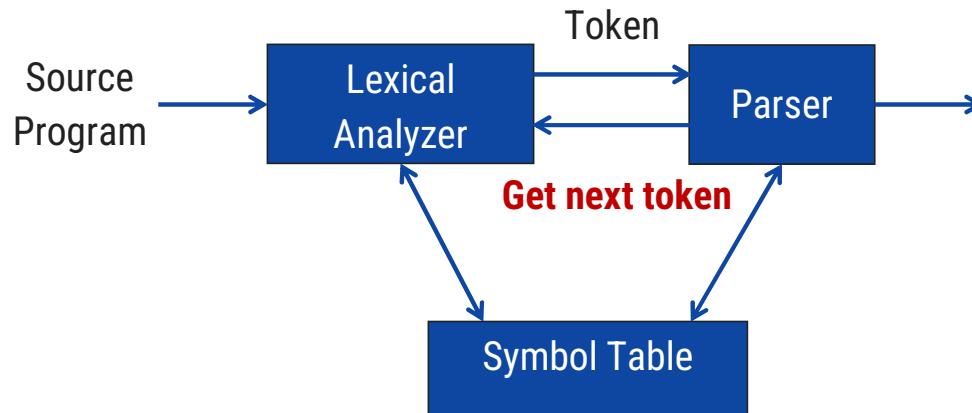
## Topics to be covered



- Interaction of scanner & parser
- Token, Pattern & Lexemes
- Input buffering
- Specification of tokens
- Regular expression & Regular definition
- Transition diagram
- Hard coding & automatic generation lexical analyzers
- Finite automata
- Regular expression to NFA using Thompson's rule
- Conversion from NFA to DFA using subset construction method
- DFA optimization
- Conversion from regular expression to DFA
- An Elementary Scanner Design & It's Implementation

# Interaction with Scanner & Parser

# Interaction of scanner & parser



- ▶ Upon receiving a “**Get next token**” command from parser, the lexical analyzer reads the input character until it can identify the next token.
- ▶ Lexical analyzer also stripping out comments and white space in the form of blanks, tabs, and newline characters from the source program.

# Why to separate lexical analysis & parsing?

1. Simplicity in **design**.
2. Improves compiler **efficiency**.
3. Enhance compiler **portability**.

# Token, Pattern & Lexemes

# Token, Pattern & Lexemes

## Token

Sequence of character having a collective meaning is known as **token**.

Categories of Tokens:

1. Identifier
2. Keyword
3. Operator
4. Special symbol
5. Constant

## Pattern

The **set of rules** called **pattern** associated with a token.

Example: “*non-empty sequence of digits*”,  
“*letter followed by letters and digits*”

## Lexemes

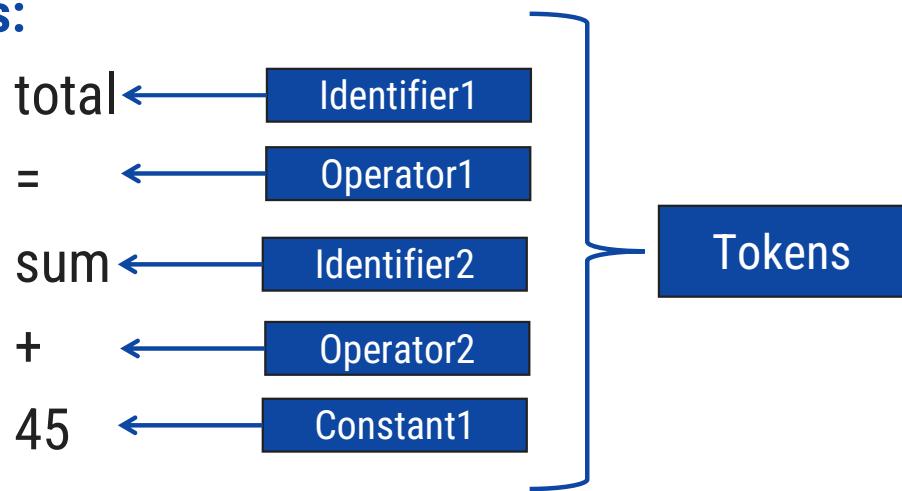
The **sequence of character** in a source program **matched with a pattern** for a **token** is called lexeme.

Example: Rate, DIET, count, Flag

# Example: Token, Pattern & Lexemes

Example: total = sum + 45

## Tokens:



## Lexemes

Lexemes of identifier: total, sum

Lexemes of operator: =, +

Lexemes of constant: 45

# Input buffering

# Input buffering

- ▶ There are mainly two techniques for input buffering:

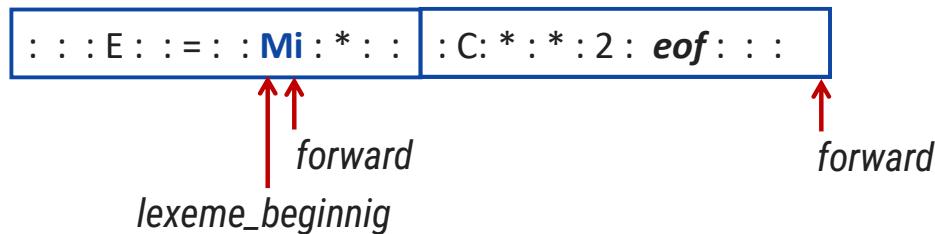
1. Buffer pairs
2. Sentinels

## Buffer Pair

- ▶ The lexical analysis scans the input string from left to right one character at a time.
- ▶ Buffer divided into two N-character halves, where N is the number of character on one disk block.

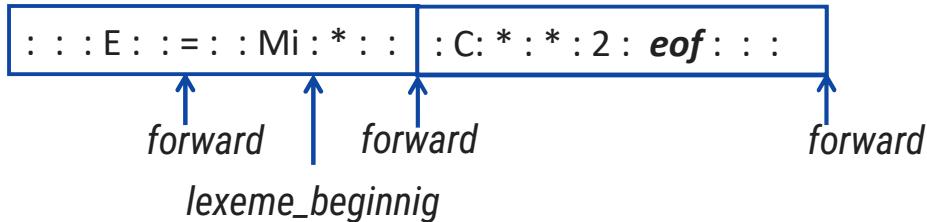
```
: : : E : : = : : Mi : * : : : C : * : * : 2 : eof : : :
```

# Buffer pairs



- ▶ Pointer *Lexeme Begin*, marks the beginning of the current lexeme.
- ▶ Pointer *Forward*, scans ahead until a pattern match is found.
- ▶ Once the next lexeme is determined, *forward* is set to character at its right end.
- ▶ Lexeme Begin is set to the character immediately after the lexeme just found.
- ▶ If forward pointer is at the end of first buffer half then second is filled with N input character.
- ▶ If forward pointer is at the end of second buffer half then first is filled with N input character.

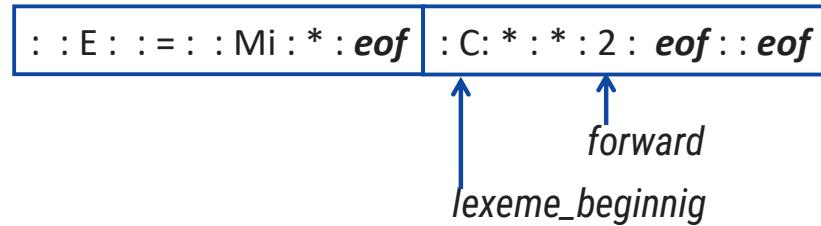
# Buffer pairs



## Code to advance forward pointer

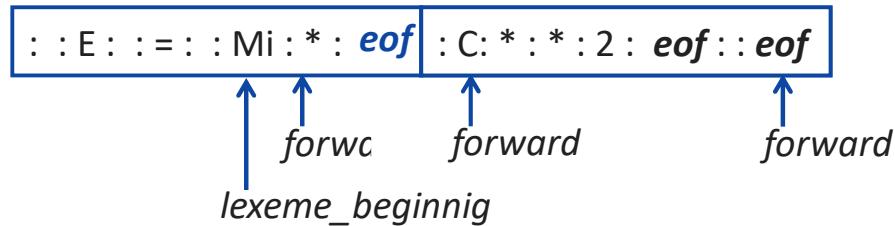
```
if forward at end of first half then begin  
    reload second half;  
    forward := forward + 1;  
end  
else if forward at end of second half then begin  
    reload first half;  
    move forward to beginning of first half;  
end  
else forward := forward + 1;
```

# Sentinels



- ▶ In buffer pairs we must check, each time we move the forward pointer that we have not moved off one of the buffers.
- ▶ Thus, for each character read, we make two tests.
- ▶ We can combine the buffer-end test with the test for the current character.
- ▶ We can reduce the two tests to one if we extend each buffer to hold a sentinel character at the end.
- ▶ The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **EOF**.

# Sentinels



*forward := forward + 1;*

***if forward = eof then begin***

***if forward at end of first half then begin***

*reload second half;*

*forward := forward + 1;*

***end***

***else if forward at the second half then begin***

*reload first half;*

*move forward to beginning of first half;*

***end***

***else terminate lexical analysis;***

***end***

# Specification of tokens

# Strings and languages

Term	Definition
<i>Prefix of s</i>	A string obtained by removing <b>zero or more trailing symbol</b> of string S. e.g., <b>ban</b> is prefix of <b>banana</b> .
<i>Suffix of S</i>	A string obtained by removing <b>zero or more leading symbol</b> of string S. e.g., <b>nana</b> is suffix of <b>banana</b> .
<i>Sub string of S</i>	A string obtained by <b>removing prefix and suffix</b> from S. e.g., <b>nan</b> is substring of <b>banana</b>
<i>Proper prefix, suffix and substring of S</i>	Any nonempty string x that is respectively proper prefix, suffix or substring of S, such that <b>s≠x</b> .
<i>Subsequence of S</i>	A string obtained by removing <b>zero or more not necessarily contiguous symbol</b> from S. e.g., <b>baaa</b> is subsequence of <b>banana</b> .

# Exercise

- ▶ Write prefix, suffix, substring, proper prefix, proper suffix and subsequence of following string:

String: **Compiler**

# Operations on languages

Operation	Definition
Union of L and M Written $L \cup M$	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M Written $LM$	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L Written $L^*$	$L^*$ denotes “zero or more concatenation of” L.
Positive closure of L Written $L^+$	$L^+$ denotes “one or more concatenation of” L.

# Regular Expression & Regular Definition

# Regular expression

- ▶ A regular expression is a sequence of characters that define a pattern.

## Notational shorthand's

1. One or more instances: +
2. Zero or more instances: \*
3. Zero or one instances: ?
4. Alphabets:  $\Sigma$

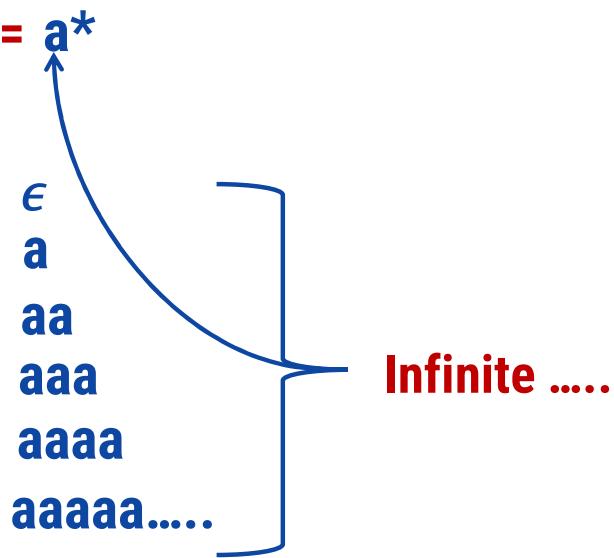
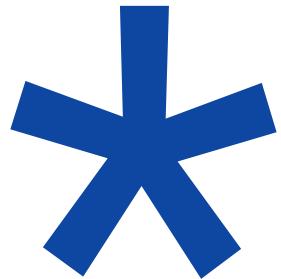
# Rules to define regular expression

1.  $\in$  is a regular expression that denotes  $\{\in\}$ , the set containing empty string.
2. If  $a$  is a symbol in  $\Sigma$  then  $a$  is a regular expression,  $L(a) = \{a\}$
3. Suppose  $r$  and  $s$  are regular expression denoting the languages  $L(r)$  and  $L(s)$ . Then,
  - a.  $(r)|(s)$  is a regular expression denoting  $L(r) \cup L(s)$
  - b.  $(r)(s)$  is a regular expression denoting  $L(r)L(s)$
  - c.  $(r)^*$  is a regular expression denoting  $(L(r))^*$
  - d.  $(r)$  is a regular expression denoting  $L((r))$

The language denoted by regular expression is said to be a **regular set**.

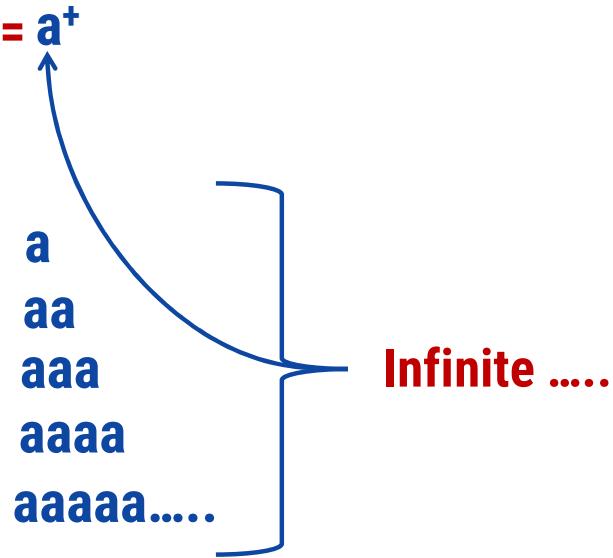
# Regular expression

- L = Zero or More Occurrences of a =  $a^*$



# Regular expression

- L = One or More Occurrences of a =  $a^+$



# Precedence and associativity of operators

Operator	Precedence	Associative
Kleene *	1	left
Concatenation	2	left
Union	3	left

# Regular expression examples

1. 0 or 1

Strings: 0, 1

R. E. = 0 | 1

2. 0 or 11 or 111

Strings: 0, 11, 111

R. E. = 0 | 11 | 111

3. String having zero or more a.

Strings:  $\epsilon$ , a, aa, aaa, aaaa .....

R. E. = a<sup>\*</sup>

4. String having one or more a.

Strings: a, aa, aaa, aaaa .....

R. E. = a<sup>+</sup>

5. Regular expression over  $\Sigma = \{a, b, c\}$  that represent all string of length 3.

Strings: abc, bca, bbb, cab, aba ....

R. E. = (a|b|c) (a|b|c) (a|b|c)

6. All binary string

Strings: 0, 11, 101, 10101, 1111 ...

R. E. = (0 | 1)<sup>†</sup>

# Regular expression examples

7. 0 or more occurrence of either a or b or both

*Strings:  $\epsilon, a, aa, abab, bab \dots$*

$$R.E. = (a | b)^*$$

8. 1 or more occurrence of either a or b or both

*Strings:  $a, aa, abab, bab, bbbaaa \dots$*

$$R.E. = (a | b)^+$$

9. Binary no. ends with 0

*Strings:  $0, 10, 100, 1010, 11110 \dots$*

$$R.E. = (0 | 1)^* 0$$

10. Binary no. ends with 1

*Strings:  $1, 101, 1001, 10101, \dots$*

$$R.E. = (0 | 1)^* 1$$

11. Binary no. starts and ends with 1

*Strings:  $11, 101, 1001, 10101, \dots$*

$$R.E. = 1 (0 | 1)^* 1$$

12. String starts and ends with same character

*Strings:  $00, 101, aba, baab \dots$*

$$R.E. = 1 (0 | 1)^* 1 \text{ or } 0 (0 | 1)^* 0 \\ a (a | b)^* a \text{ or } b (a | b)^* b$$

# Regular expression examples

13. All string of a and b starting with a

*Strings: a, ab, aab, abb...*

$$R.E. = a(a \mid b)^*$$

14. String of 0 and 1 ends with 00

*Strings: 00, 100, 000, 1000, 1100...*

$$R.E. = (0 \mid 1)^* 00$$

15. String ends with abb

*Strings: abb, babb, ababb...*

$$R.E. = (a \mid b)^* abb$$

16. String starts with 1 and ends with 0

*Strings: 10, 100, 110, 1000, 1100...*

$$R.E. = 1(0 \mid 1)^* 0$$

17. All binary string with at least 3 characters and 3<sup>rd</sup> character should be zero

*Strings: 000, 100, 1100, 1001...*

$$R.E. = (0|1)(0|1)0(0 \mid 1)^*$$

18. Language which consist of exactly two b's over the set  $\Sigma = \{a, b\}$

*Strings: bb, bab, aabb, abba...*

$$R.E. = a^* b a^* b a^*$$

# Regular expression examples

19. The language with  $\Sigma = \{a, b\}$  such that 3<sup>rd</sup> character from right end of the string is always a.

*Strings: aaa, aba, aaba, abb...*      *R.E. = (a | b) \* a(a|b)(a|b)*

20. Any no. of a followed by any no. of b followed by any no. of c

*Strings:  $\epsilon$ , abc, aabbcc, aabc, abb...*      *R.E. = a \* b \* c \**

21. String should contain at least three 1

*Strings: 111, 01101, 0101110....*      *R.E. = (0|1)\*1 (0|1)\*1 (0|1)\*1 (0|1)\**

22. String should contain exactly two 1

*Strings: 11, 0101, 1100, 010010, 100100....*      *R.E. = 0\*10\*10\**

23. Length of string should be at least 1 and at most 3

*Strings: 0, 1, 11, 01, 111, 010, 100....*      *R.E. = (0|1) | (0|1)(0|1) | (0|1)(0|1)(0|1)*

24. No. of zero should be multiple of 3

*Strings: 000, 010101, 110100, 000000, 100010010....*      *R.E. = (1\*01\*01\*01\*)\**

# Regular expression examples

25. The language with  $\Sigma = \{a, b, c\}$  where  $a$  should be multiple of 3

*Strings: aaa, baaa, bacaba, aaaaaa... R.E. = ((b|c)\*a(b|c)\*a(b|c)\*a(b|c)\*)\**

26. Even no. of 0

*Strings: 00, 0101, 0000, 100100.... R.E. = (1\*01\*01\*)\**

27. String should have odd length

*Strings: 0, 010, 110, 000, 10010.... R.E. = (0|1) ((0|1)(0|1))\**

28. String should have even length

*Strings: 00, 0101, 0000, 100100.... R.E. = ((0|1)(0|1))\**

29. String start with 0 and has odd length

*Strings: 0, 010, 010, 000, 00010.... R.E. = (0) ((0|1)(0|1))\**

30. String start with 1 and has even length

*Strings: 10, 1100, 1000, 100100.... R.E. = 1(0|1)((0|1)(0|1))\**

31. All string begins or ends with 00 or 11

*Strings: 00101, 10100, 110, 01011 ... R.E. = (00|11)(0 | 1) \* | (0|1) \* (00|11)*

# Regular expression examples

32. Language of all string containing both 11 and 00 as substring

*Strings:* 0011, 1100, 100110, 010011 ...      *R.E.* =  $((0|1)^*00(0|1)^*11(0|1)^*) \mid ((0|1)^*11(0|1)^*00(0|1)^*)$

33. String ending with 1 and not contain 00

*Strings:* 011, 1101, 1011 ....      *R.E.* =  $(1|01)^+$

34. Language of identifier

*Strings:* area, i, redious, grade1 ....      *R.E.* =  $(\_ + L)(\_ + L + D)^*$   
*where L is Letter & D is digit*

# Regular definition

- ▶ A regular definition gives names to certain regular expressions and uses those names in other regular expressions.
- ▶ Regular definition is a sequence of definitions of the form:

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

.....

$$d_n \rightarrow r_n$$

Where  $d_i$  is a **distinct name** &  $r_i$  is a **regular expression**.

- Example: Regular definition for identifier

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

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

**id**  $\rightarrow$  **letter** (**letter** | **digit**)\*

# Regular definition example

- ▶ Example: Unsigned Pascal numbers

3

5280

39.37

6.336E4

1.894E-4

2.56E+7

## Regular Definition

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

digits  $\rightarrow$  digit digit\*

optional\_fraction  $\rightarrow$  .digits |  $\epsilon$

optional\_exponent  $\rightarrow$  (E(+|-| $\epsilon$ )digits)| $\epsilon$

num  $\rightarrow$  digits optional\_fraction optional\_exponent

# Transition Diagram

# Transition Diagram

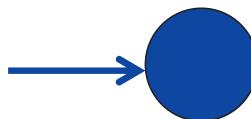
- ▶ A stylized flowchart is called transition diagram.



is a state



is a transition

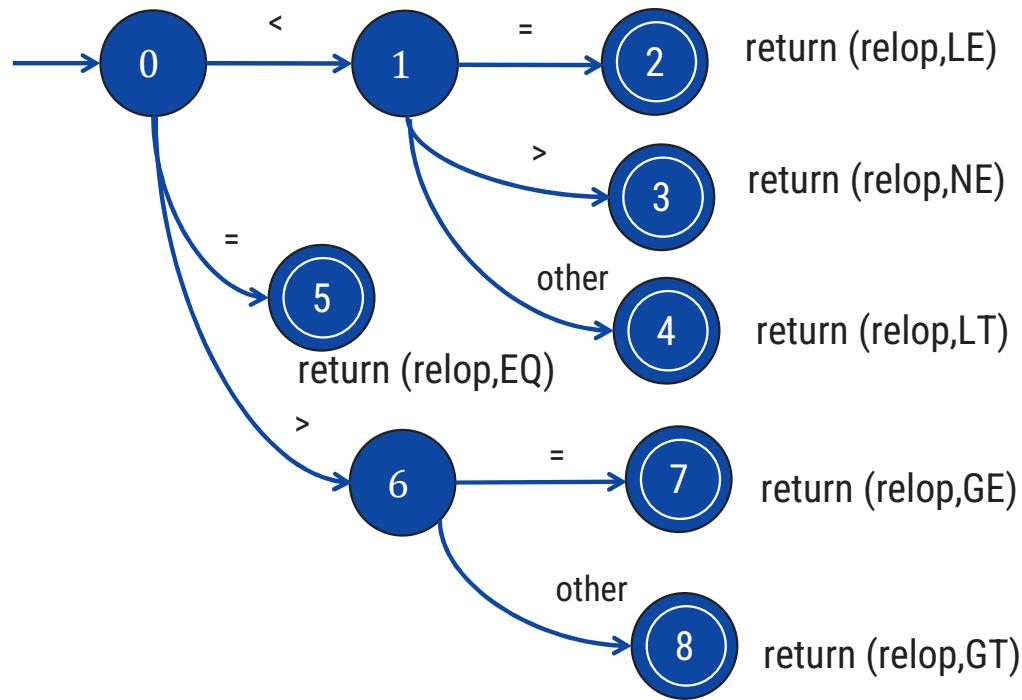


is a start state

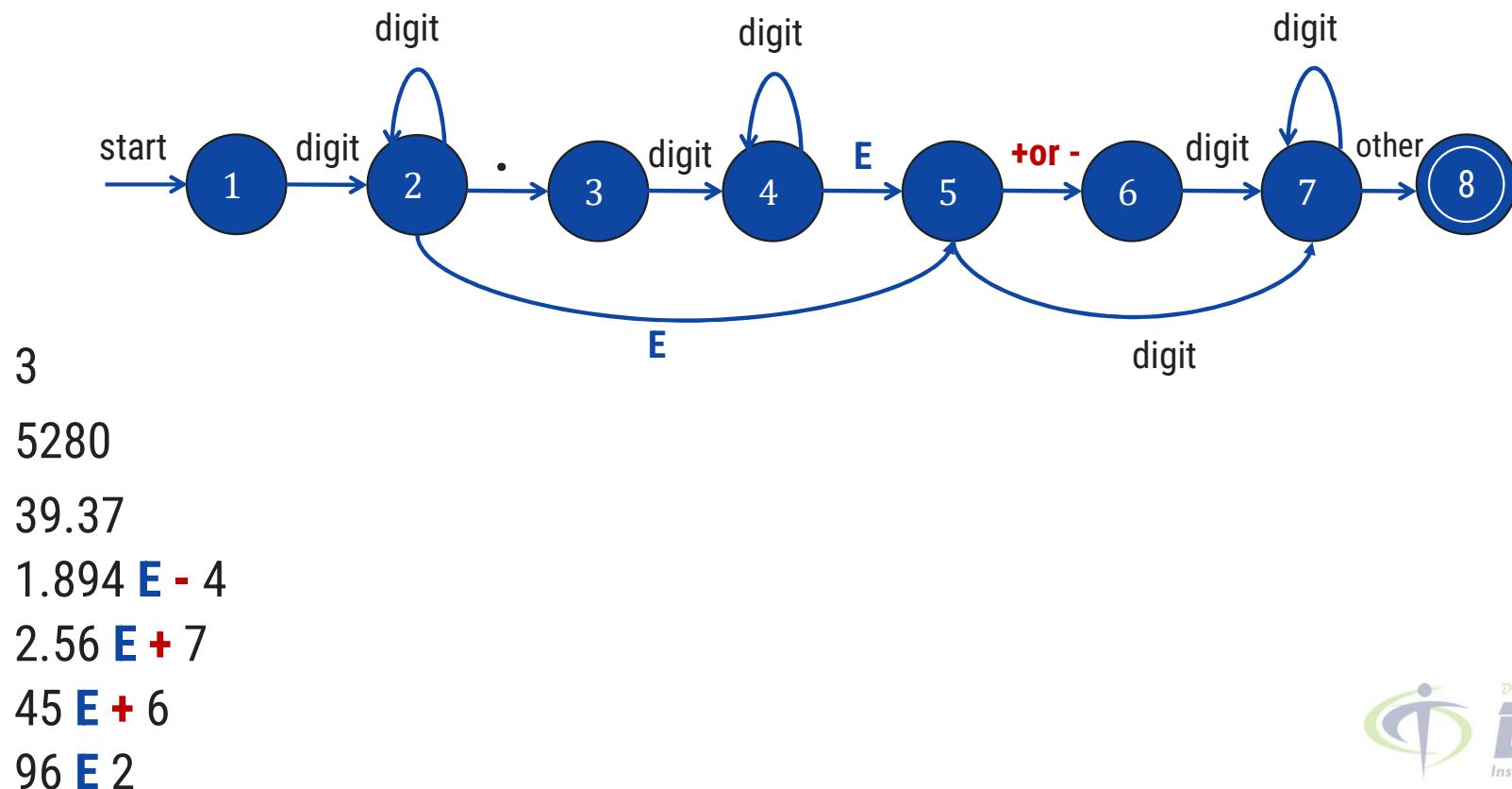


is a final state

# Transition Diagram : Relational operator



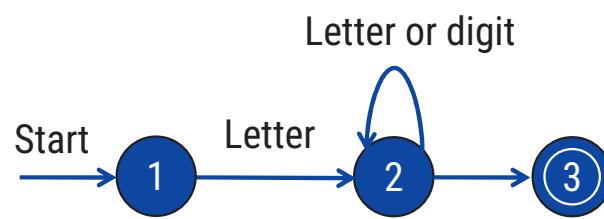
# Transition diagram : Unsigned number



# Hard coding & automatic generation Lexical analyzers

# Hard coding and automatic generation lexical analyzers

- ▶ Lexical analysis is about identifying the pattern from the input.
- ▶ To recognize the pattern, transition diagram is constructed.
- ▶ It is known as hard coding lexical analyzer.
- ▶ Example: to represent identifier in 'C', the first character must be letter and other characters are either letter or digits.
- ▶ To recognize this pattern, hard coding lexical analyzer will work with a transition diagram.
- ▶ The automatic generation lexical analyzer takes special notation as input.
- ▶ For example, lex compiler tool will take regular expression as input and finds out the pattern matching to that regular expression.



# Finite Automata

# Finite Automata

- ▶ Finite Automata are recognizers.
  - FA simply say “Yes” or “No” about each possible input string.
- ▶ Finite Automata is a mathematical model consist of:
  1. Set of states ***S***
  2. Set of input symbol ***Σ***
  3. A transition function ***move***
  4. Initial state ***S<sub>0</sub>***
  5. Final states or accepting states ***F***

# Types of finite automata

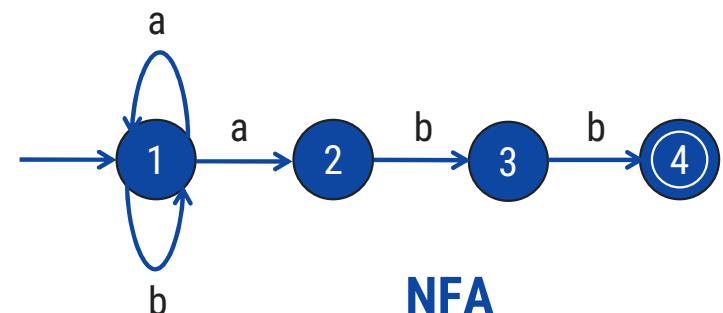
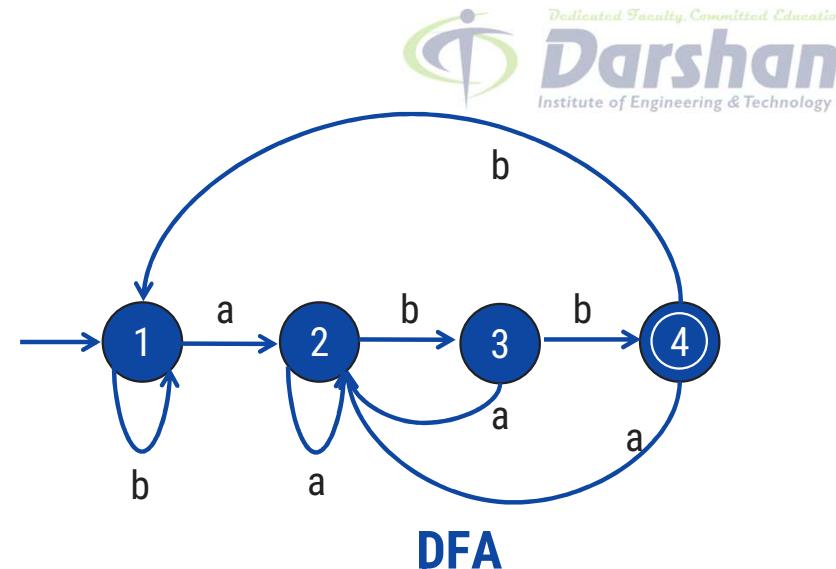
- ▶ Types of finite automata are:

DFA

- ▶ Deterministic finite automata (DFA): have for each state **exactly one edge** leaving out for each symbol.

NFA

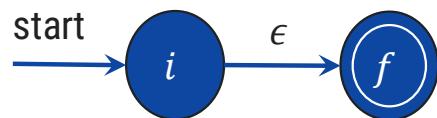
- ▶ Nondeterministic finite automata (NFA): There are **no restrictions** on the edges leaving a state. There can be **several with the same symbol as label** and some edges can be labeled with  $\epsilon$ .



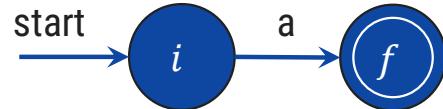
# Regular expression to NFA using Thompson's rule

# Regular expression to NFA using Thompson's rule

1. For  $\epsilon$ , construct the NFA



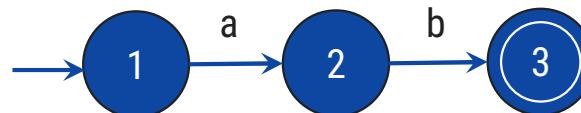
2. For  $a$  in  $\Sigma$ , construct the NFA



3. For regular expression  $st$

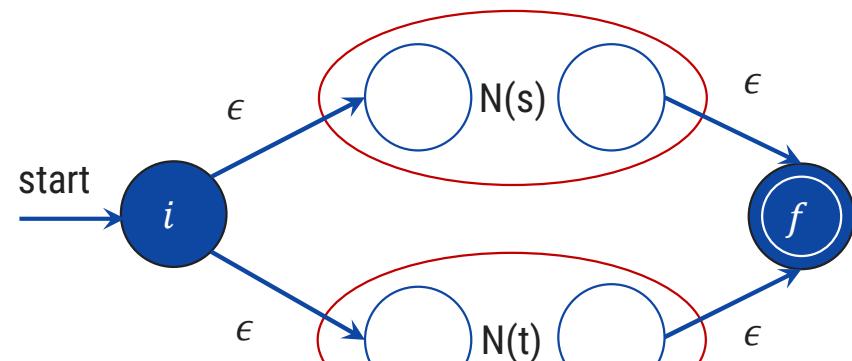


Ex: ab

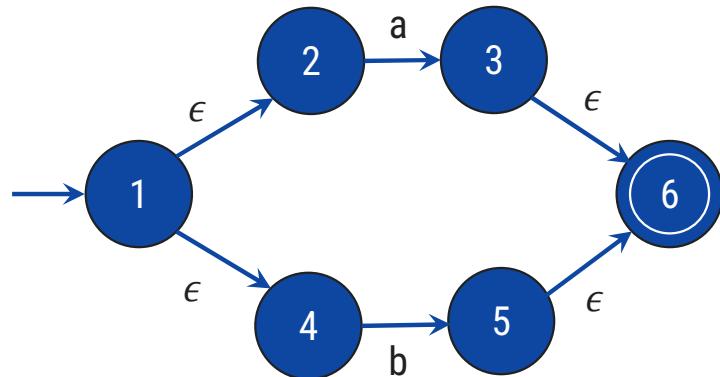


# Regular expression to NFA using Thompson's rule

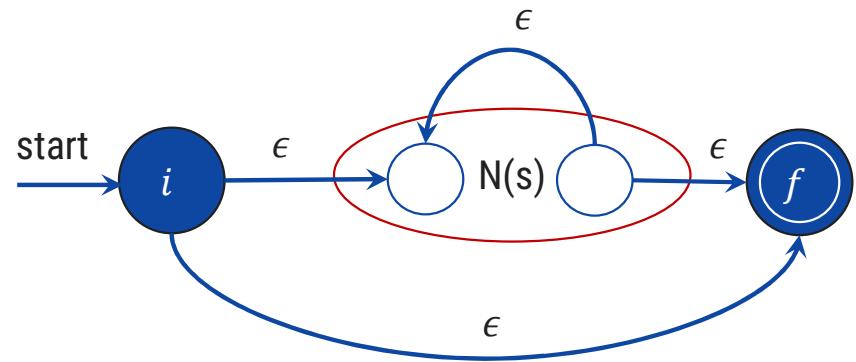
4. For regular expression  $s|t$



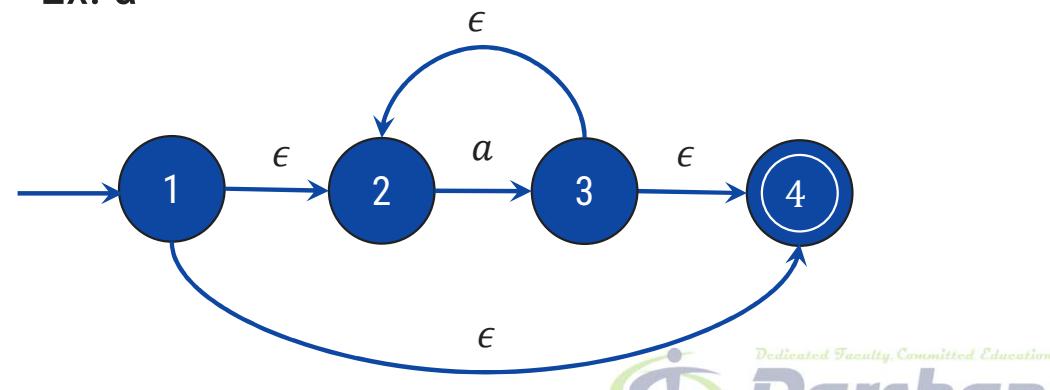
Ex:  $(a|b)$



5. For regular expression  $s^*$

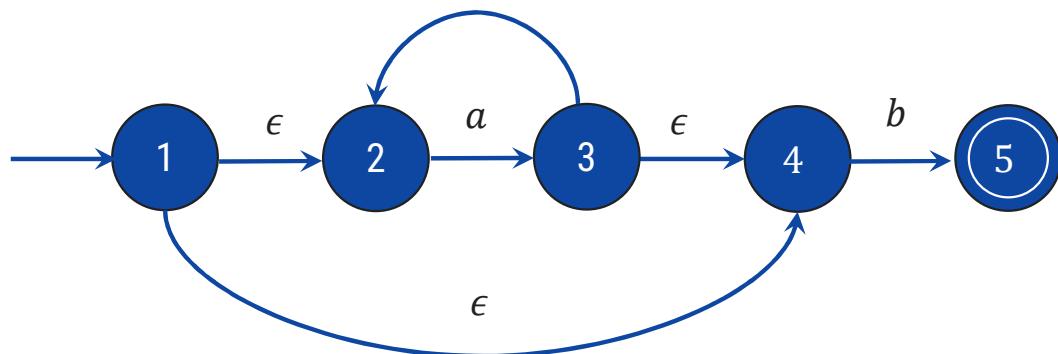


Ex:  $a^*$

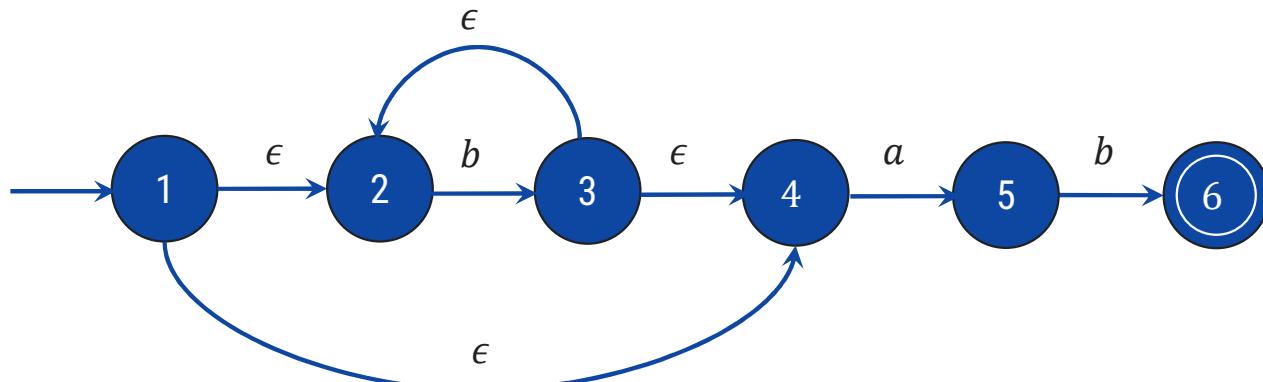


# Regular expression to NFA using Thompson's rule

►  $a^*b$



►  $b^*ab$



# Exercise

Convert following regular expression to NFA:

1. abba
2. bb(a)\*
3. (a|b)\*
4. a\* | b\*
5. a(a)\*ab
6. aa\*+ bb\*
7. (a+b)\*abb
8. 10(0+1)\*1
9. (a+b)\*a(a+b)
10. (0+1)\*010(0+1)\*
11. (010+00)\*(10)\*
12. 100(1)\*00(0+1)\*

# Conversion from NFA to DFA using subset construction method

# Subset construction algorithm

**Input:** An NFA  $N$ .

**Output:** A DFA  $D$  accepting the same language.

**Method:** Algorithm construct a transition table  $D_{tran}$  for  $D$ . We use the following operation:

OPERATION	DESCRIPTION
$\in - closure(s)$	Set of NFA states reachable from NFA state $s$ on $\in$ – transition alone.
$\in - closure(T)$	Set of NFA states reachable from some NFA state $s$ in $T$ on $\in$ – transition alone.
$Move (T, a)$	Set of NFA states to which there is a transition on input symbol $a$ from some NFA state $s$ in $T$ .

# Subset construction algorithm

initially  $\epsilon - \text{closure}(s_0)$  be the only state in  $Dstates$  and it is unmarked;

while there is unmarked states  $T$  in  $Dstates$  **do begin**

    mark  $T$ ;

        for each input symbol  $a$  **do begin**

$U = \epsilon - \text{closure}(\text{move}(T, a))$ ;

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

                add  $U$  as unmarked state to  $Dstates$ ;

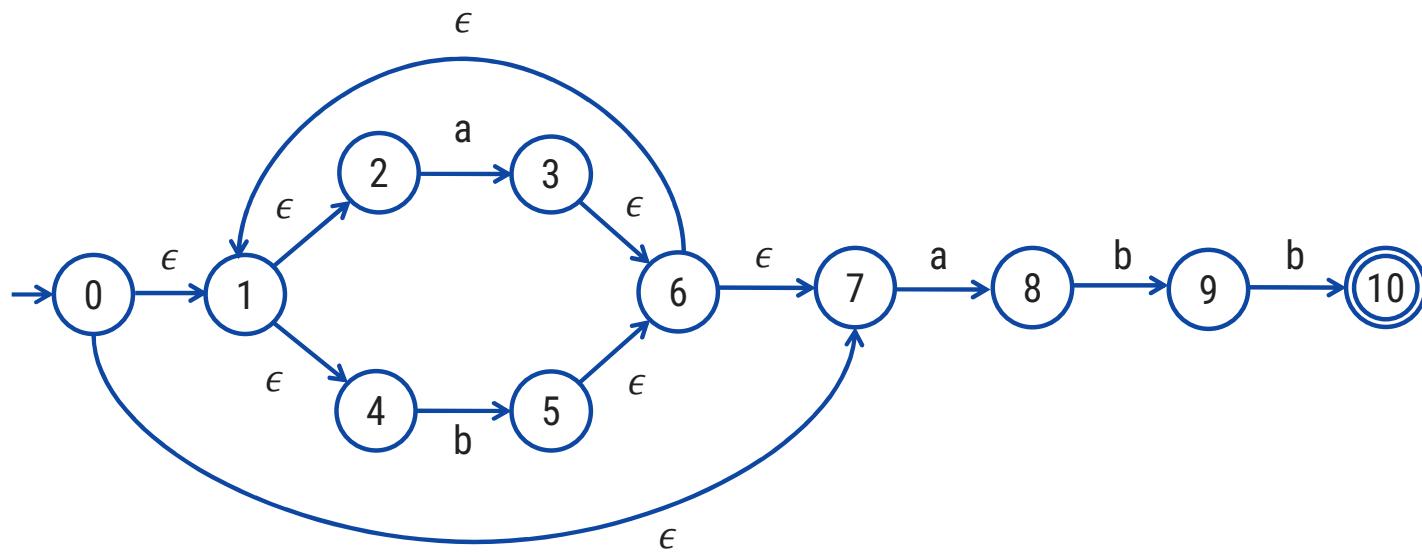
$Dtran[T, a] = U$

**end**

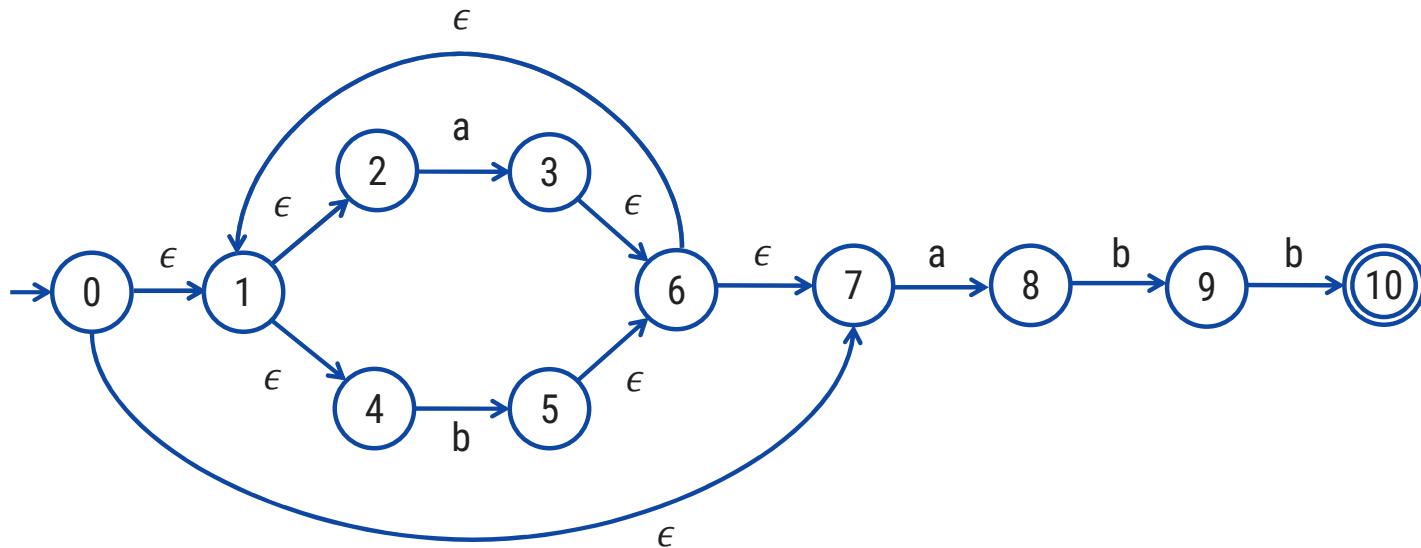
**end**

# Conversion from NFA to DFA

$(a|b)^*abb$



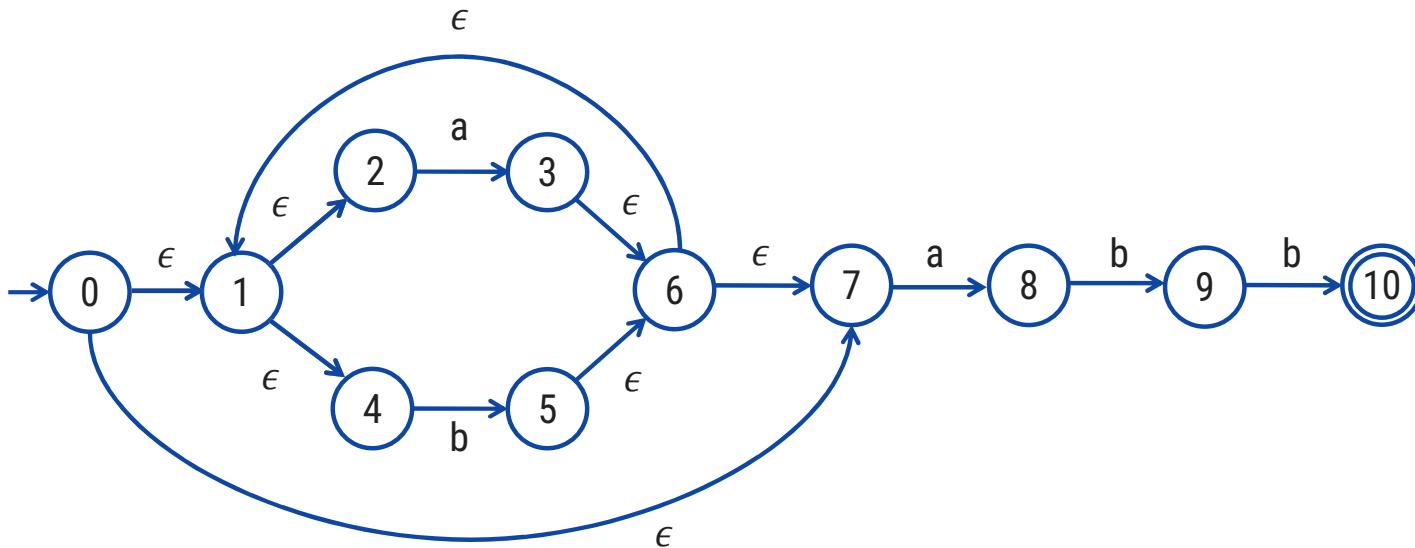
# Conversion from NFA to DFA



$\epsilon$ - Closure(0)=

$$= \{0,1,2,4,7\} \quad \text{---- A}$$

# Conversion from NFA to DFA



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

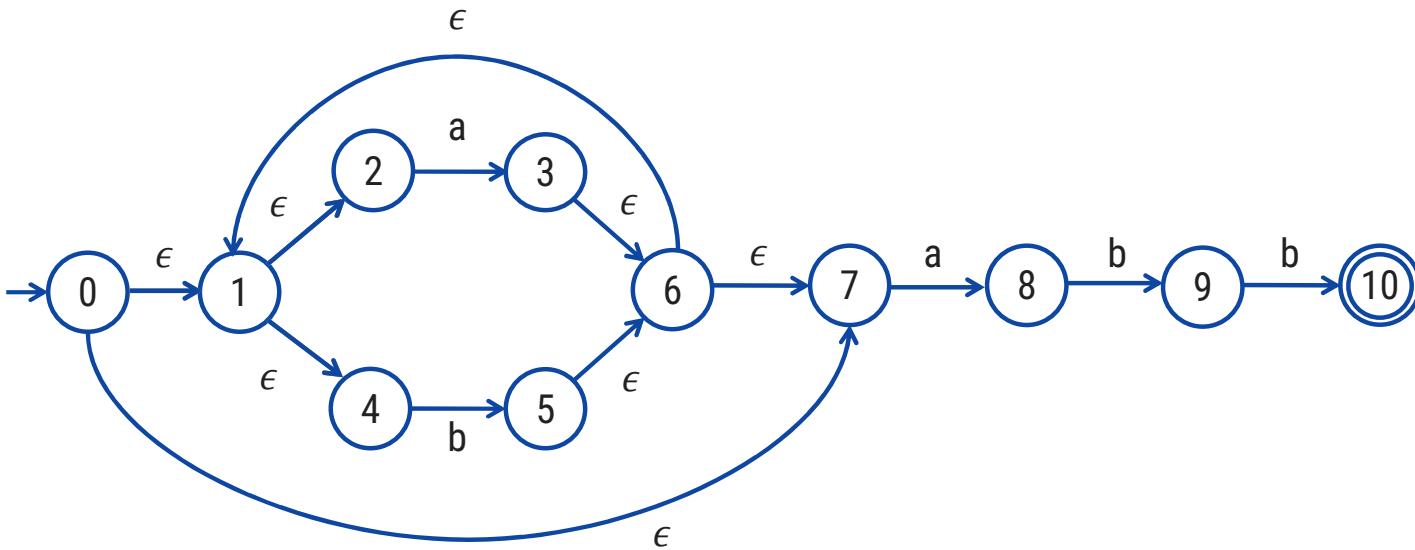
$$\text{Move}(A, a) = \{3, 8\}$$

$$\epsilon\text{-Closure}(\text{Move}(A, a))$$

$$= \{1, 2, 3, 4, 6, 7, 8\} \text{ ---- B}$$

States	a	b
$A = \{0, 1, 2, 4, 7\}$		
$B = \{1, 2, 3, 4, 6, 7, 8\}$		

# Conversion from NFA to DFA



**A = {0, 1, 2, 4, 7}**

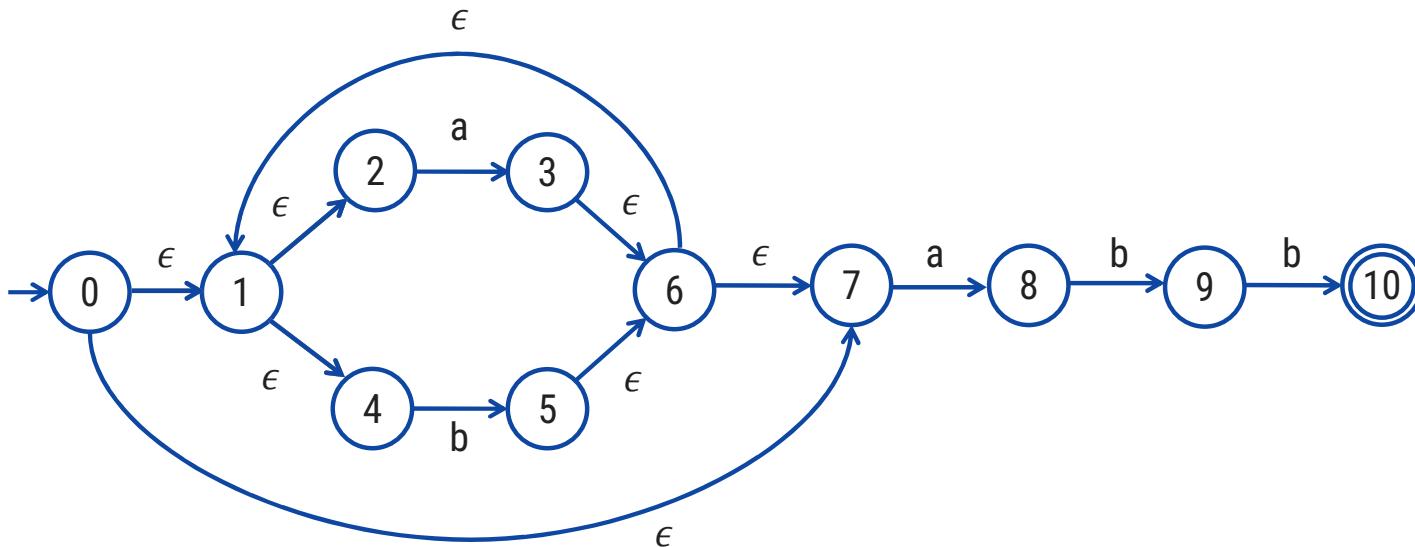
Move(A,b) =

$\epsilon$ - Closure(Move(A,b)) =

$$= \{1, 2, 4, 5, 6, 7\} \text{ ---- C}$$

States	a	b
A = {0,1,2,4,7}	B	
B = {1,2,3,4,6,7,8}		
C = {1,2,4,5,6,7}		

# Conversion from NFA to DFA



$$B = \{1, \underline{2}, 3, 4, \underline{6}, \underline{7}, 8\}$$

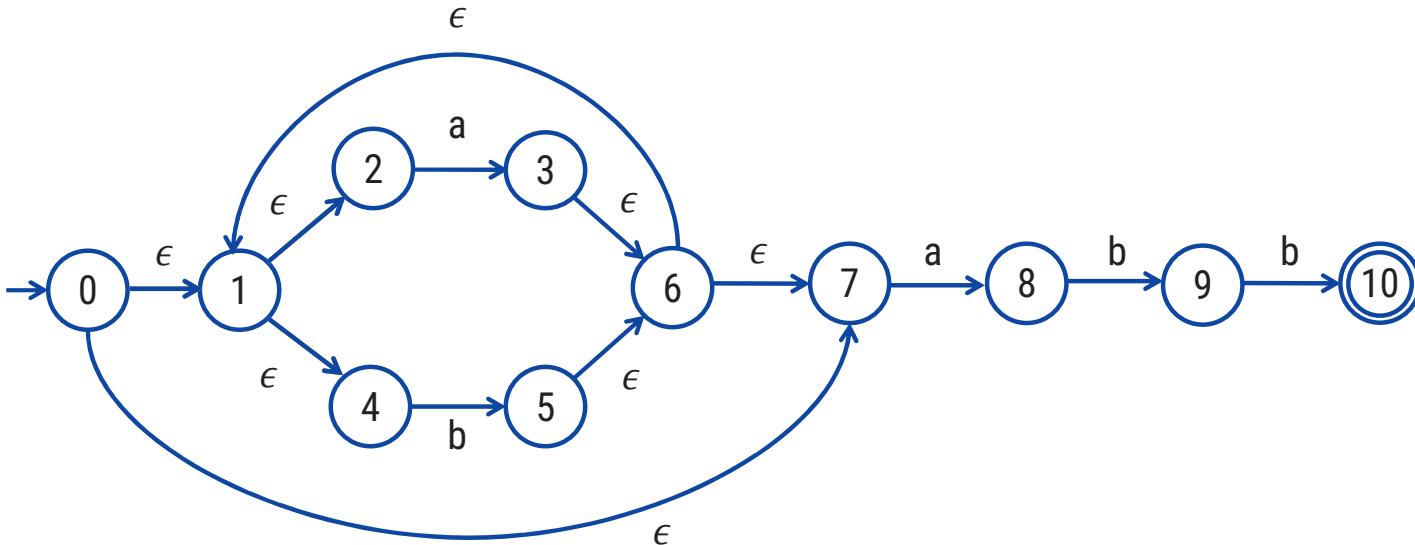
$$\text{Move}(B, a) = \{3, 8\}$$

$$\epsilon\text{-Closure}(\text{Move}(B, a))$$

$$= \{1, 2, 3, 4, 6, 7, 8\} \text{ ---- } B$$

States	a	b
A = {0,1,2,4,7}	B	C
B = {1,2,3,4,6,7,8}		
C = {1,2,4,5,6,7}		

# Conversion from NFA to DFA



$$B = \{1, 2, 3, \underline{4}, \underline{6}, 7, \underline{8}\}$$

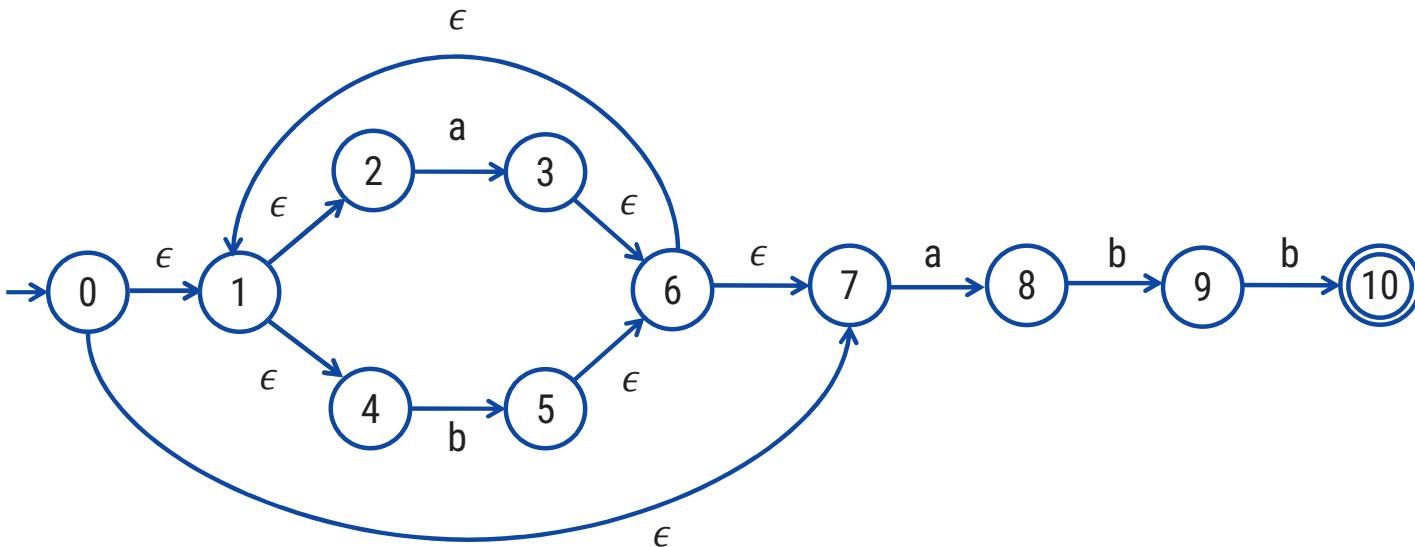
$$\text{Move}(B, b) = \{5, 9\}$$

$$\epsilon\text{-Closure}(\text{Move}(B, b))$$

$$= \{1, 2, 4, 5, 6, 7, 9\} \text{ ---- D}$$

States	a	b
A = {0,1,2,4,7}	B	C
B = {1,2,3,4,6,7,8}	B	
C = {1,2,4,5,6,7}		
D = {1,2,4,5,6,7,9}		

# Conversion from NFA to DFA



$$C = \{1, \underline{2}, 4, 5, \underline{6}, \underline{7}\}$$

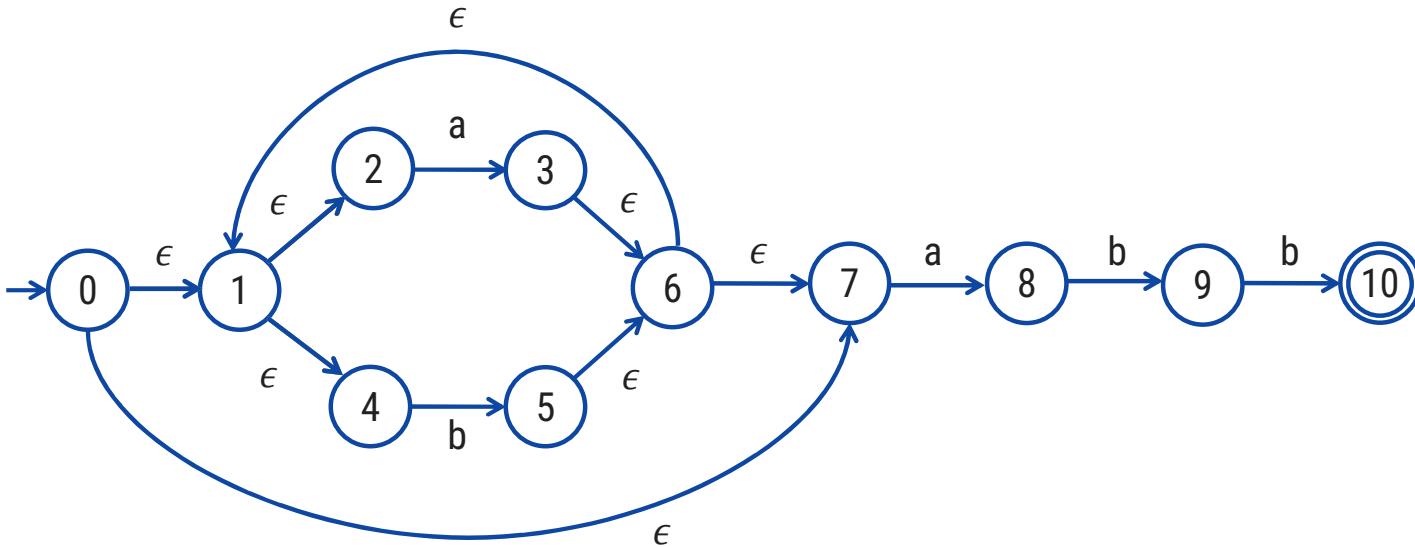
$$\text{Move}(C, a) = \{3, 8\}$$

$\epsilon$ - Closure( $\text{Move}(C, a)$ )

$$= \{1, 2, 3, 4, 6, 7, 8\} \text{ ---- B}$$

States	a	b
A = {0,1,2,4,7}	B	C
B = {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}		
D = {1,2,4,5,6,7,9}		

# Conversion from NFA to DFA



$$C = \{1, 2, \underline{4}, 5, 6, 7\}$$

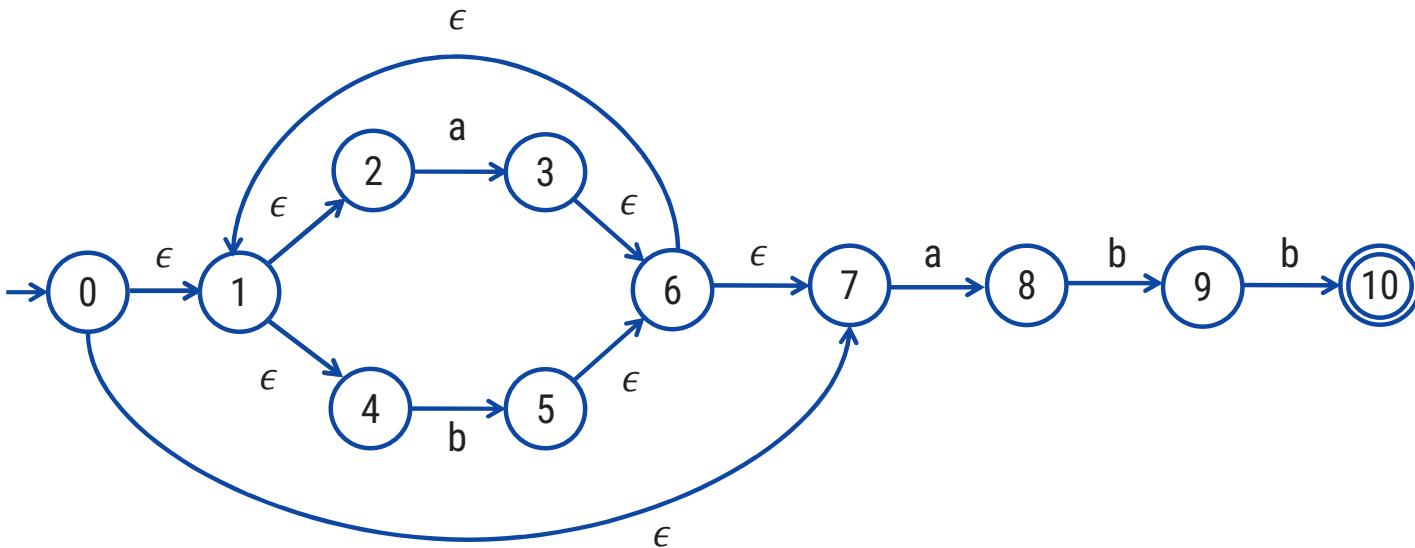
$\text{Move}(C, b) =$

$\epsilon\text{-Closure}(\text{Move}(C, b)) =$

$$= \{1, 2, 4, 5, 6, 7\} \text{ ---- } C$$

States	a	b
A = {0,1,2,4,7}	B	C
B = {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}	B	
D = {1,2,4,5,6,7,9}		

# Conversion from NFA to DFA



$$D = \{1, \underline{2}, 4, 5, \underline{6}, \underline{7}, 9\}$$

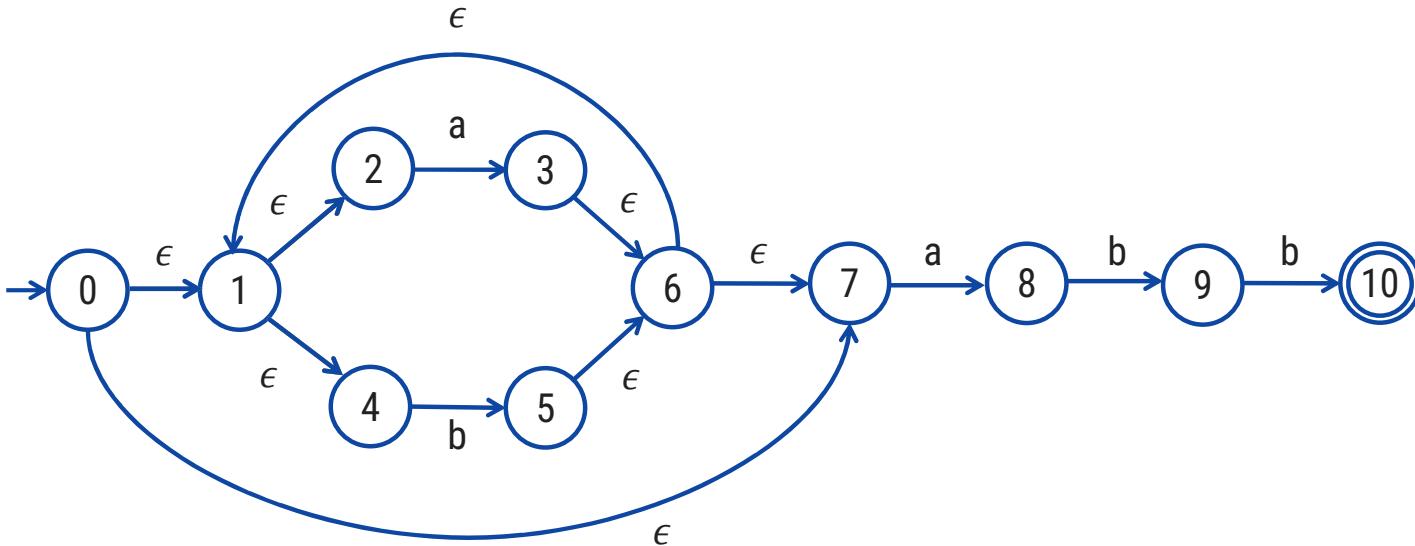
$$\text{Move}(D, a) = \{3, 8\}$$

$$\epsilon\text{-Closure}(\text{Move}(D, a))$$

$$= \{1, 2, 3, 4, 6, 7, 8\} \text{ ---- B}$$

States	a	b
A = {0,1,2,4,7}	B	C
B = {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}	B	C
D = {1,2,4,5,6,7,9}		

# Conversion from NFA to DFA



$$D = \{1, 2, \underline{4}, 5, 6, 7, \underline{9}\}$$

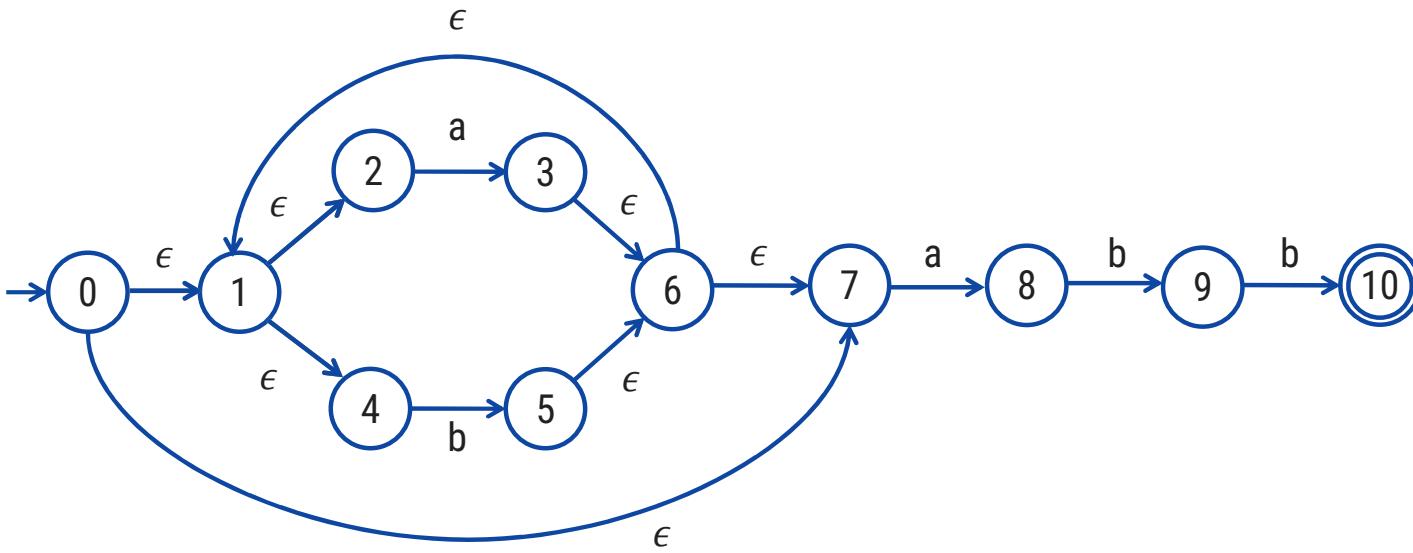
$$\text{Move}(D, b) = \{5, 10\}$$

$\epsilon$ - Closure( $\text{Move}(D, b)$ )

$$= \{1, 2, 4, 5, 6, 7, 10\} \text{ ---- E}$$

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

# Conversion from NFA to DFA



$$E = \{1, \underline{2}, 4, 5, \underline{6}, \underline{7}, 10\}$$

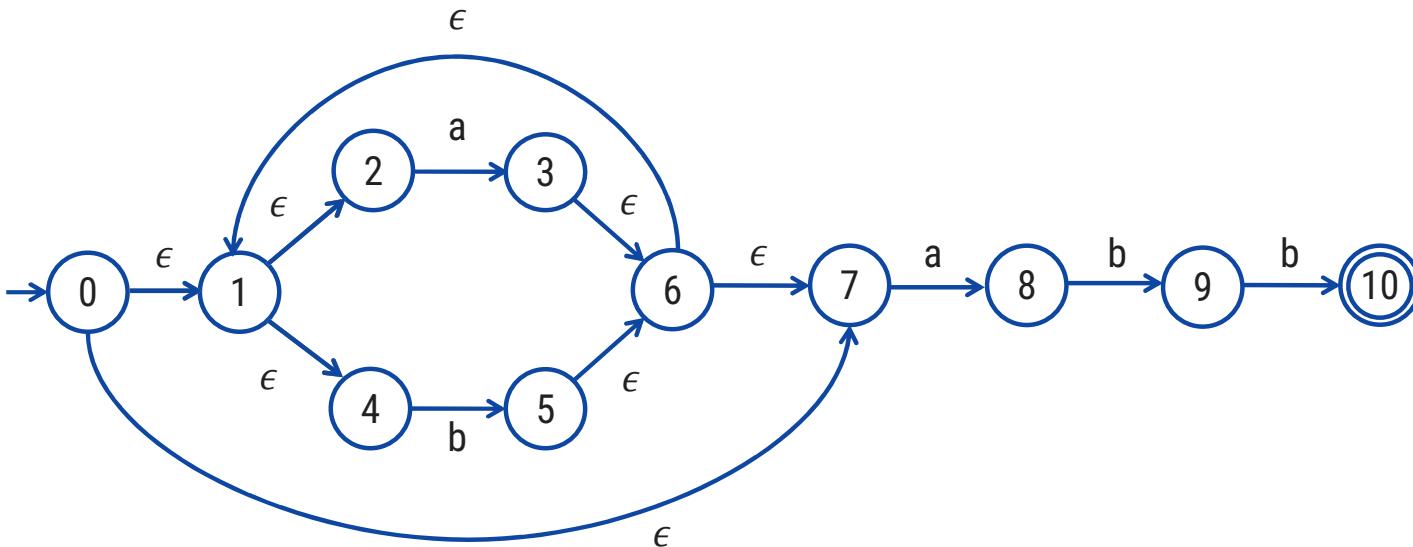
$$\text{Move}(E, a) = \{3, 8\}$$

$\epsilon$ - Closure( $\text{Move}(E, a)$ )

$$= \{1, 2, 3, 4, 6, 7, 8\} \text{ ---- B}$$

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

# Conversion from NFA to DFA



$$E = \{1, 2, \underline{4}, 5, 6, 7, 10\}$$

$\text{Move}(E, b) =$

$\epsilon\text{-Closure}(\text{Move}(E, b)) =$

$$= \{1, 2, 4, 5, 6, 7\} \text{ ---- C}$$

States	a	b
$A = \{0, 1, 2, 4, 7\}$	B	C
$B = \{1, 2, 3, 4, 6, 7, 8\}$	B	D
$C = \{1, 2, 4, 5, 6, 7\}$	B	C
$D = \{1, 2, 4, 5, 6, 7, 9\}$	B	E
$E = \{1, 2, 4, 5, 6, 7, 10\}$	B	

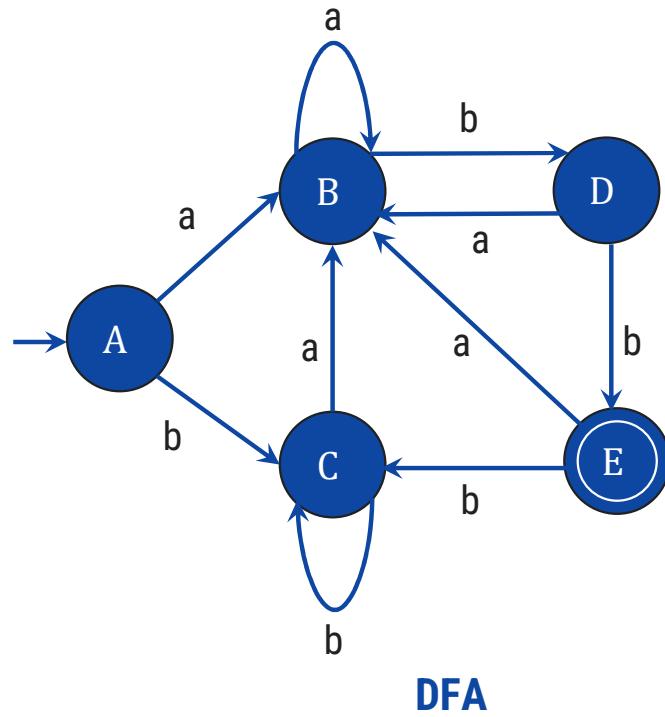
# Conversion from NFA to DFA

States	a	b
A = {0,1,2,4,7}	B	C
B = {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}	B	C
D = {1,2,4,5,6,7,9}	B	E
E = {1,2,4,5,6,7, <b>10</b> }	B	C

Transition Table

**Note:**

- Accepting state in NFA is 10
- 10 is element of E
- So, E is acceptance state in DFA



# Exercise

► Convert following regular expression to DFA using subset construction method:

1.  $(a+b)^*a(a+b)$
2.  $(a+b)^*ab^*a$

# DFA optimization

# DFA optimization

1. Construct an initial partition  $\Pi$  of the set of states with two groups: the accepting states  $F$  and the non-accepting states  $S - F$ .
2. Apply the repartition procedure to  $\Pi$  to construct a new partition  $\Pi_{new}$ .
3. If  $\Pi_{new} = \Pi$ , let  $\Pi_{final} = \Pi$  and continue with step (4). Otherwise, repeat step (2) with  $\Pi = \Pi_{new}$ .

**for** each group  $G$  of  $\Pi$  **do begin**

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

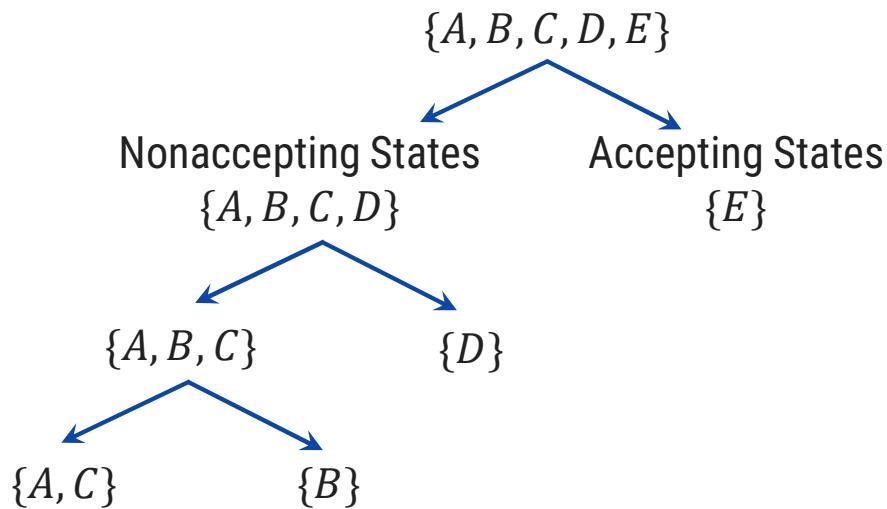
replace  $G$  in  $\Pi_{new}$  by the set of all subgroups formed.

**end**

# DFA optimization

4. Choose one state in each group of the partition  $\Pi_{final}$  as the representative for that group. The representatives will be the states of  $M'$ . Let  $s$  be a representative state, and suppose on input  $a$  there is a transition of  $M$  from  $s$  to  $t$ . Let  $r$  be the representative of  $t$ 's group. Then  $M'$  has a transition from  $s$  to  $r$  on  $a$ . Let the start state of  $M'$  be the representative of the group containing start state  $s_0$  of  $M$ , and let the accepting states of  $M'$  be the representatives that are in  $F$ .
5. If  $M'$  has a dead state  $d$ , then remove  $d$  from  $M'$ . Also remove any state not reachable from the start state.

# DFA optimization



- ▶ Now no more splitting is possible.
- ▶ If we chose A as the representative for group (AC), then we obtain reduced transition table

States	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

States	a	b
A	B	A
B	B	D
D	B	E
E	B	A

Optimized  
Transition Table

# Conversion from regular expression to DFA

# Rules to compute nullable, firstpos, lastpos

## ▶ nullable( $n$ )

- The subtree at node  $n$  generates languages including the empty string.

## ▶ firstpos( $n$ )

- The set of positions that can match the first symbol of a string generated by the subtree at node  $n$ .

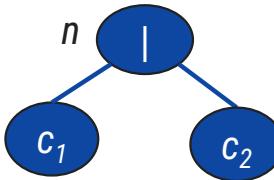
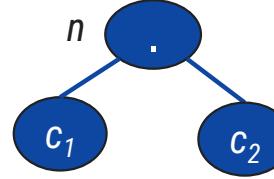
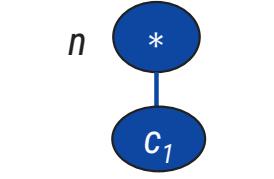
## ▶ lastpos( $n$ )

- The set of positions that can match the last symbol of a string generated by the subtree at node  $n$ .

## ▶ followpos( $i$ )

- The set of positions that can follow position  $i$  in the tree.

# Rules to compute nullable, firstpos, lastpos

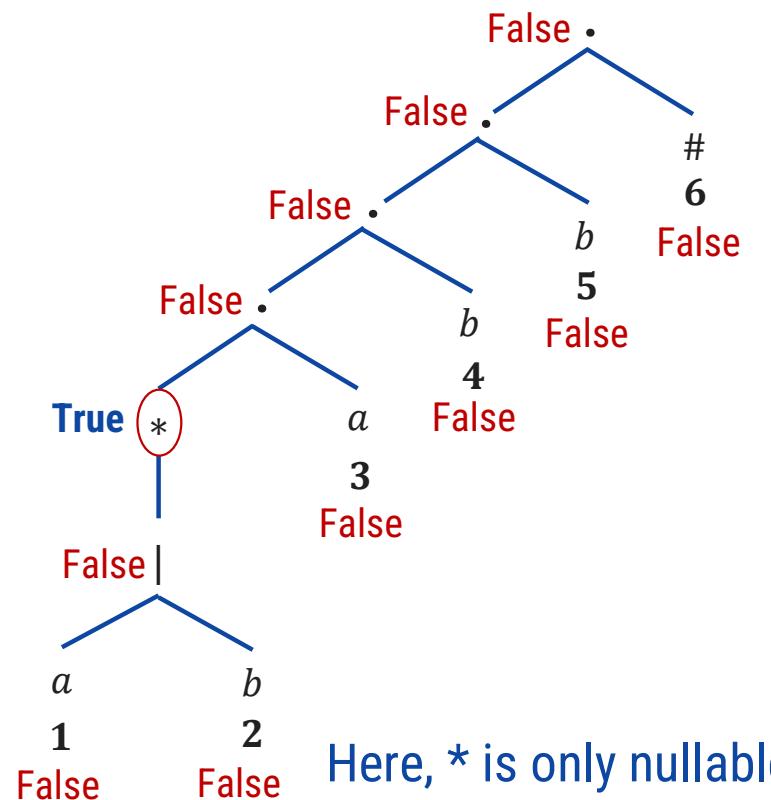
Node n	nullable(n)	firstpos(n)	lastpos(n)
A leaf labeled by $\epsilon$	<b>true</b>	$\emptyset$	$\emptyset$
A leaf with position i	<b>false</b>	{i}	{i}
 $n$   $c_1$ $c_2$	nullable( $c_1$ ) or nullable( $c_2$ )	$\text{firstpos}(c_1)$ $\cup$ $\text{firstpos}(c_2)$	$\text{lastpos}(c_1)$ $\cup$ $\text{lastpos}(c_2)$
 $n$ . $c_1$ $c_2$	nullable( $c_1$ ) and nullable( $c_2$ )	<b>if</b> (nullable( $c_1$ )) <b>then</b> firstpos( $c_1$ ) $\cup$ firstpos( $c_2$ ) <b>else</b> firstpos( $c_1$ )	<b>if</b> (nullable( $c_2$ )) <b>then</b> lastpos( $c_1$ ) $\cup$ lastpos( $c_2$ ) <b>else</b> lastpos( $c_2$ )
 $n$ * $c_1$	<b>true</b>	firstpos( $c_1$ )	lastpos( $c_1$ )

# Rules to compute followpos

1. If  $n$  is **concatenation** node with left child  $c_1$  and right child  $c_2$  and  $i$  is a position in  $\text{lastpos}(c_1)$ , then all position in  $\text{firstpos}(c_2)$  are in  $\text{followpos}(i)$
2. If  $n$  is **\*** node and  $i$  is position in  $\text{lastpos}(n)$ , then all position in  $\text{firstpos}(n)$  are in  $\text{followpos}(i)$

# Conversion from regular expression to DFA

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

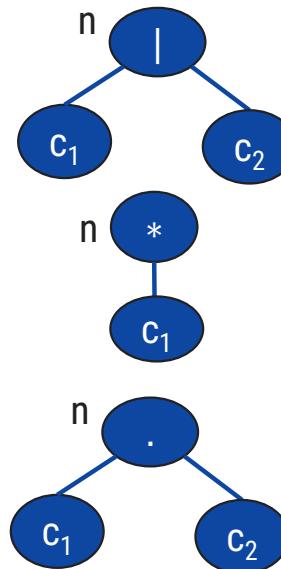


Step 1: Construct Syntax Tree

Step 2: Nullable node

A leaf labeled by  $\epsilon = \text{True}$

A leaf with position  $i = \text{false}$



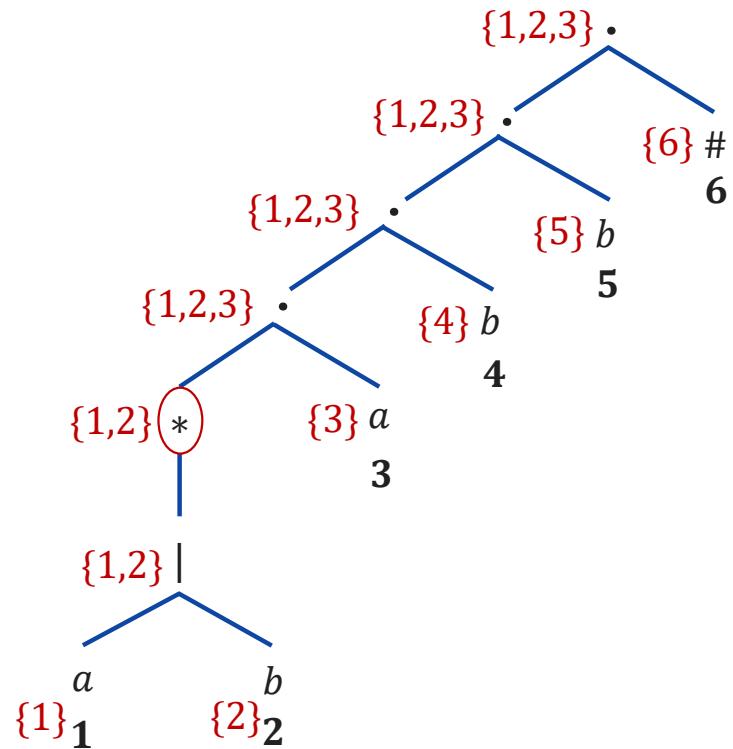
nullable( $c_1$ )  
or  
nullable( $c_2$ )

true

nullable( $c_1$ )  
and  
nullable( $c_2$ )

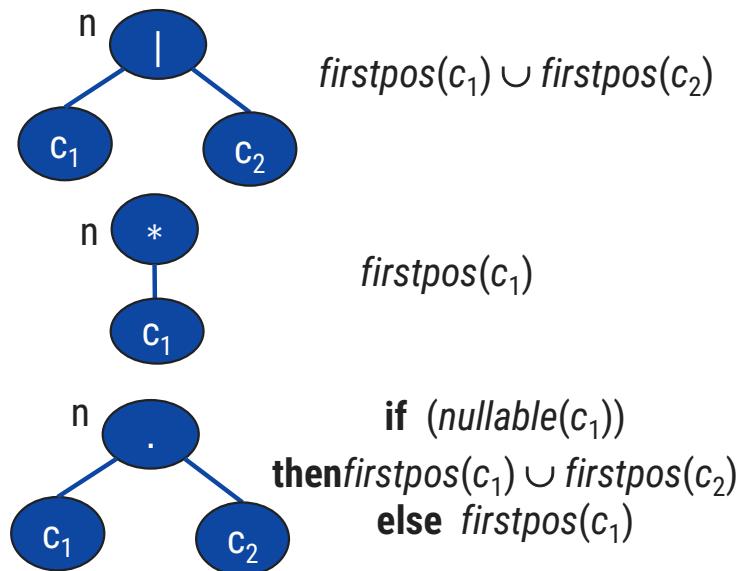
# Conversion from regular expression to DFA

Step 3: Calculate firstpos



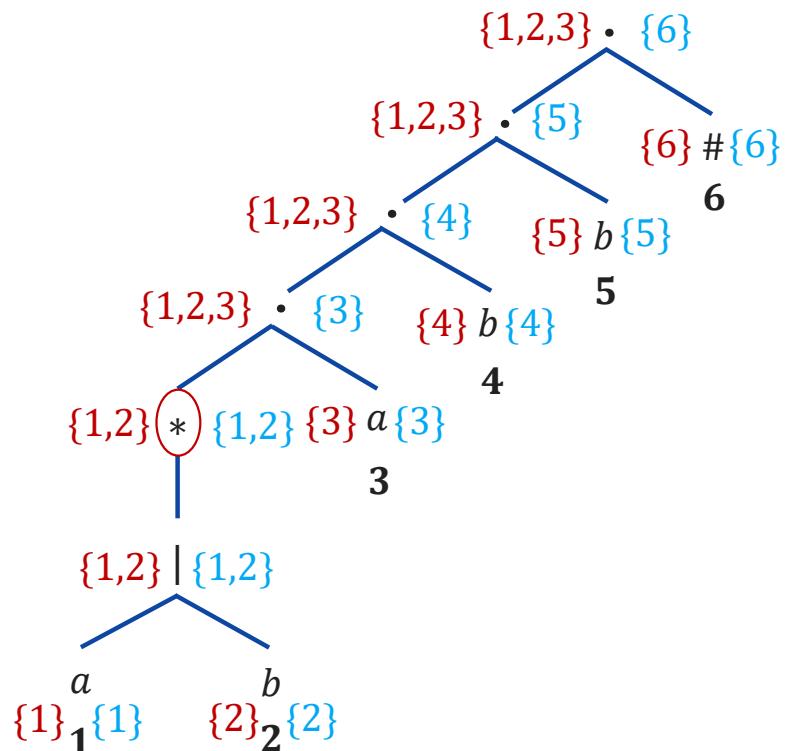
## Firstpos —

A leaf with position  $i = \{i\}$



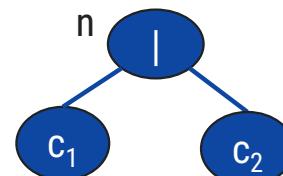
# Conversion from regular expression to DFA

Step 3: Calculate lastpos

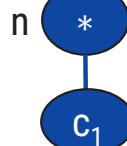


Lastpos —

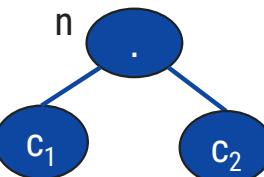
A leaf with position  $i = \{i\}$



$\text{lastpos}(c_1) \cup \text{lastpos}(c_2)$



$\text{lastpos}(c_1)$

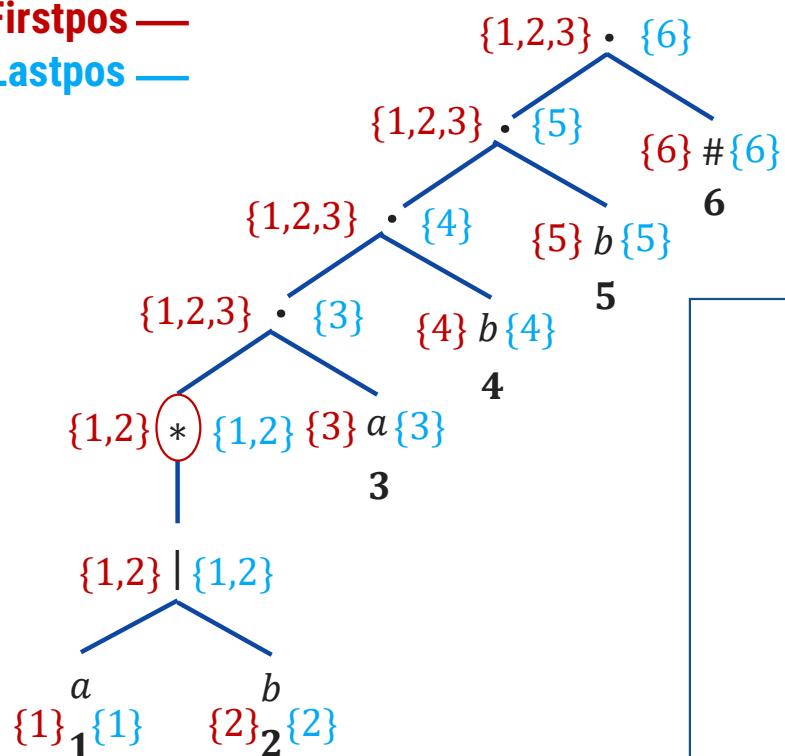


**if** (*nullable*( $c_2$ )) **then**  
 $\text{lastpos}(c_1) \cup \text{lastpos}(c_2)$   
**else**  $\text{lastpos}(c_2)$

# Conversion from regular expression to DFA

## Step 4: Calculate followpos

**Firstpos** —  
**Lastpos** —



Position	followpos
5	
4	
3	
2	1,2,
1	1,2,

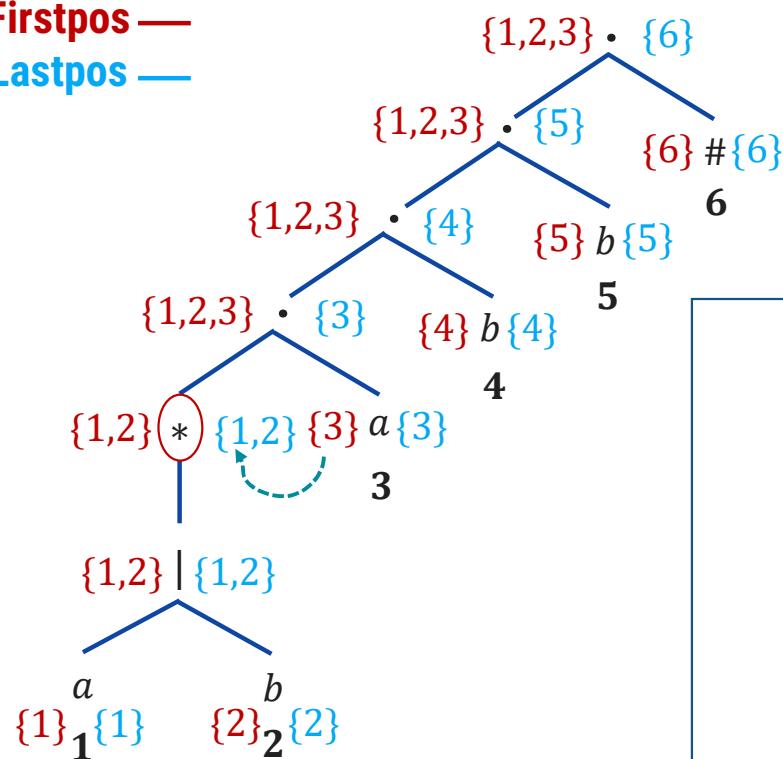
$$\{1,2\} \circledast_{\eta} \{1,2\}$$

$i = \text{lastpos}(n) = \{1,2\}$   
 $\text{firstpos}(n) = \{1,2\}$   
 $\text{followpos}(1) = \{1,2\}$   
 $\text{followpos}(2) = \{1,2\}$

# Conversion from regular expression to DFA

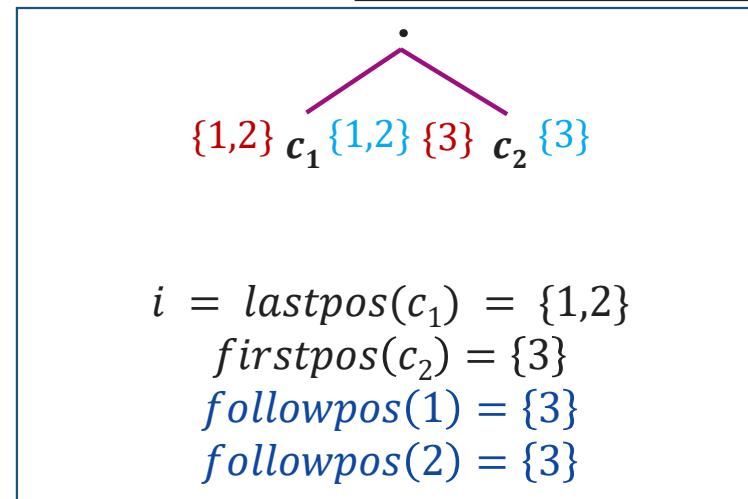
Step 4: Calculate followpos

**Firstpos** —  
**Lastpos** —



Position	followpos
5	
4	
3	
2	1,2,3
1	1,2,3

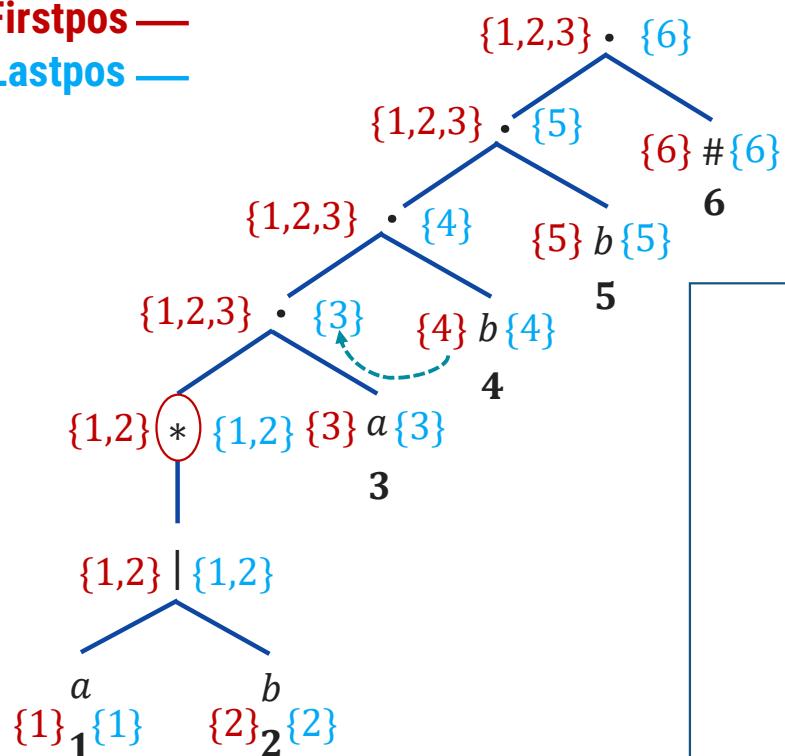
**Rule:**  
If  $n$  is **concatenation node** with left child  $c_1$  and right child  $c_2$  and  $i$  is a position in  $\text{lastpos}(c_1)$ , then all positions in  $\text{firstpos}(c_2)$  are in  $\text{followpos}(i)$



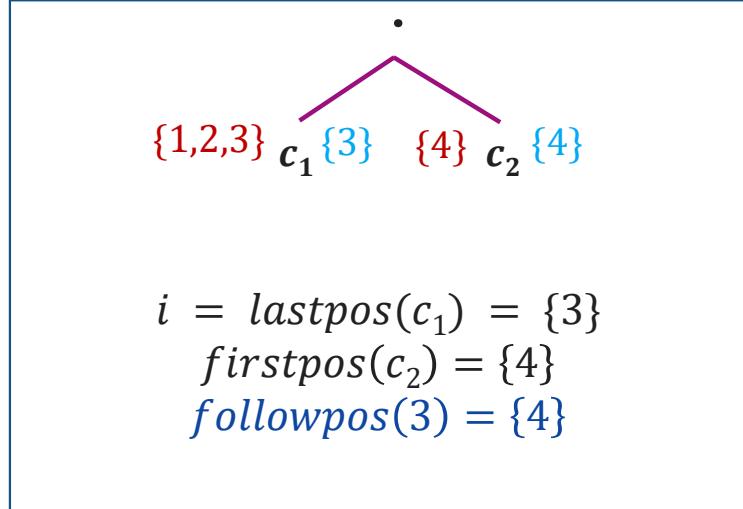
# Conversion from regular expression to DFA

Step 4: Calculate followpos

**Firstpos** —  
**Lastpos** —



Position	followpos
5	
4	
3	4
2	1,2,3
1	1,2,3

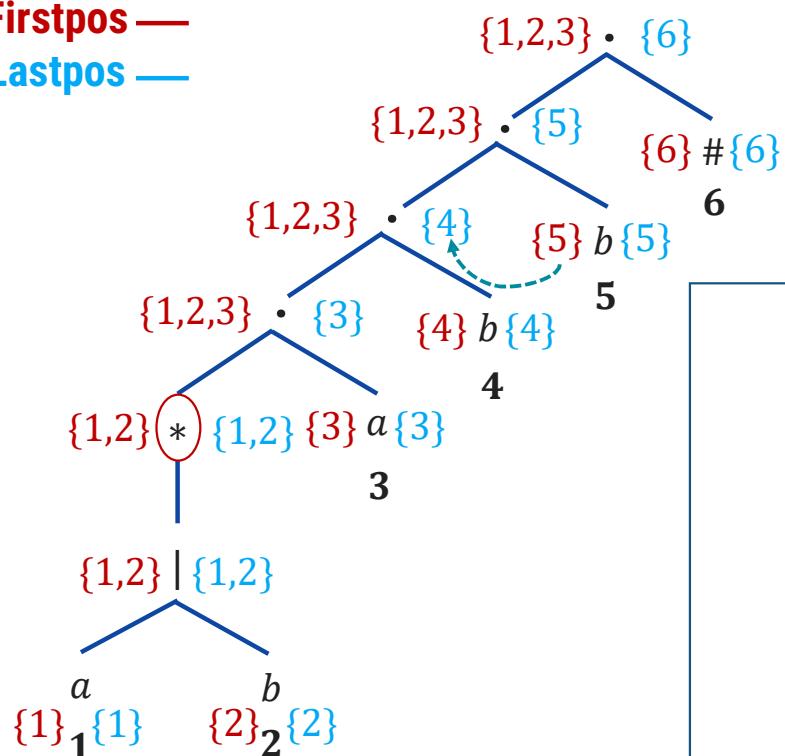


**Rule:**  
**If n is concatenation node**  
**with left child c1 and right**  
**child c2 and i is a position in**  
**lastpos(c1), then all position**  
**in firstpos(c2) are in**  
**followpos(i)**

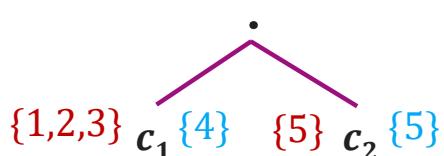
# Conversion from regular expression to DFA

Step 4: Calculate followpos

**Firstpos** —  
**Lastpos** —



Position	followpos
5	
4	5
3	4
2	1,2,3
1	1,2,3



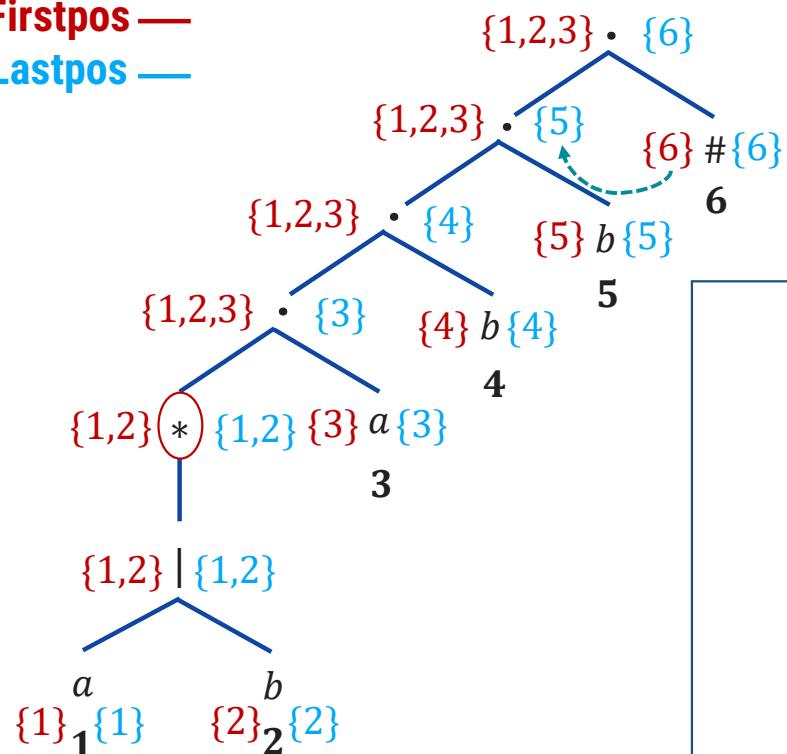
$$\begin{aligned}
 i &= \text{lastpos}(c_1) = \{4\} \\
 \text{firstpos}(c_2) &= \{5\} \\
 \text{followpos}(4) &= \{5\}
 \end{aligned}$$

**Rule:**  
If n is **concatenation** node with left child c<sub>1</sub> and right child c<sub>2</sub> and i is a position in **lastpos(c<sub>1</sub>)**, then all position in **firstpos(c<sub>2</sub>)** are in **followpos(i)**

# Conversion from regular expression to DFA

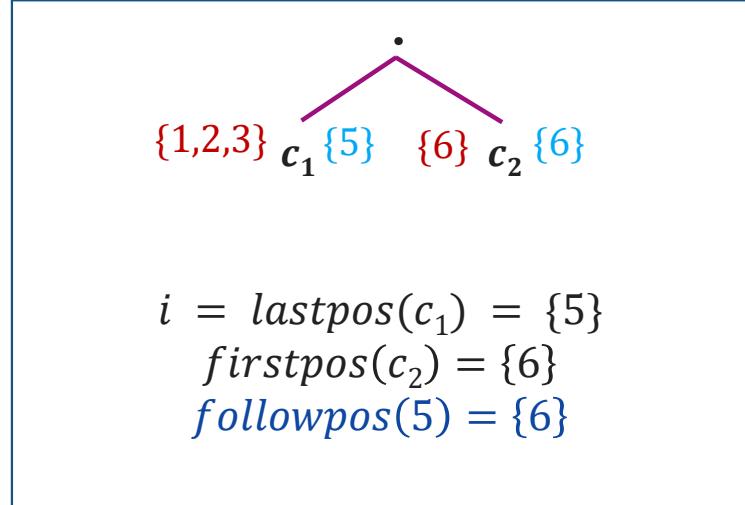
Step 4: Calculate followpos

**Firstpos** —  
**Lastpos** —



Position	followpos
5	6
4	5
3	4
2	1,2,3
1	1,2,3

**Rule:**  
If  $n$  is **concatenation node** with left child  $c_1$  and right child  $c_2$  and  $i$  is a position in  $\text{lastpos}(c_1)$ , then all positions in  $\text{firstpos}(c_2)$  are in  $\text{followpos}(i)$



# Conversion from regular expression to DFA

Initial state = *firstpos* of root = {1,2,3} ---- A

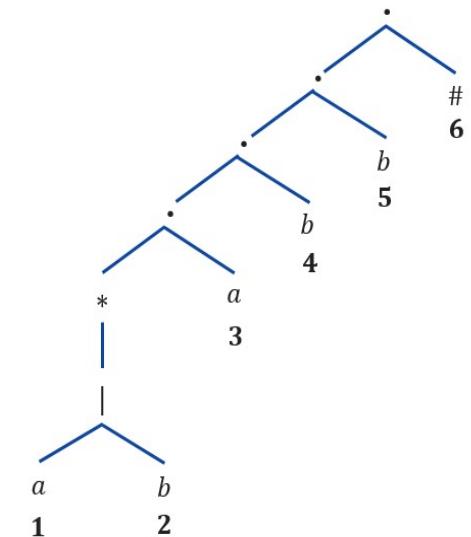
## State A

$$\begin{aligned}\delta((1,2,3),a) &= \text{followpos}(1) \cup \text{followpos}(3) \\ &= (1,2,3) \cup (4) = \{1,2,3,4\} \text{ ---- B}\end{aligned}$$

$$\begin{aligned}\delta((1,2,3),b) &= \text{followpos}(2) \\ &= (1,2,3) \text{ ---- A}\end{aligned}$$

Position	followpos
5	6
4	5
3	4
2	1,2,3
1	1,2,3

States	a	b
A={1,2,3}		
B={1,2,3,4}		



# Conversion from regular expression to DFA

## State B

$$\begin{aligned}\delta((1,2,3,4),a) &= \text{followpos}(1) \cup \text{followpos}(3) \\ &= (1,2,3) \cup (4) = \{1,2,3,4\} \text{ ---- B}\end{aligned}$$

$$\begin{aligned}\delta((1,2,3,4),b) &= \text{followpos}(2) \cup \text{followpos}(4) \\ &= (1,2,3) \cup (5) = \{1,2,3,5\} \text{ ---- C}\end{aligned}$$

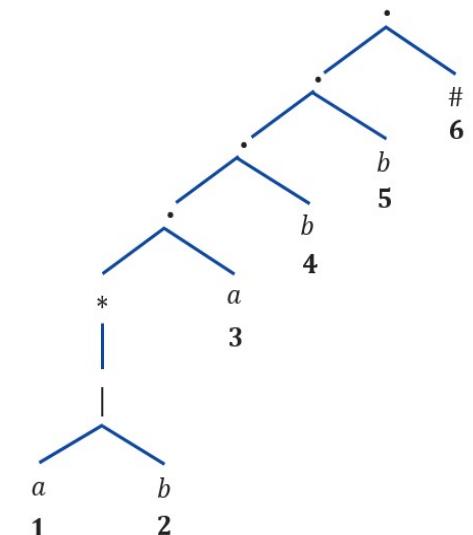
## State C

$$\begin{aligned}\delta((1,2,3,5),a) &= \text{followpos}(1) \cup \text{followpos}(3) \\ &= (1,2,3) \cup (4) = \{1,2,3,4\} \text{ ---- B}\end{aligned}$$

$$\begin{aligned}\delta((1,2,3,5),b) &= \text{followpos}(2) \cup \text{followpos}(5) \\ &= (1,2,3) \cup (6) = \{1,2,3,6\} \text{ ---- D}\end{aligned}$$

Position	followpos
5	6
4	5
3	4
2	1,2,3
1	1,2,3

States	a	b
A={1,2,3}	B	A
B={1,2,3,4}		
C={1,2,3,5}		
D={1,2,3,6}		

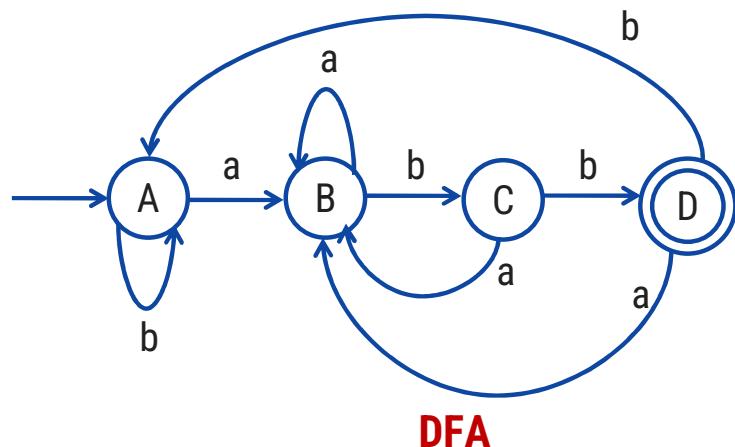


# Conversion from regular expression to DFA

## State D

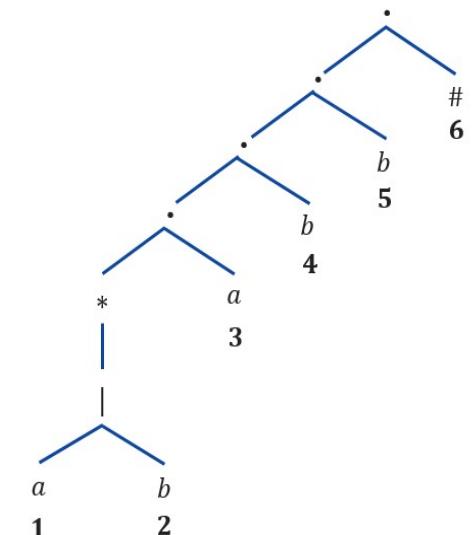
$$\delta((1,2,3,6),a) = \text{followpos}(1) \cup \text{followpos}(3) \\ = (1,2,3) \cup (4) = \{1,2,3,4\} \text{ ---- B}$$

$$\delta((1,2,3,6),b) = \text{followpos}(2) \\ = (1,2,3) \text{ ---- A}$$



Position	followpos
5	6
4	5
3	4
2	1,2,3
1	1,2,3

States	a	b
A={1,2,3}	B	A
B={1,2,3,4}	B	C
C={1,2,3,5}	B	D
D={1,2,3,6}		



# An Elementary Scanner Design & It's Implementation

# An Elementary Scanner Design & Its Implementation

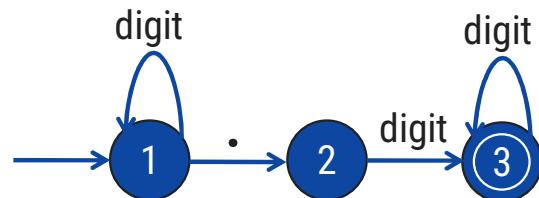
## Tasks of Scanner

1. The main purpose of the scanner is to return the next input token to the parser.
2. The scanner must identify the complete token and sometimes differentiate between keywords and identifiers.
3. The scanner may perform symbol-table maintenance, inserting identifiers, literals, and constants into the tables.
4. The scanner also eliminate the white spaces.

**Regular Expression:** Tokens can be **Specified** using regular expression.

Example: id → letter(letter | digit)\*

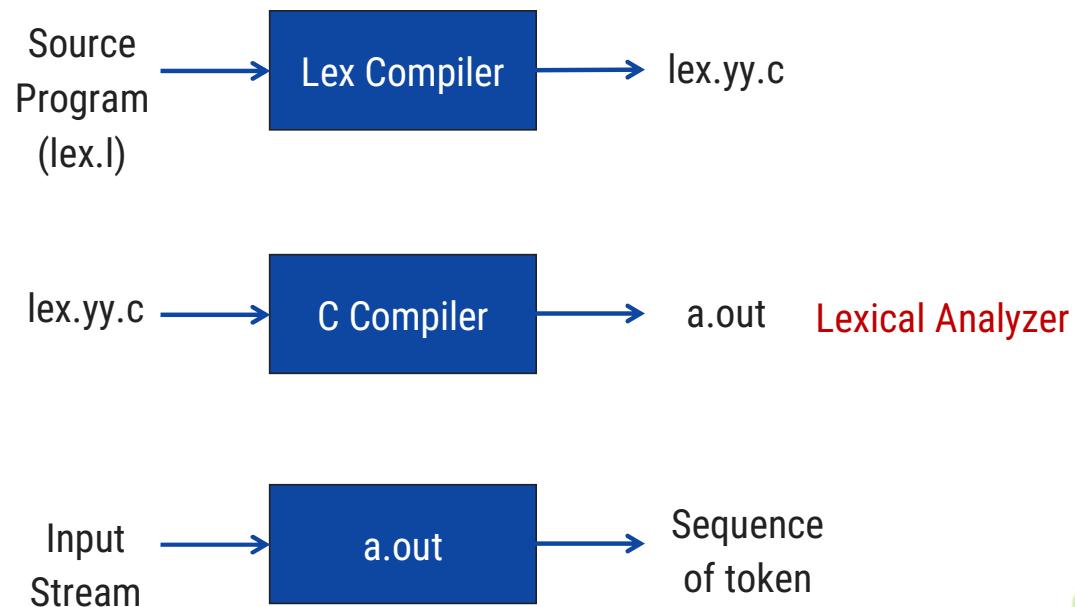
**Transition Diagram:** Finite-state diagrams or transition diagrams are often used to recognize a token



# Implementation of Lexical Analyzer (Lex)

- ▶ Lex is tool or a language which is useful for generating a lexical Analyzer and it specifies the regular expression
- ▶ Regular expression is used to represent the patterns for a token.

## Creating Lexical Analyzer with LEX



# Structure of Lex Program

- ▶ Any lex program contains mainly three sections
  1. Declaration
  2. Translation rules
  3. Auxiliary Procedures

## Structure of Program

**Declaration** → It is used to declare variables, constant & regular definition

Syntax: Pattern {Action}

E

Example:

%

%%

**Translation rule** → pattern1 {Action1}

%%

%

pattern2 {Action2}

D

pattern3 {Action3}

**Auxiliary Procedures** → All the function needed are specified over here.

# Example: Lex Program

- Program: Write Lex program to recognize identifier, keywords, relational operator and numbers

```
/* Declaration */                                /* Translation rule */  
%{  
    /* Lex program for recognizing tokens */  
}  
%%  
Letter      [a-zA-Z]  
Digit       [0-9]  
Id          {Letter}({Letter}|{Digit})*  
Numbers     {Digit}+ (. {Digit}+)? (E[+ -]?) {Digit}+?  
  
{Id}        {printf("%s is an identifier",yytext);}  
If          {printf("%s is a keyword",yytext);}  
else        {printf("%s is a keyword",yytext);}  
<           {printf("%s is a less than operator",yytext);}  
>=          {printf("%s is a greater than equal to operator",yytext);}  
{Numbers}   {printf("%s is a number",yytext);}  
%%
```

## /\* Auxiliary Procedures \*/

```
install_id()  
{  
    /* procedure to lexeme into the symbol table and return a pointer */  
}
```

Input string: If year < 2021

# References

## Books:

### 1. **Compilers Principles, Techniques and Tools, PEARSON Education (Second Edition)**

Authors: Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

### 2. **Compiler Design, PEARSON (for Gujarat Technological University)**

Authors: Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman

# Thank You



# Unit – 3

# Syntax Analysis (I)



**Prof. Dixita B. Kagathara**  
Computer Engineering Department  
Darshan Institute of Engineering & Technology, Rajkot  

---

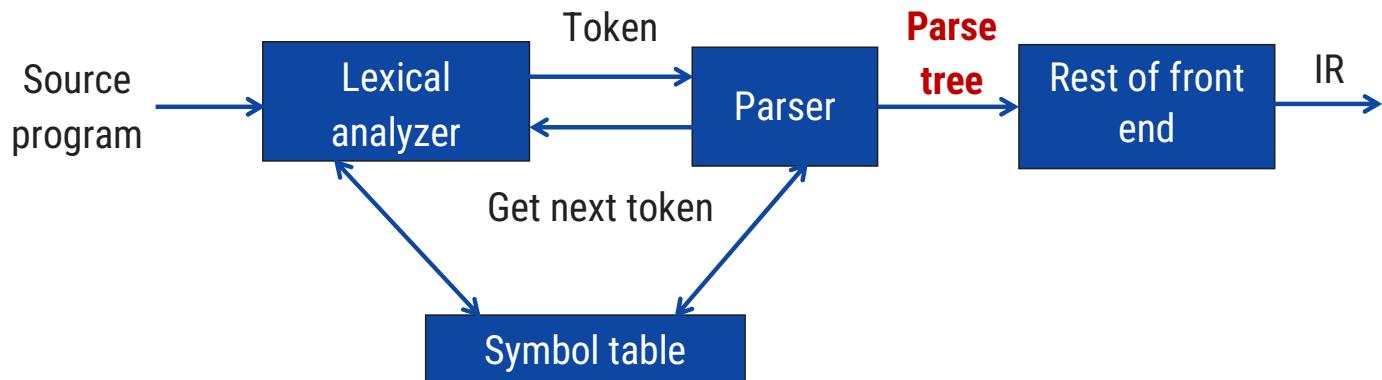
✉ dixita.kagathara@darshan.ac.in  
📞 +91 - 97277 47317 (CE Department)

# Topics to be covered

- Role of parser
- Context free grammar
- Derivation & Ambiguity
- Left recursion & Left factoring
- Classification of parsing
- Backtracking
- LL(1) parsing
- Recursive descent paring
- Shift reduce parsing
- Operator precedence parsing
- LR parsing
- Parser generator

# Role of Parser

# Role of parser



- ▶ Parser obtains a string of token from the lexical analyzer and reports **syntax error** if any otherwise generates **parse tree**.
- ▶ There are two types of parser:
  1. Top-down parser
  2. Bottom-up parser

# Context free grammar

# Context free grammar

- ▶ A context free grammar (CFG) is a 4-tuple  $G = (V, \Sigma, S, P)$  where,
  - $V$  is finite set of non terminals,
  - $\Sigma$  is disjoint finite set of terminals,
  - $S$  is an element of  $V$  and it's a start symbol,
  - $P$  is a finite set formulas of the form  $A \rightarrow \alpha$  where  $A \in V$  and  $\alpha \in (V \cup \Sigma)^*$

## ▶ Nonterminal symbol:

- ↳ The name of syntax category of a language, e.g., noun, verb, etc.
- ↳ It is written as a **single capital letter**, or as a **name enclosed between < ... >**, e.g., A or <Noun>

<Noun Phrase> → <Article><Noun>  
<Article> → a | an | the  
<Noun> → boy | apple

# Context free grammar

- ▶ A context free grammar (CFG) is a 4-tuple  $G = (V, \Sigma, S, P)$  where,
  - $V$  is finite set of non terminals,
  - $\Sigma$  is disjoint finite set of terminals,
  - $S$  is an element of  $V$  and it's a start symbol,
  - $P$  is a finite set formulas of the form  $A \rightarrow \alpha$  where  $A \in V$  and  $\alpha \in (V \cup \Sigma)^*$

- ▶ Terminal symbol:
  - ↳ A symbol in the alphabet.
  - ↳ It is denoted by lower case letter and punctuation marks used in language.

<Noun Phrase> → <Article><Noun>  
<Article> → a | an | the  
<Noun> → boy | apple

# Context free grammar

- ▶ A context free grammar (CFG) is a 4-tuple  $G = (V, \Sigma, S, P)$  where,
  - $V$  is finite set of non terminals,
  - $\Sigma$  is disjoint finite set of terminals,
  - $S$  is an element of  $V$  and it's a start symbol,
  - $P$  is a finite set formulas of the form  $A \rightarrow \alpha$  where  $A \in V$  and  $\alpha \in (V \cup \Sigma)^*$

## ▶ Start symbol:

- First nonterminal symbol of the grammar is called start symbol.

**<Noun Phrase>** → <Article><Noun>  
<Article> → a | an | the  
<Noun> → boy | apple

# Context free grammar

- ▶ A context free grammar (CFG) is a 4-tuple  $G = (V, \Sigma, S, P)$  where,
  - $V$  is finite set of non terminals,
  - $\Sigma$  is disjoint finite set of terminals,
  - $S$  is an element of  $V$  and it's a start symbol,
  - $P$  is a finite set formulas of the form  $A \rightarrow \alpha$  where  $A \in V$  and  $\alpha \in (V \cup \Sigma)^*$

## ▶ Production:

- ▶ A production, also called a rewriting rule, is a rule of grammar. It has the form of
  - A nonterminal symbol  $\rightarrow$  String of terminal and nonterminal symbols

```
<Noun Phrase> → <Article><Noun>
<Article> → a | an | the
<Noun> → boy | apple
```

# Example: Context Free Grammar

Write non terminals, terminals, start symbol, and productions for following grammar.

$E \rightarrow E \cup E | (E) | id$

$O \rightarrow + | - | * | / | \uparrow$

**Non terminals:**  $E, O$

**Terminals:**  $id, +, -, *, /, \uparrow, ()$

**Start symbol:**  $E$

**Productions:**  $E \rightarrow E \cup E | (E) | id$

$O \rightarrow + | - | * | / | \uparrow$

# Derivation

# Derivation

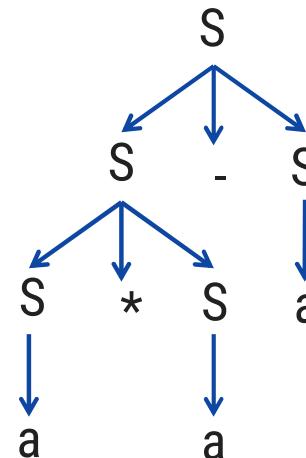
- ▶ A derivation is basically a sequence of production rules, in order to get the input string.
- ▶ To decide which non-terminal to be replaced with production rule, we can have two options:
  1. Leftmost derivation
  2. Rightmost derivation

# Leftmost derivation

- ▶ A derivation of a string  $W$  in a grammar  $G$  is a left most derivation if at every step the **left most non terminal** is replaced.
- ▶ Grammar:  $S \rightarrow S+S \mid S-S \mid S^*S \mid S/S \mid a$       Output string:  $a^*a-a$

$S$   
 $\rightarrow \underline{S-S}$   
 $\rightarrow \underline{S^*S-S}$   
 $\rightarrow a^*\underline{S-S}$   
 $\rightarrow a^*\underline{a-S}$   
 $\rightarrow a^*a-a$

**Leftmost Derivation**



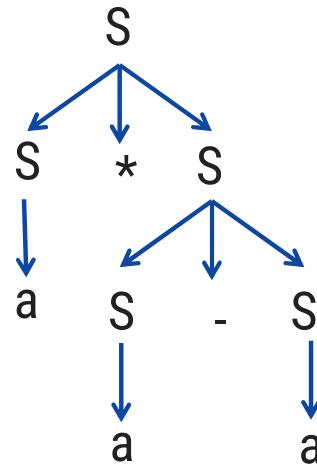
**Parse tree**

# Rightmost derivation

- ▶ A derivation of a string  $W$  in a grammar  $G$  is a right most derivation if at every step the right most non terminal is replaced.
- ▶ It is all called canonical derivation.
- ▶ Grammar:  $S \rightarrow S+S \mid S-S \mid S^*S \mid S/S \mid a$       Output string:  $a^*a-a$

S  
 $\rightarrow S^*S$   
 $\rightarrow S^*S-S$   
 $\rightarrow S^*S-a$   
 $\rightarrow S^*a-a$   
 $\rightarrow a^*a-a$

## Rightmost Derivation

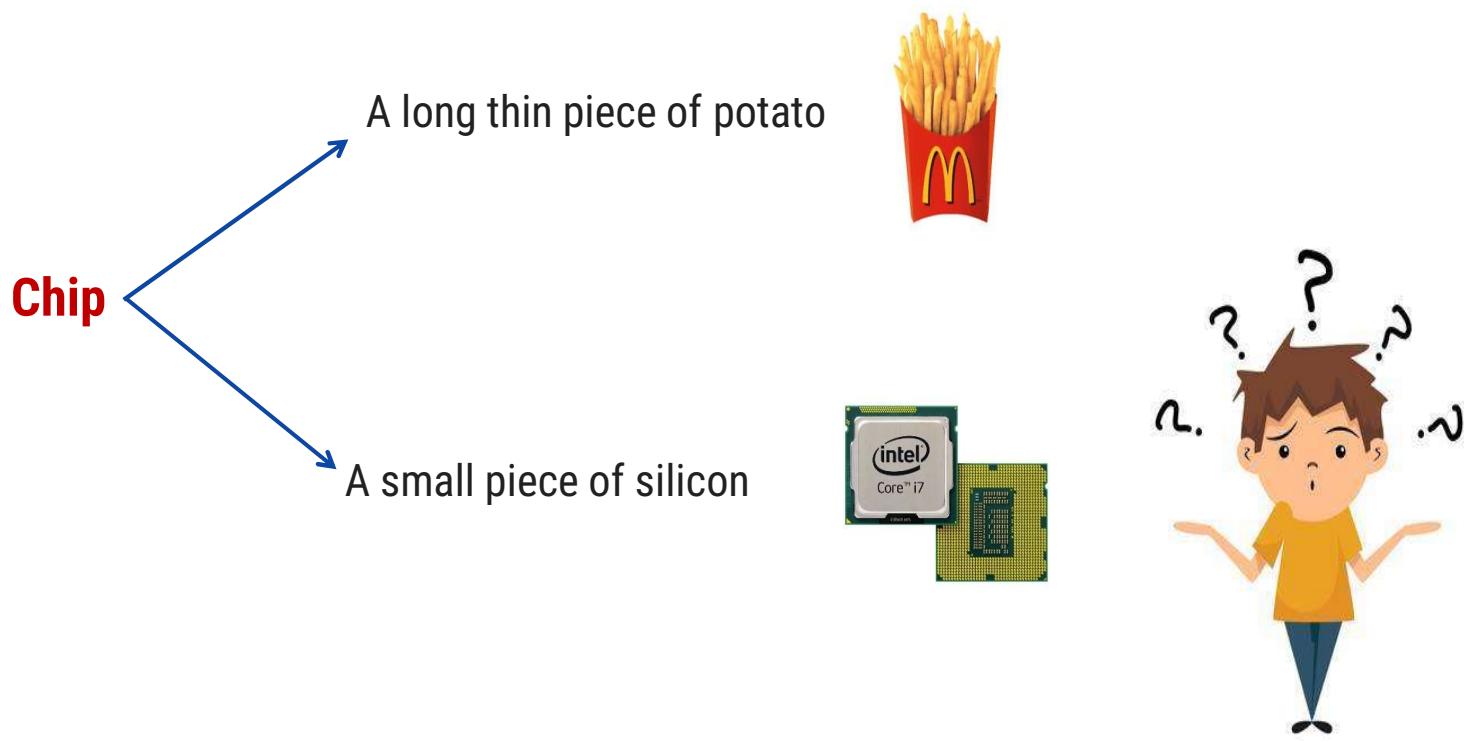


Parse Tree

# Ambiguous grammar

# Ambiguity

- ▶ Ambiguity, is a word, phrase, or statement which contains **more than one meaning**.



# Ambiguity

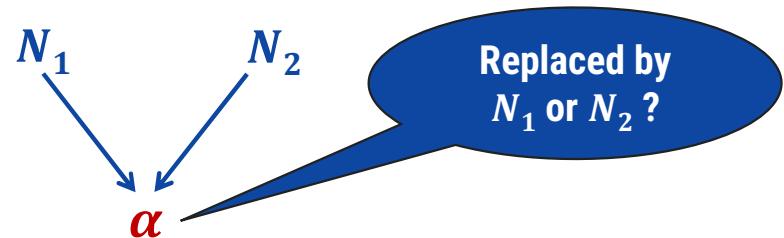
- In formal language grammar, ambiguity would arise **if identical string can occur on the RHS of two or more productions.**

- Grammar:

$$N_1 \rightarrow \alpha$$

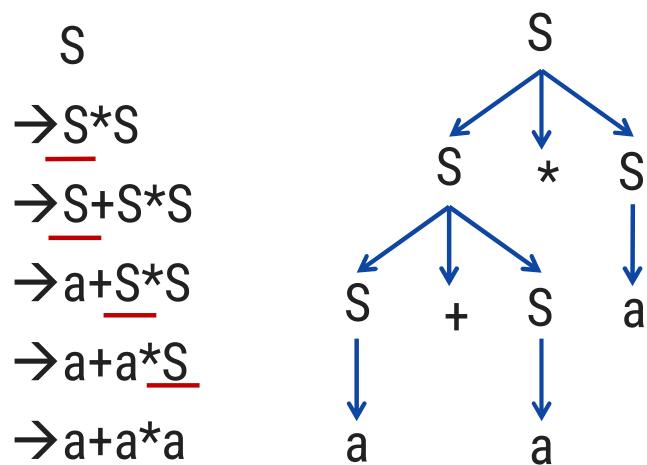
$$N_2 \rightarrow \alpha$$

- $\alpha$  can be derived from either  $N_1$  or  $N_2$

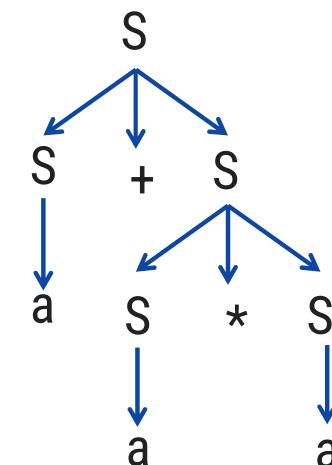
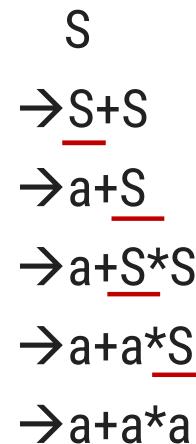


# Ambiguous grammar

- ▶ Ambiguous grammar is one that produces more than one leftmost or more than one rightmost derivation for the same sentence.
- ▶ Grammar:  $S \rightarrow S+S \mid S^*S \mid (S) \mid a$



Output string:  $a+a^*a$



- ▶ Here, Two leftmost derivation for string  $a+a^*a$  is possible hence, above grammar is ambiguous.

# Parsing

# Parsing

- ▶ Parsing is a technique that takes input string and produces output either a **parse tree** if string is valid sentence of grammar, or an **error message** indicating that string is not a valid.

## Types of Parsing

**Top down parsing:** In top down parsing parser build parse tree from top to bottom.

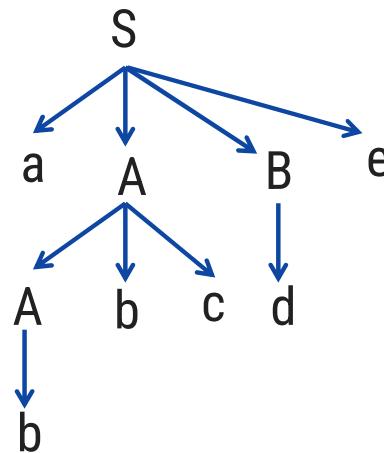
Grammar:

$$S \rightarrow aABe$$

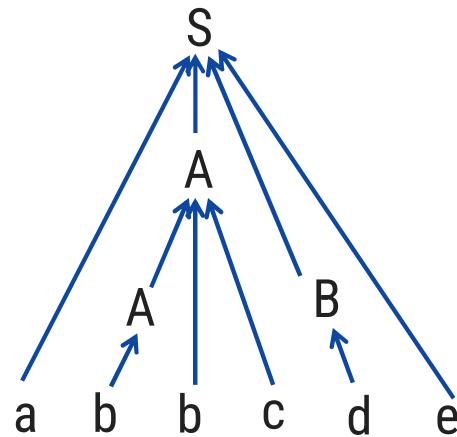
$$A \rightarrow Abc \mid b$$

$$B \rightarrow d$$

String: abcd<sup>e</sup>

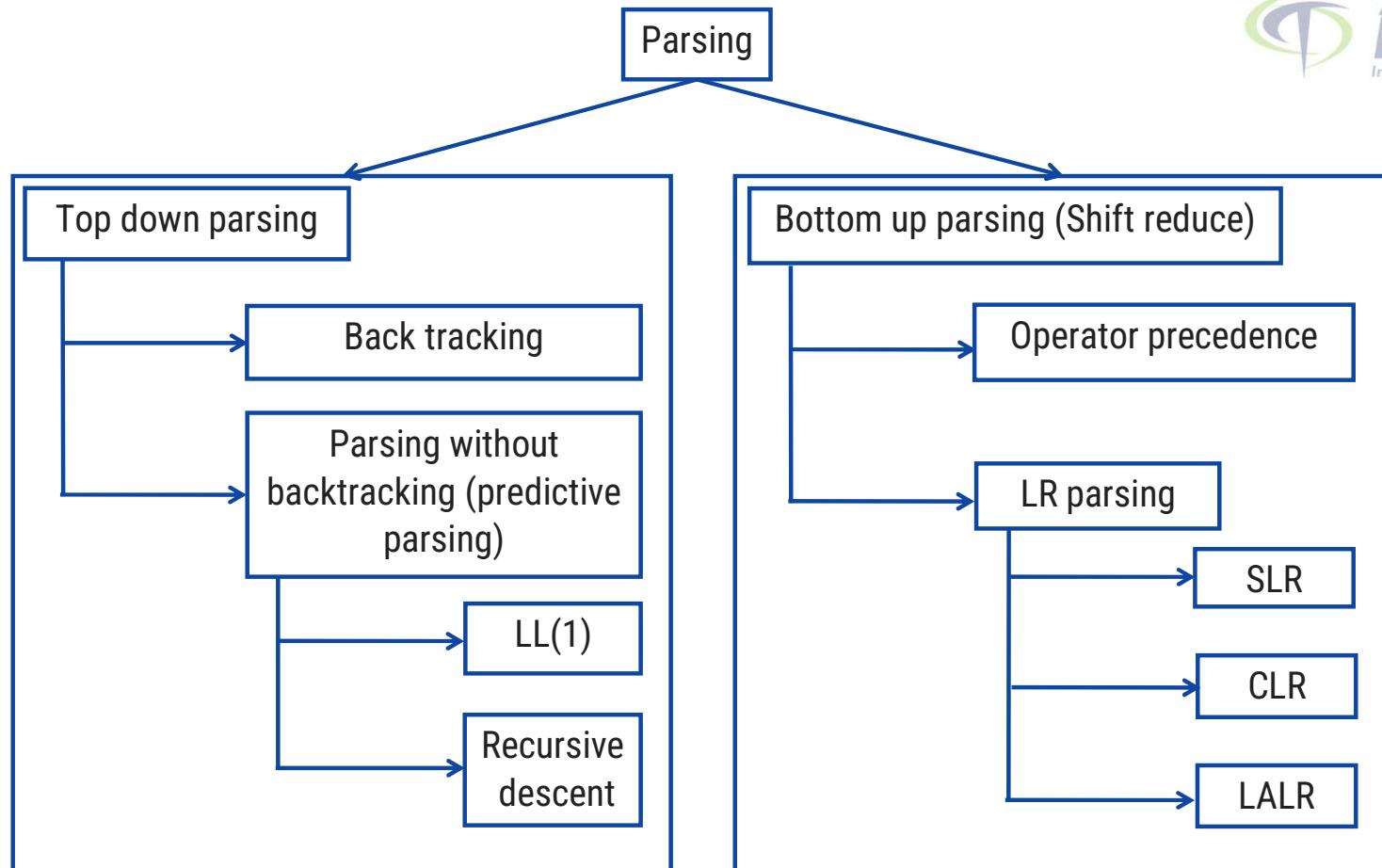


**Bottom up parsing:** Bottom up parser starts from leaves and work up to the root.

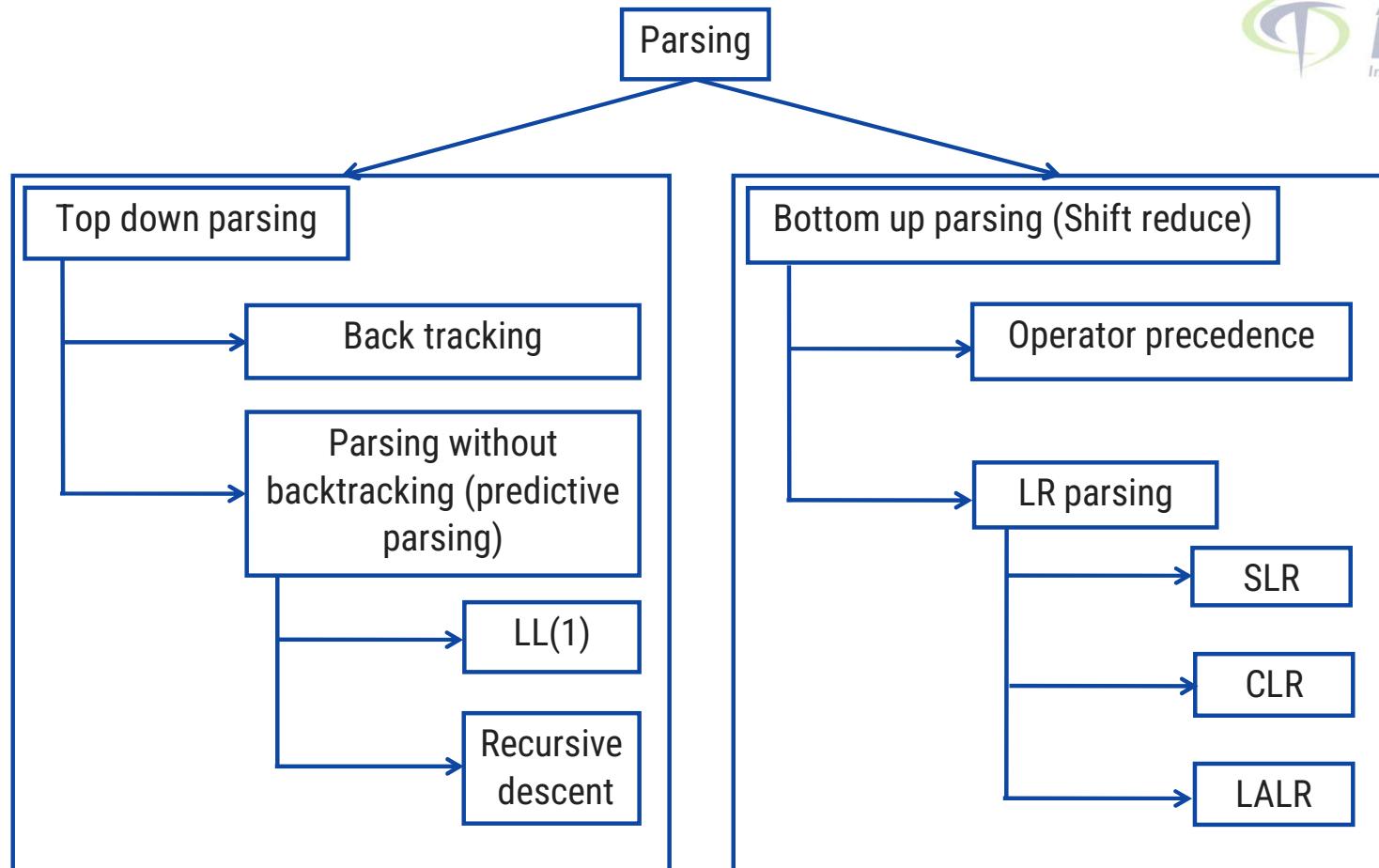


# Classification of Parsing

# Classification of parsing



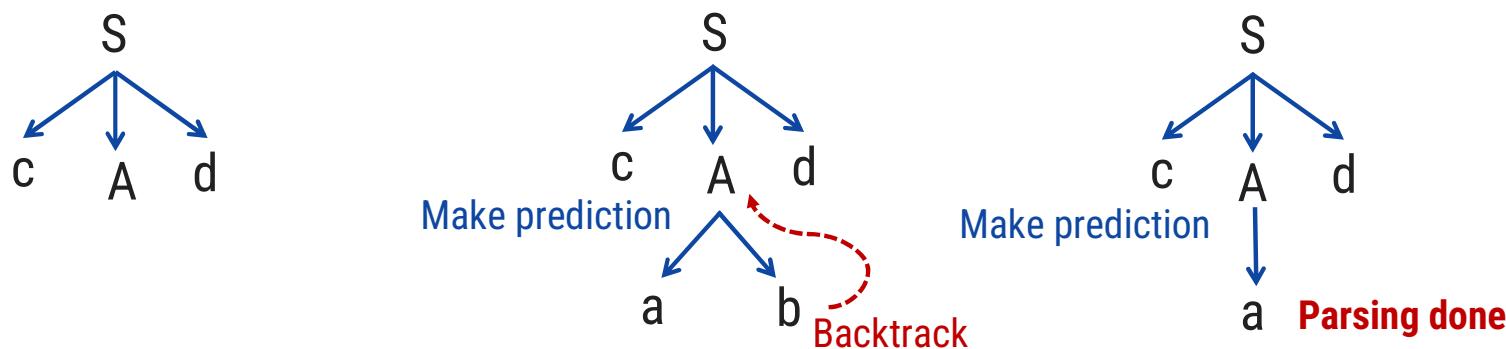
# Classification of parsing



# Backtracking

# Backtracking

- In backtracking, expansion of nonterminal symbol we choose one alternative and if any mismatch occurs then we try another alternative.
- Grammar:  $S \rightarrow cAd$       Input string: cad  
 $A \rightarrow ab \mid a$

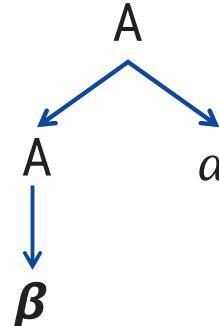
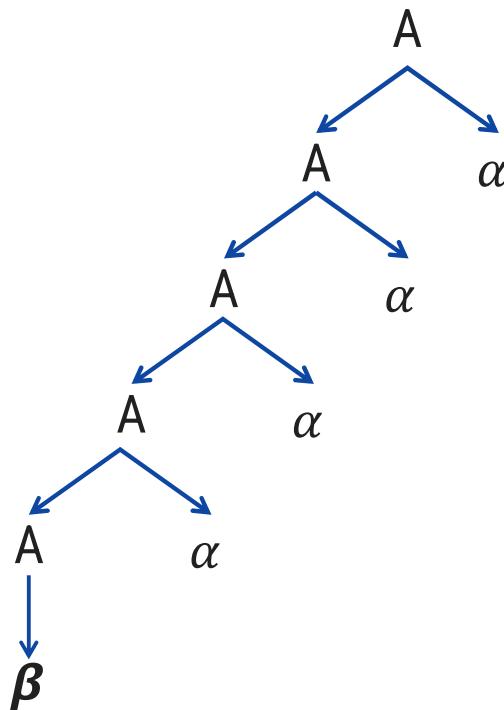


# Left Recursion

Problems in Top-down Parsing

# Left recursion

- ▶ A grammar is said to be **left recursive** if it has a non terminal  $A$  such that there is a derivation  $A \rightarrow A\alpha$  for some string  $\alpha$ .
- ▶ Grammar:  $A \rightarrow A\alpha \mid \beta$



# Left recursion elimination

$\beta\alpha^*$

$$A \rightarrow A\alpha \mid \beta \quad \xrightarrow{\beta\alpha^*} \quad A \rightarrow \quad A'$$
$$A' \rightarrow \quad A' \mid \epsilon$$

# Examples: Left recursion elimination

$E \rightarrow E + T \mid T$

$E \rightarrow TE'$

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

$T \rightarrow T * F \mid F$

$T \rightarrow FT'$

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

$X \rightarrow X \% Y \mid Z$

$X \rightarrow ZX'$

$X' \rightarrow \% Y X' \mid \epsilon$

# Left Factoring

Problems in Top-down Parsing

# Left factoring

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$$

- ▶ Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.
- ▶ It is used to remove nondeterminism from the grammar.

# Left factoring

$$A \rightarrow \alpha \beta \mid \alpha \delta \xrightarrow{\hspace{1cm}} A \rightarrow A'$$
$$A' \rightarrow \mid$$

# Example: Left factoring

$S \rightarrow aAB \mid aCD$

$S \rightarrow aS'$

$S' \rightarrow AB \mid CD$

$A \rightarrow xByA \mid xByAzA \mid a$

$A \rightarrow xByAA' \mid a$

$A' \rightarrow \epsilon \mid zA$

# First & Follow

# Rules to compute first of non terminal

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

Everything in  $FIRST(Y_1)$  is surely in  $FIRST(X)$ . If  $Y_1$  does not derive  $\epsilon$ , then we do nothing more to  $FIRST(X)$ , but if  $Y_1 \Rightarrow \epsilon$ , then we add  $FIRST(Y_2)$  and so on.

# Rules to compute first of non terminal

## Simplification of Rule 3

If  $A \rightarrow Y_1 Y_2 \dots \dots Y_K$ ,

- If  $Y_1$  does not derives  $\epsilon$  then,  $FIRST(A) = FIRST(Y_1)$
- If  $Y_1$  derives  $\epsilon$  then,  
 $FIRST(A) = FIRST(Y_1) - \epsilon \cup FIRST(Y_2)$
- If  $Y_1 \& Y_2$  derives  $\epsilon$  then,  
 $FIRST(A) = FIRST(Y_1) - \epsilon \cup FIRST(Y_2) - \epsilon \cup FIRST(Y_3)$
- If  $Y_1, Y_2 \& Y_3$  derives  $\epsilon$  then,  
 $FIRST(A) = FIRST(Y_1) - \epsilon \cup FIRST(Y_2) - \epsilon \cup FIRST(Y_3) - \epsilon \cup FIRST(Y_4)$
- If  $Y_1, Y_2, Y_3 \dots Y_K$  all derives  $\epsilon$  then,  
 $FIRST(A) = FIRST(Y_1) - \epsilon \cup FIRST(Y_2) - \epsilon \cup FIRST(Y_3) - \epsilon \cup FIRST(Y_4) - \epsilon \cup \dots \dots FIRST(Y_k)$  (note: if all non terminals derives  $\epsilon$  then add  $\epsilon$  to  $FIRST(A)$ )

# Rules to compute FOLLOW of non terminal

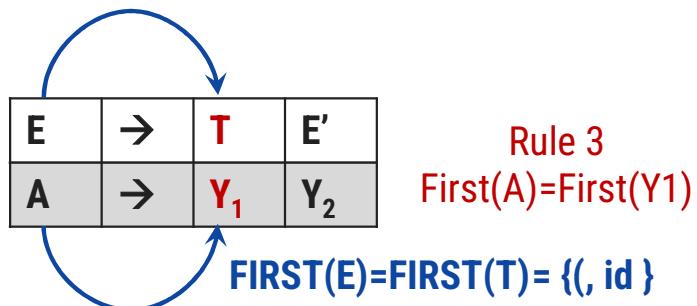
1. Place  $\$$  in  $follow(S)$ . ( $S$  is start symbol)
2. If  $A \rightarrow \alpha B \beta$ , then everything in  $FIRST(\beta)$  except for  $\epsilon$  is placed in  $FOLLOW(B)$
3. If there is a production  $A \rightarrow \alpha B$  or a production  $A \rightarrow \alpha B \beta$  where  $FIRST(\beta)$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

# Example-1: First & Follow

Compute FIRST

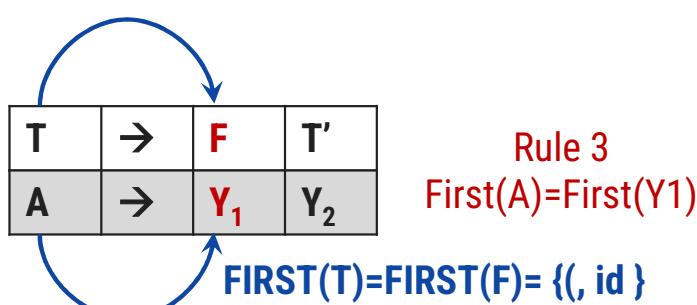
$\text{First}(E)$

$E \rightarrow TE'$



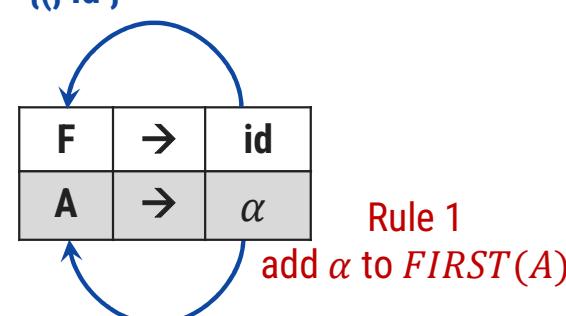
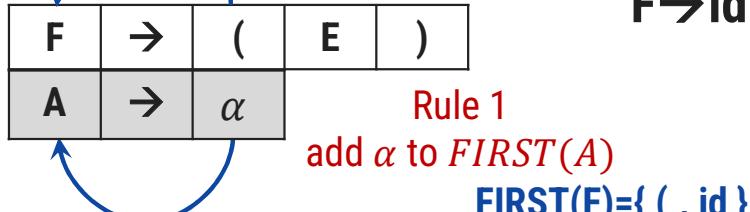
$\text{First}(T)$

$T \rightarrow FT'$



$\text{First}(F)$

$F \rightarrow (E)$



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

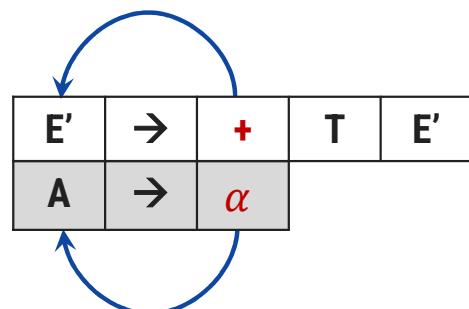
NT	First
E	
E'	
T	
T'	
F	

# Example-1: First & Follow

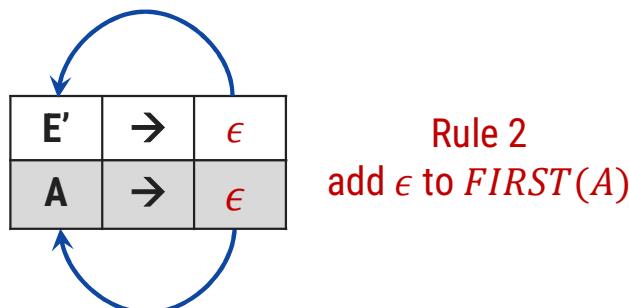
Compute FIRST

$\text{First}(E')$

$E' \rightarrow +TE'$



Rule 1  
add  $\alpha$  to  $\text{FIRST}(A)$



Rule 2  
add  $\epsilon$  to  $\text{FIRST}(A)$

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

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

NT	First
E	{ (, id }
E'	
T	{ (, id }
T'	
F	{ (, id }

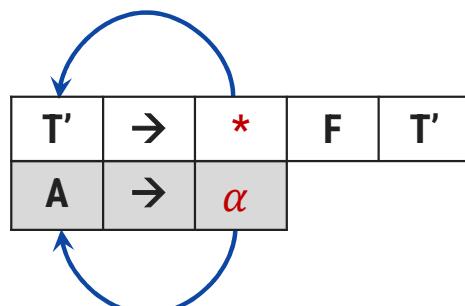
# Example-1: First & Follow

Compute FIRST

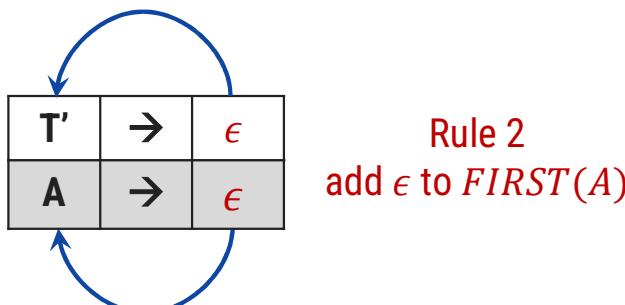
$\text{First}(T')$

$T' \rightarrow *FT'$

$T' \rightarrow \epsilon$



Rule 1  
add  $\alpha$  to  $\text{FIRST}(A)$



Rule 2  
add  $\epsilon$  to  $\text{FIRST}(A)$

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

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

NT	First
E	{ (, id }
E'	{ +, $\epsilon$ }
T	{ (, id }
T'	
F	{ (, id }

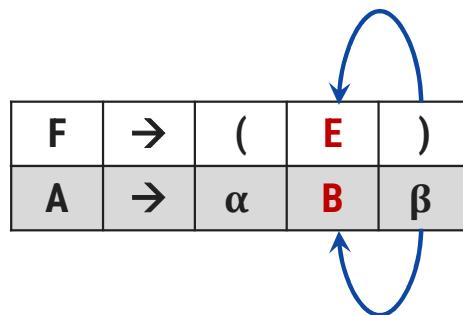
# Example-1: First & Follow

Compute FOLLOW

$\text{FOLLOW}(E)$

Rule 1: Place \$ in FOLLOW(E)

$F \rightarrow (E)$



Rule 2

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

NT	First	Follow
E	{ (, id } { +, * }	
E'	{ +, * }	
T	{ (, id } { +, * }	
T'	{ *, * }	
F	{ (, id }	

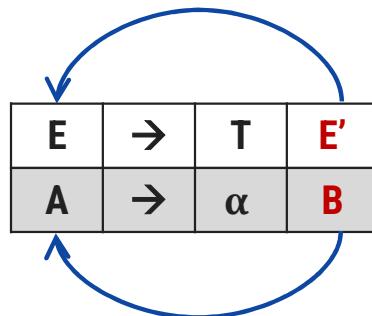
$\text{FOLLOW}(E)=\{ \$, ) \}$

# Example-1: First & Follow

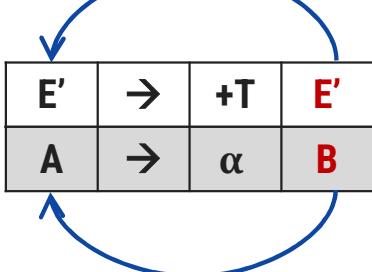
Compute FOLLOW

$\text{FOLLOW}(E')$

$E \rightarrow TE'$



$E' \rightarrow +TE'$



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

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

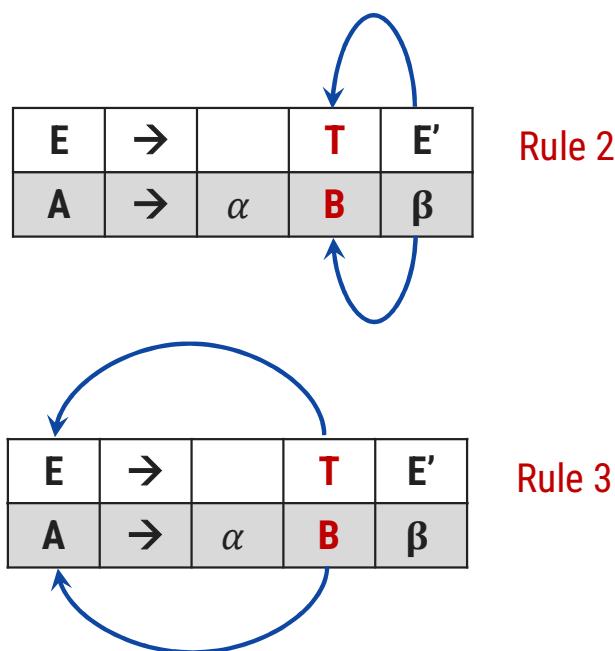
NT	First	Follow
E	{ (, id }	{ \$, ) }
E'	{ +, ε }	
T	{ (, id }	
T'	{ *, ε }	
F	{ (, id }	

# Example-1: First & Follow

Compute FOLLOW

$\text{FOLLOW}(T)$

$E \rightarrow TE'$



Rule 2

Rule 3

$$\text{FOLLOW}(T)=\{ +, \$, ) \}$$

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

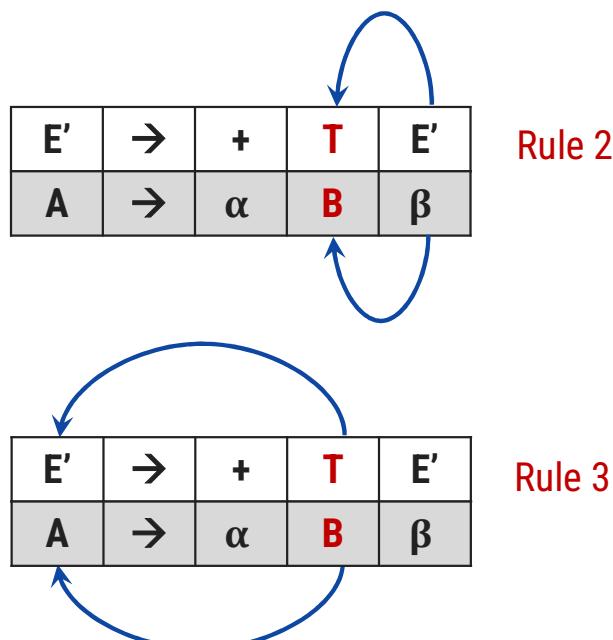
NT	First	Follow
$E$	$\{ (, \text{id} \} \}$	$\{ \$, ) \}$
$E'$	$\{ +, \epsilon \}$	$\{ \$, ) \}$
$T$	$\{ (, \text{id} \} \}$	
$T'$	$\{ *, \epsilon \}$	
$F$	$\{ (, \text{id} \} \}$	

# Example-1: First & Follow

Compute FOLLOW

$\text{FOLLOW}(T)$

$E' \rightarrow +TE'$



$$\text{FOLLOW}(T) = \{ +, \$, ) \}$$

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

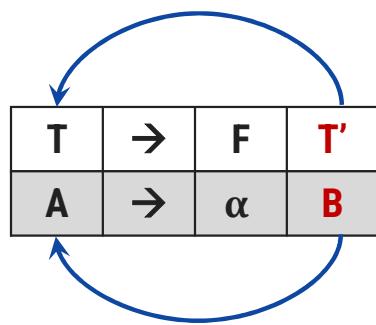
NT	First	Follow
E	{ (, id } { \$, ) }	
E'	{ +, ε }	{ \$, ) }
T	{ (, id }	
T'	{ *, ε }	
F	{ (, id }	

# Example-1: First & Follow

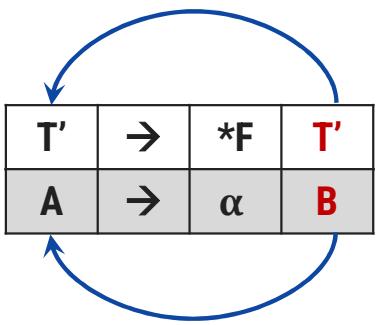
Compute FOLLOW

$\text{FOLLOW}(T')$

$T \rightarrow FT'$



$T' \rightarrow *FT'$



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

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

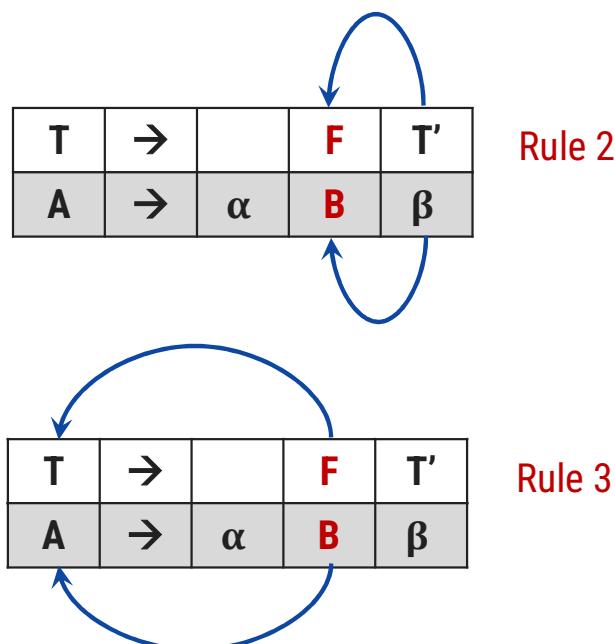
NT	First	Follow
E	{ (,id } { \$, ) }	{ \$, ) }
E'	{ +, ε }	{ \$, ) }
T	{ (,id }	{ +,\$, ) }
T'	{ *, ε }	
F	{ (,id }	

# Example-1: First & Follow

Compute FOLLOW

$\text{FOLLOW}(F)$

$T \rightarrow FT'$



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

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

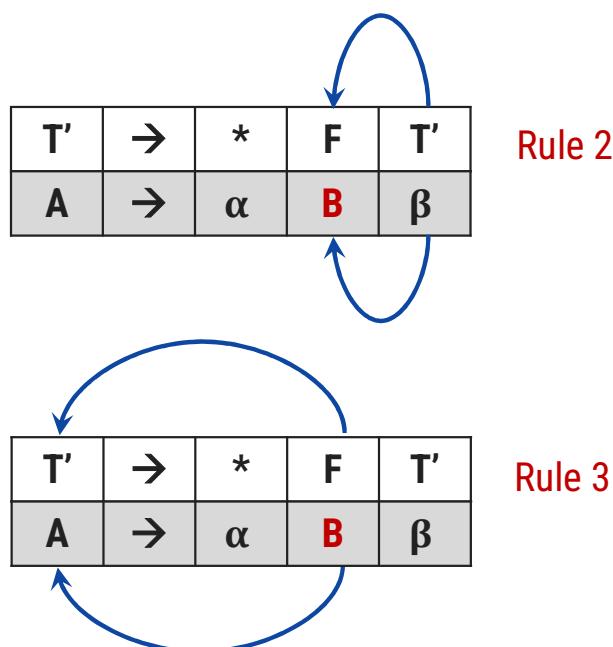
NT	First	Follow
E	{ (, id } { \$, ) }	{ \$, ) }
E'	{ +, $\epsilon$ }	{ \$, ) }
T	{ (, id }	{ +, \$, ) }
T'	{ *, $\epsilon$ }	{ +, \$, ) }
F	{ (, id }	

# Example-1: First & Follow

Compute FOLLOW

$\text{FOLLOW}(F)$

$T' \rightarrow *FT'$



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

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

NT	First	Follow
$E$	$\{ (, \text{id} \} \}$	$\{ \$, ) \}$
$E'$	$\{ +, \epsilon \}$	$\{ \$, ) \}$
$T$	$\{ (, \text{id} \} \}$	$\{ +, \$, ) \}$
$T'$	$\{ *, \epsilon \}$	$\{ +, \$, ) \}$
$F$	$\{ (, \text{id} \} \}$	

## Example-2: First & Follow

$S \rightarrow ABCDE$

$A \rightarrow a | \epsilon$

$B \rightarrow b | \epsilon$

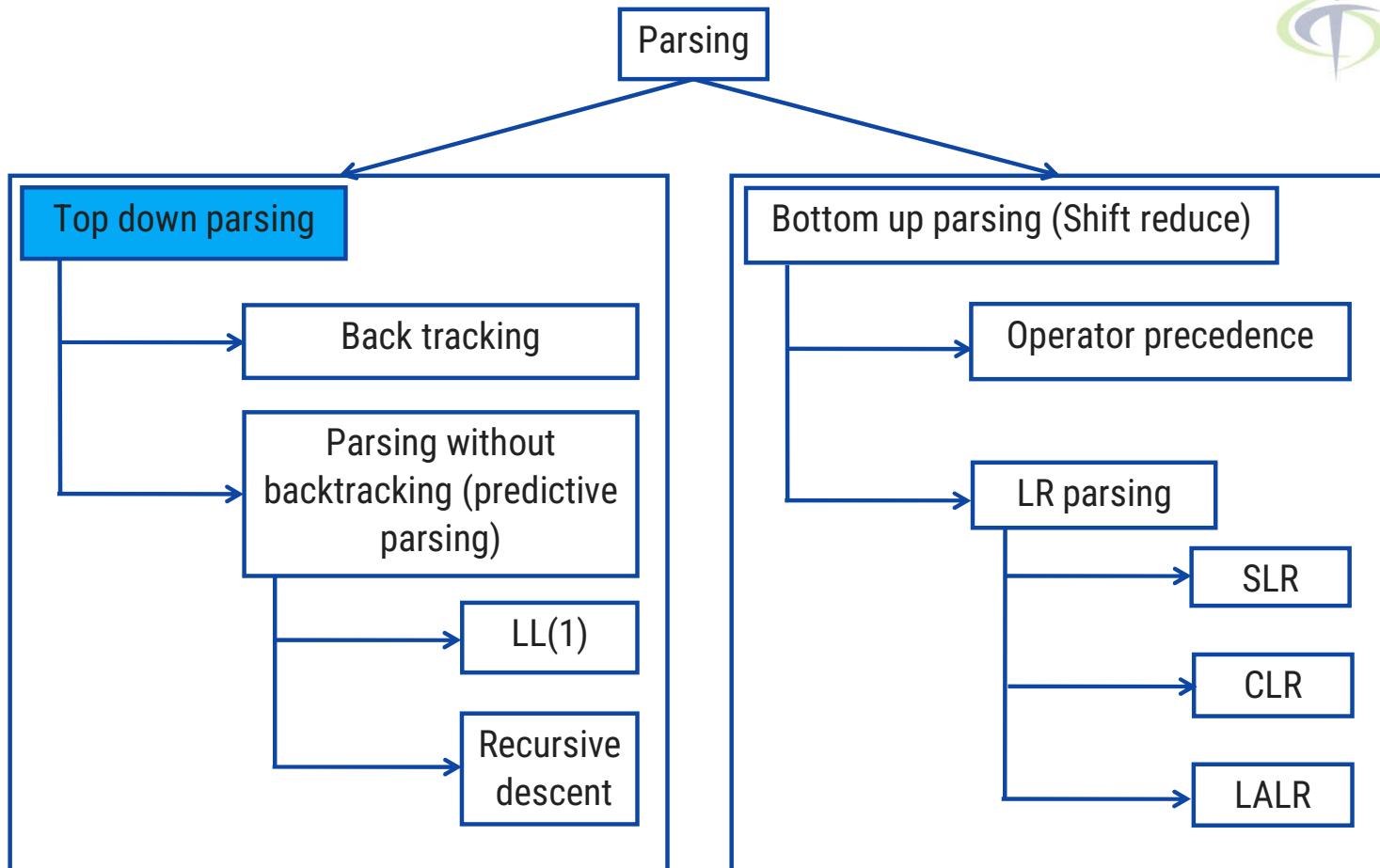
$C \rightarrow c$

$D \rightarrow d | \epsilon$

$E \rightarrow e | \epsilon$

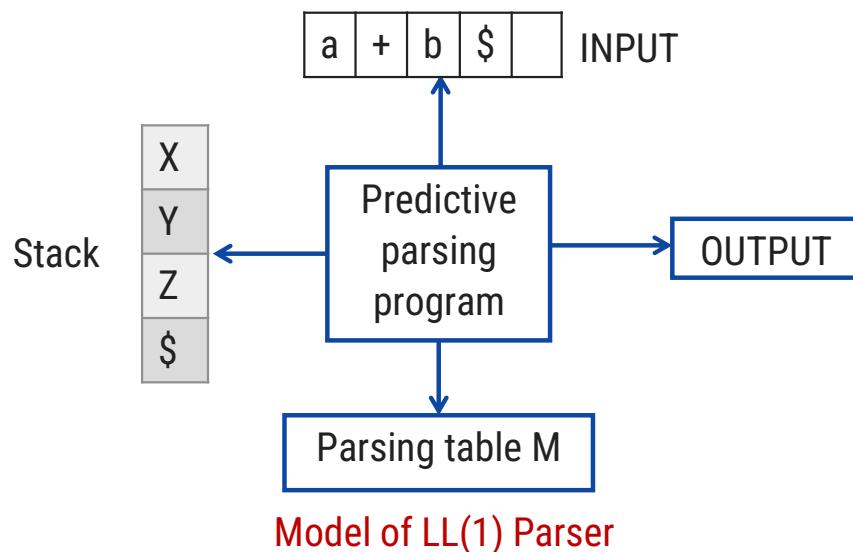
NT	First	Follow
S		
A	a	cde
B	b	cde
C	c	
D	d	cde
E	e	cde

# Parsing Methods



# LL(1) parser (Predictive parser or Non recursive descent parser)

- ▶ LL(1) is non recursive top down parser.
  1. First **L** indicates input is scanned from left to right.
  2. The second **L** means it uses leftmost derivation for input string
  3. **1** means it uses only input symbol to predict the parsing process.



# LL(1) parsing (predictive parsing)

Steps to construct LL(1) parser

1. Remove left recursion / Perform left factoring (if any).
2. Compute FIRST and FOLLOW of non terminals.
3. Construct predictive parsing table.
4. Parse the input string using parsing table.

# Rules to construct predictive parsing table

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

# Example-1: LL(1) parsing

$S \rightarrow aBa$

$B \rightarrow bB \mid \epsilon$

Step 1: Not required

Step 2: Compute FIRST

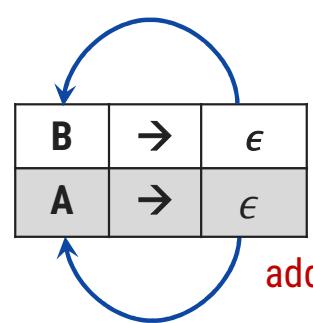
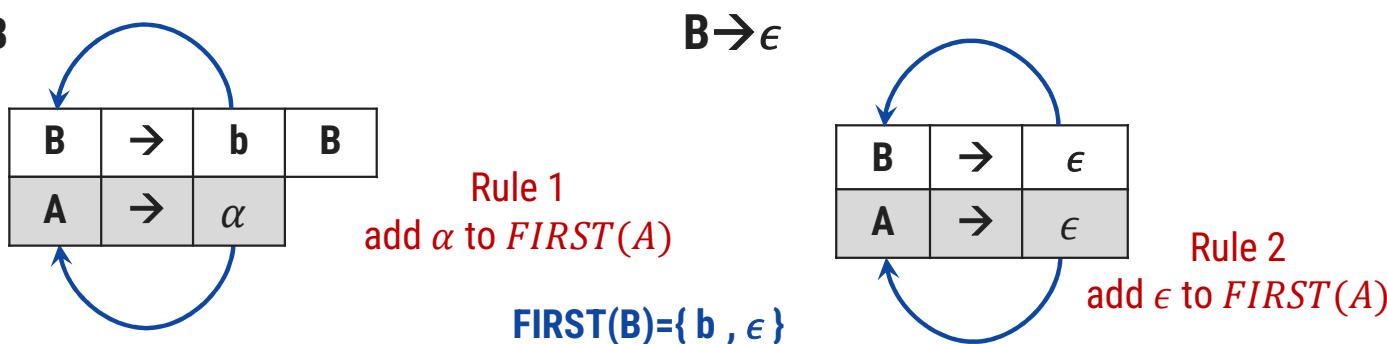
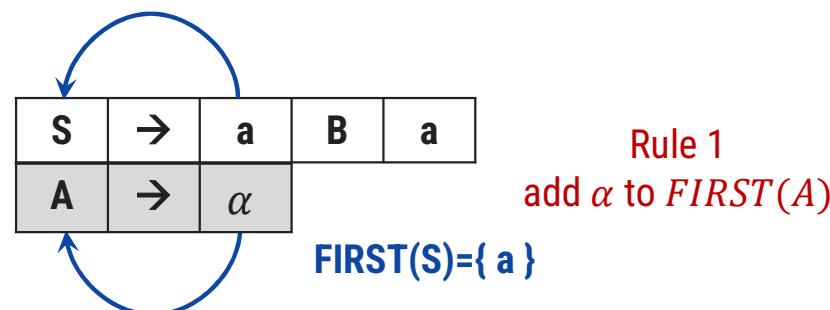
$\text{First}(S)$

$S \rightarrow aBa$

$\text{First}(B)$

$B \rightarrow bB$

NT	First
S	
B	



# Example-1: LL(1) parsing

$S \rightarrow aBa$

$B \rightarrow bB \mid \epsilon$

Step 2: Compute FOLLOW

$\text{Follow}(S)$

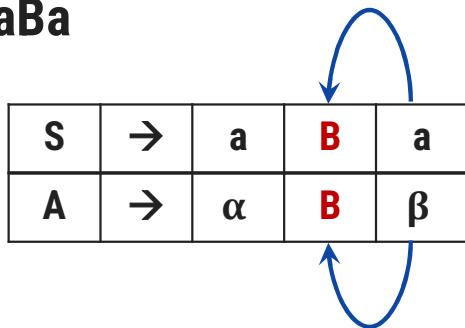
NT	First	Follow
S	{a}	
B	{b, $\epsilon$ }	

Rule 1: Place \$ in FOLLOW(S)

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

$\text{Follow}(B)$

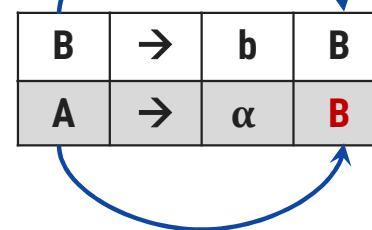
$S \rightarrow aBa$



Rule 2  
 $\text{First}(\beta) - \epsilon$

$$\text{Follow}(B) = \{ a \}$$

$B \rightarrow bB$



Rule 3  
 $\text{Follow}(A) = \text{follow}(B)$

# Example-1: LL(1) parsing

$S \rightarrow aBa$

$B \rightarrow bB \mid \epsilon$

Step 3: Prepare predictive parsing table

NT	First	Follow
S	{a}	{\$}
B	{b, $\epsilon$ }	{a}

NT	Input Symbol		
	a	b	\$
S			
B			

$S \rightarrow aBa$

$a = FIRST(aBa) = \{ a \}$

$M[S,a] = S \rightarrow aBa$

Rule: 2  
 $A \rightarrow \alpha$   
 $a = first(\alpha)$   
 $M[A,a] = A \rightarrow \alpha$

# Example-1: LL(1) parsing

$S \rightarrow aBa$

$B \rightarrow bB \mid \epsilon$

Step 3: Prepare predictive parsing table

NT	First	Follow
S	{a}	{\$}
B	{b, $\epsilon$ }	{a}

NT	Input Symbol		
	a	b	\$
S	S $\rightarrow$ aBa		
B			

$B \rightarrow bB$

$a = FIRST(bB) = \{ b \}$

$M[B, b] = B \rightarrow bB$

Rule: 2  
 $A \rightarrow \alpha$   
 $a = first(\alpha)$   
 $M[A, a] = A \rightarrow \alpha$

# Example-1: LL(1) parsing

$S \rightarrow aBa$

$B \rightarrow bB \mid \epsilon$

Step 3: Prepare predictive parsing table

NT	First	Follow
S	{a}	{\$}
B	{b, $\epsilon$ }	{a}

NT	Input Symbol		
	a	b	\$
S	S $\rightarrow$ aBa		
B		B $\rightarrow$ bB	

$B \rightarrow \epsilon$

$b = FOLLOW(B) = \{ a \}$

$M[B,a] = B \rightarrow \epsilon$

Rule: 3  
 $A \rightarrow \alpha$   
 $b = follow(A)$   
 $M[A,b] = A \rightarrow \alpha$

# Example-2: LL(1) parsing

$S \rightarrow aB \mid \epsilon$

$B \rightarrow bC \mid \epsilon$

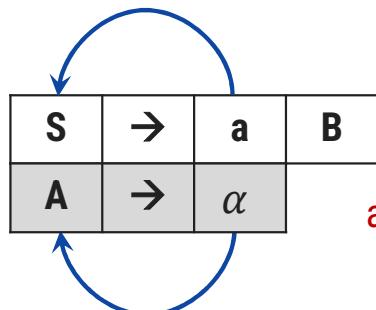
$C \rightarrow cS \mid \epsilon$

Step 1: Not required

Step 2: Compute FIRST

$\text{First}(S)$

$S \rightarrow aB$

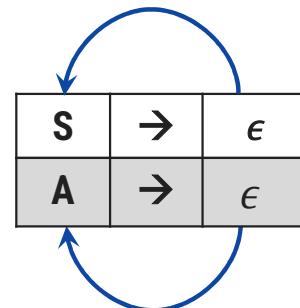


Rule 1  
add  $\alpha$  to  $\text{FIRST}(A)$

$\text{FIRST}(S) = \{ a, \epsilon \}$

NT	First
S	
B	
C	

$S \rightarrow \epsilon$



Rule 2  
add  $\epsilon$  to  $\text{FIRST}(A)$

# Example-2: LL(1) parsing

$S \rightarrow aB \mid \epsilon$

$B \rightarrow bC \mid \epsilon$

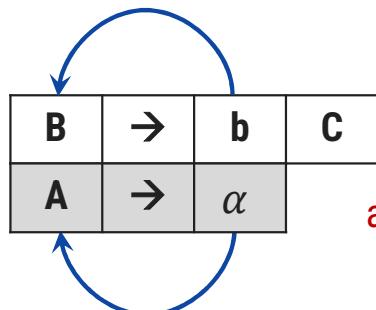
$C \rightarrow cS \mid \epsilon$

Step 1: Not required

Step 2: Compute FIRST

$\text{First}(B)$

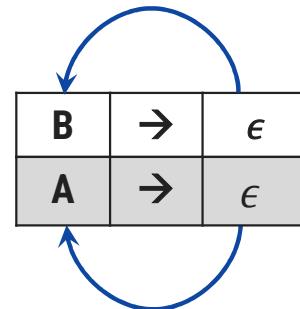
$B \rightarrow bC$



Rule 1  
add  $\alpha$  to  $\text{FIRST}(A)$

$\text{FIRST}(B) = \{ b, \epsilon \}$

$B \rightarrow \epsilon$



Rule 2  
add  $\epsilon$  to  $\text{FIRST}(A)$

NT	First
S	{ a, $\epsilon$ }
B	
C	

# Example-2: LL(1) parsing

$S \rightarrow aB \mid \epsilon$

$B \rightarrow bC \mid \epsilon$

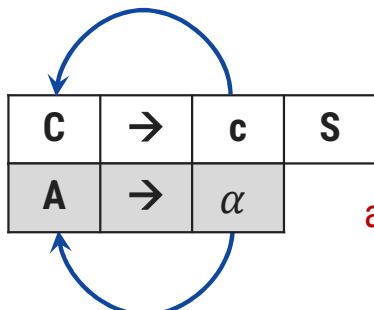
$C \rightarrow cS \mid \epsilon$

Step 1: Not required

Step 2: Compute FIRST

$\text{First}(C)$

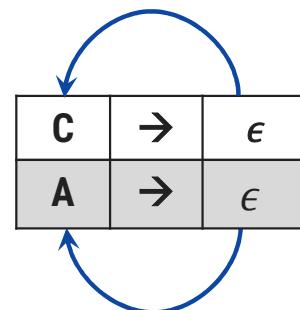
$C \rightarrow cS$



Rule 1  
add  $\alpha$  to  $\text{FIRST}(A)$

$\text{FIRST}(B) = \{c, \epsilon\}$

$C \rightarrow \epsilon$



Rule 2  
add  $\epsilon$  to  $\text{FIRST}(A)$

NT	First
S	{a, $\epsilon$ }
B	{b, $\epsilon$ }
C	

# Example-2: LL(1) parsing

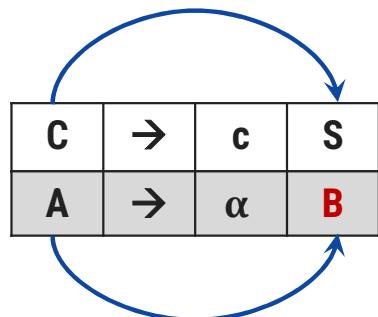
Step 2: Compute FOLLOW

$\text{Follow}(S)$

Rule 1: Place  $\$$  in  $\text{FOLLOW}(S)$

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

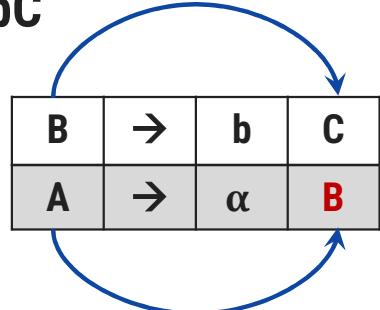
$C \rightarrow cS$



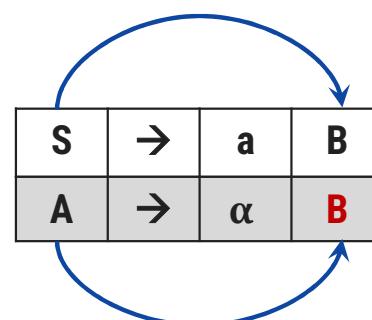
Rule 3  
 $\text{Follow}(A)=\text{follow}(B)$   
 $\text{Follow}(S)=\text{Follow}(C)=\{\$\}$

$S \rightarrow aB | \epsilon$   
 $B \rightarrow bC | \epsilon$   
 $C \rightarrow cS | \epsilon$

$B \rightarrow bC$



Rule 3  
 $\text{Follow}(A)=\text{follow}(B)$   
 $\text{Follow}(C)=\text{Follow}(B)=\{\$\}$



Rule 3  
 $\text{Follow}(A)=\text{follow}(B)$   
 $\text{Follow}(B)=\text{Follow}(S)=\{\$\}$

NT	First	Follow
S	{a, $\epsilon$ }	
B	{b, $\epsilon$ }	
C	{c, $\epsilon$ }	

# Example-2: LL(1) parsing

$S \rightarrow aB \mid \epsilon$

$B \rightarrow bC \mid \epsilon$

$C \rightarrow cS \mid \epsilon$

Step 3: Prepare predictive parsing table

NT	First	Follow
S	{a, $\epsilon$ }	{\$}
B	{b, $\epsilon$ }	{\$}
C	{c, $\epsilon$ }	{\$}

NT	Input Symbol			
	a	b	c	\$
S				
B				
C				

$S \rightarrow aB$

$a = \text{FIRST}(aB) = \{ a \}$

$M[S, a] = S \rightarrow aB$

Rule: 2  
 $A \rightarrow \alpha$   
 $a = \text{first}(\alpha)$   
 $M[A, a] = A \rightarrow \alpha$

# Example-2: LL(1) parsing

$S \rightarrow aB \mid \epsilon$

$B \rightarrow bC \mid \epsilon$

$C \rightarrow cS \mid \epsilon$

Step 3: Prepare predictive parsing table

NT	First	Follow
S	{a}	{\$}
B	{b, $\epsilon$ }	{\$}
C	{c, $\epsilon$ }	{\$}

NT	Input Symbol			
	a	b	c	\$
S	S $\rightarrow$ aB			
B				
C				

$S \rightarrow \epsilon$

b=FOLLOW(S)={ \$ }

M[S,\$]=S  $\rightarrow$   $\epsilon$

Rule: 3  
 $A \rightarrow \alpha$   
 b = follow(A)  
 $M[A,b] = A \rightarrow \alpha$

# Example-2: LL(1) parsing

$S \rightarrow aB \mid \epsilon$

$B \rightarrow bC \mid \epsilon$

$C \rightarrow cS \mid \epsilon$

Step 3: Prepare predictive parsing table

NT	First	Follow
S	{a}	{\$}
B	{b, $\epsilon$ }	{\$}
C	{c, $\epsilon$ }	{\$}

NT	Input Symbol			
	a	b	c	\$
S	S $\rightarrow$ aB			S $\rightarrow$ $\epsilon$
B				
C				

$B \rightarrow bC$

$a = \text{FIRST}(bC) = \{ b \}$

$M[B, b] = B \rightarrow bC$

Rule: 2  
 $A \rightarrow \alpha$   
 $a = \text{first}(\alpha)$   
 $M[A, a] = A \rightarrow \alpha$

# Example-2: LL(1) parsing

$S \rightarrow aB \mid \epsilon$

$B \rightarrow bC \mid \epsilon$

$C \rightarrow cS \mid \epsilon$

Step 3: Prepare predictive parsing table

NT	First	Follow
S	{a}	{\$}
B	{b, $\epsilon$ }	{\$}
C	{c, $\epsilon$ }	{\$}

NT	Input Symbol			
	a	b	c	\$
S	S $\rightarrow$ aB			S $\rightarrow$ $\epsilon$
B		B $\rightarrow$ bC		
C				

$B \rightarrow \epsilon$

b=FOLLOW(B)={ \$ }

M[B,\$]=B  $\rightarrow$   $\epsilon$

Rule: 3  
 $A \rightarrow \alpha$   
 $b = \text{follow}(A)$   
 $M[A,b] = A \rightarrow \alpha$

# Example-2: LL(1) parsing

$S \rightarrow aB \mid \epsilon$

$B \rightarrow bC \mid \epsilon$

$C \rightarrow cS \mid \epsilon$

Step 3: Prepare predictive parsing table

NT	First	Follow
S	{a}	{\$}
B	{b, $\epsilon$ }	{\$}
C	{c, $\epsilon$ }	{\$}

NT	Input Symbol			
	a	b	c	\$
S	S $\rightarrow$ aB			S $\rightarrow$ $\epsilon$
B		B $\rightarrow$ bC		B $\rightarrow$ $\epsilon$
C				

$C \rightarrow cS$

$a = \text{FIRST}(cS) = \{ c \}$

$M[C, c] = C \rightarrow cS$

Rule: 2  
 $A \rightarrow \alpha$   
 $a = \text{first}(\alpha)$   
 $M[A, a] = A \rightarrow \alpha$

## Example-2: LL(1) parsing

$S \rightarrow aB \mid \epsilon$

$B \rightarrow bC \mid \epsilon$

$C \rightarrow cS \mid \epsilon$

Step 3: Prepare predictive parsing table

NT	First	Follow
S	{a}	{\$}
B	{b, $\epsilon$ }	{\$}
C	{c, $\epsilon$ }	{\$}

NT	Input Symbol			
	a	b	c	\$
S	S $\rightarrow$ aB			S $\rightarrow$ $\epsilon$
B		B $\rightarrow$ bB		B $\rightarrow$ $\epsilon$
C			C $\rightarrow$ cS	

$C \rightarrow \epsilon$

b=FOLLOW(C)={ \$ }

M[C,\$]=C  $\rightarrow$   $\epsilon$

Rule: 3  
 $A \rightarrow \alpha$   
 $b = \text{follow}(A)$   
 $M[A,b] = A \rightarrow \alpha$

# Example-3: LL(1) parsing

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Step 1: Remove left recursion

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

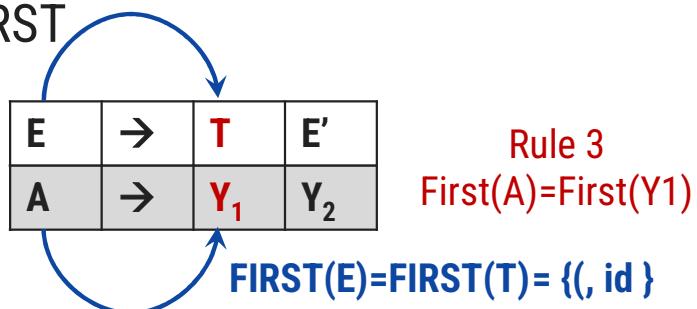
$F \rightarrow (E) \mid id$

# Example-3: LL(1) parsing

Step 2: Compute FIRST

$\text{First}(E)$

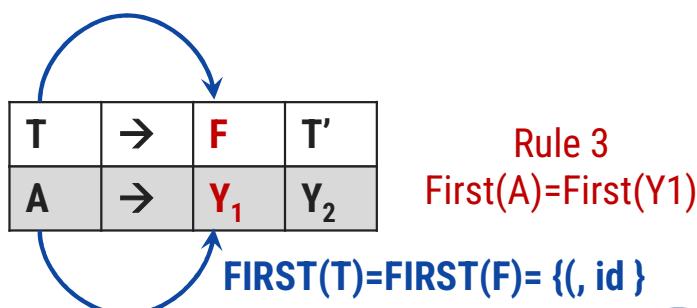
$E \rightarrow TE'$



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

$\text{First}(T)$

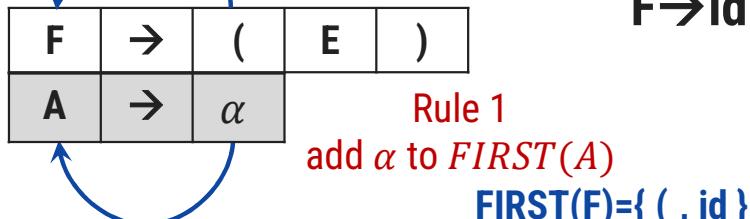
$T \rightarrow FT'$



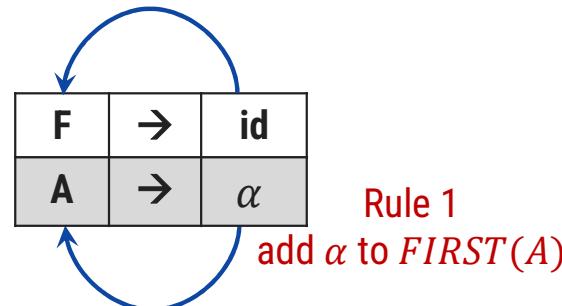
NT	First
$E$	
$E'$	
$T$	
$T'$	
$F$	

$\text{First}(F)$

$F \rightarrow (E)$



$F \rightarrow id$

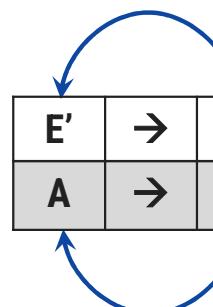


# Example-3: LL(1) parsing

Step 2: Compute FIRST

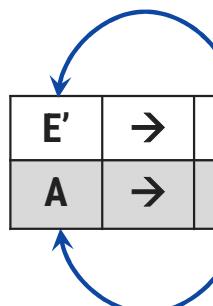
$\text{First}(E')$

$E' \rightarrow +TE'$



Rule 1  
add  $\alpha$  to  $\text{FIRST}(A)$

$E' \rightarrow \epsilon$



Rule 2  
add  $\epsilon$  to  $\text{FIRST}(A)$

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

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

NT	First
E	{ (, id }
E'	
T	{ (, id }
T'	
F	{ (, id }

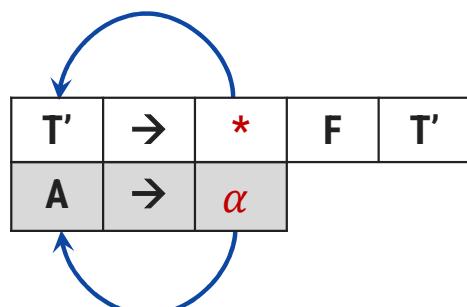
# Example-3: LL(1) parsing

Step 2: Compute FIRST

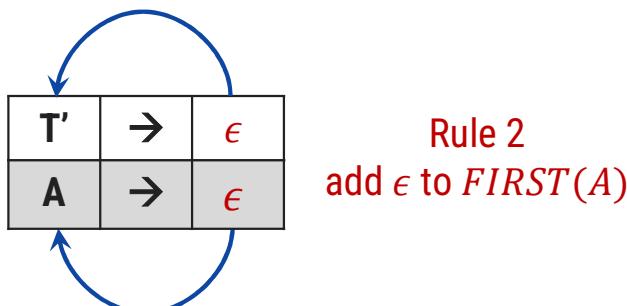
$\text{First}(T')$

$T' \rightarrow *FT'$

$T' \rightarrow \epsilon$



Rule 1  
add  $\alpha$  to  $\text{FIRST}(A)$



Rule 2  
add  $\epsilon$  to  $\text{FIRST}(A)$

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

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

NT	First
E	{ (, id }
E'	{ +, $\epsilon$ }
T	{ (, id }
T'	
F	{ (, id }

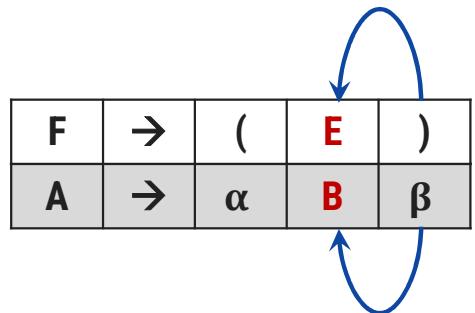
# Example-3: LL(1) parsing

Step 2: Compute FOLLOW

$\text{FOLLOW}(E)$

Rule 1: Place \$ in FOLLOW(E)

$F \rightarrow (E)$



Rule 2

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

NT	First	Follow
E	{ (, id } { +, * }	
E'	{ +, $\epsilon$ }	
T	{ (, id }	
T'	{ *, $\epsilon$ }	
F	{ (, id }	

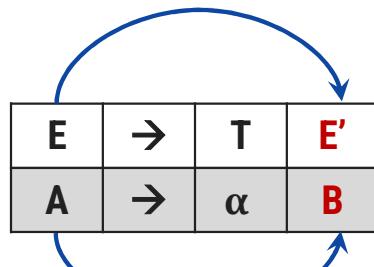
$\text{FOLLOW}(E)=\{ \$, ) \}$

# Example-3: LL(1) parsing

Step 2: Compute FOLLOW

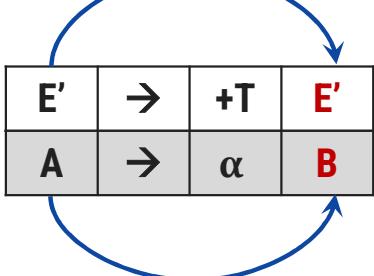
$\text{FOLLOW}(E')$

$E \rightarrow TE'$



Rule 3

$E' \rightarrow +TE'$



Rule 3

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

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

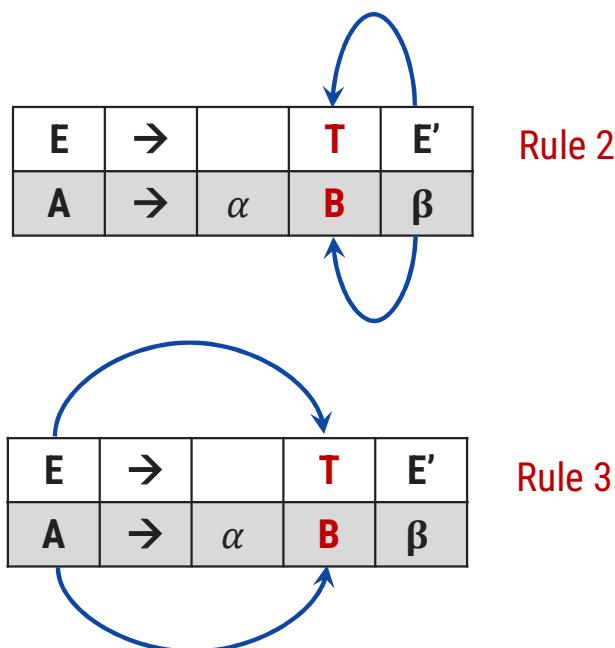
NT	First	Follow
$E$	$\{ (, id \}$	$\{ \$, ) \}$
$E'$	$\{ +, \epsilon \}$	
$T$	$\{ (, id \}$	
$T'$	$\{ *, \epsilon \}$	
$F$	$\{ (, id \}$	

# Example-3: LL(1) parsing

Step 2: Compute FOLLOW

$\text{FOLLOW}(T)$

$E \rightarrow TE'$



$\text{FOLLOW}(T)=\{ +, \$, ) \}$

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

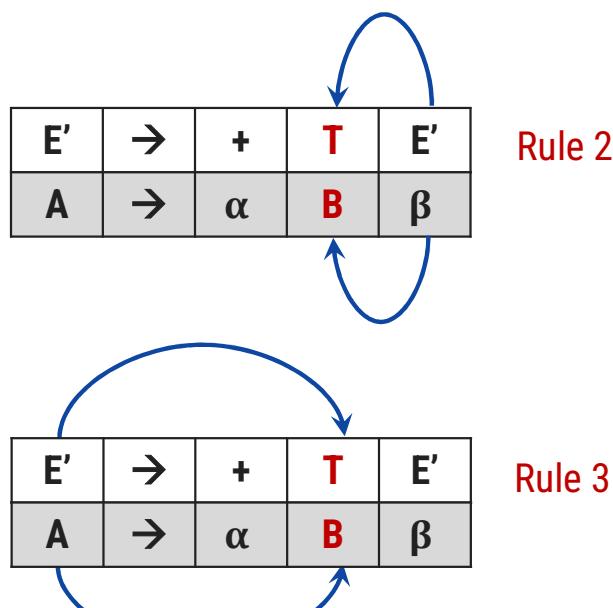
NT	First	Follow
$E$	$\{ (, \text{id} \} \}$	$\{ \$, ) \}$
$E'$	$\{ +, \epsilon \}$	$\{ \$, ) \}$
$T$	$\{ (, \text{id} \} \}$	
$T'$	$\{ *, \epsilon \}$	
$F$	$\{ (, \text{id} \} \}$	

# Example-3: LL(1) parsing

Step 2: Compute FOLLOW

$\text{FOLLOW}(T)$

$E' \rightarrow +TE'$



$$\text{FOLLOW}(T) = \{ +, \$, ) \}$$

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

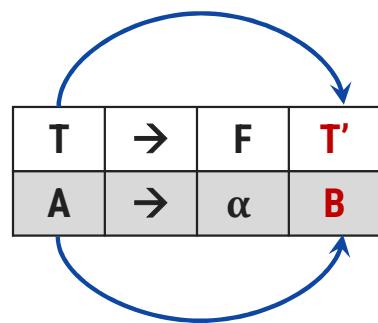
NT	First	Follow
E	{ (, id } { \$, ) }	
E'	{ +, ε }	{ \$, ) }
T	{ (, id }	
T'	{ *, ε }	
F	{ (, id }	

# Example-3: LL(1) parsing

Step 2: Compute FOLLOW

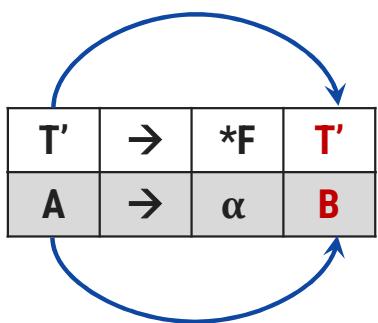
$\text{FOLLOW}(T')$

$T \rightarrow FT'$



Rule 3

$T' \rightarrow *FT'$



Rule 3

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

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

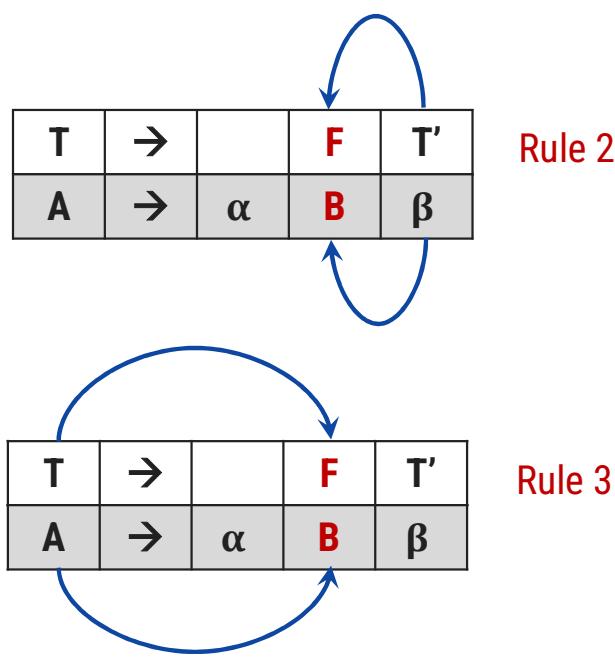
NT	First	Follow
E	{ (,id } { \$, ) }	{ \$, ) }
E'	{ +, $\epsilon$ }	{ \$, ) }
T	{ (,id }	{ +,\$, ) }
T'	{ *, $\epsilon$ }	
F	{ (,id }	

# Example-3: LL(1) parsing

Step 2: Compute FOLLOW

$\text{FOLLOW}(F)$

$T \rightarrow FT'$



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

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

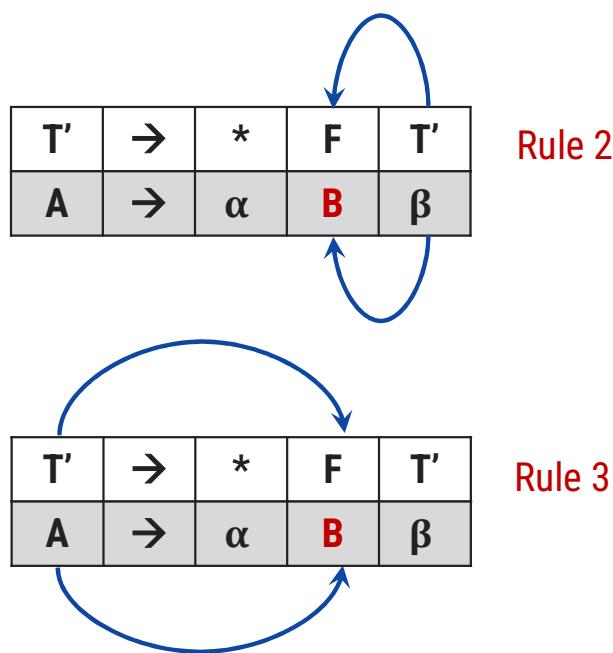
NT	First	Follow
E	{ (, id } { \$, ) }	{ \$, ) }
E'	{ +, $\epsilon$ }	{ \$, ) }
T	{ (, id }	{ +, \$, ) }
T'	{ *, $\epsilon$ }	{ +, \$, ) }
F	{ (, id }	

# Example-3: LL(1) parsing

Step 2: Compute FOLLOW

$\text{FOLLOW}(F)$

$T' \rightarrow *FT'$



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

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

NT	First	Follow
$E$	$\{ (, \text{id} \} \}$	$\{ \$, ) \}$
$E'$	$\{ +, \epsilon \}$	$\{ \$, ) \}$
$T$	$\{ (, \text{id} \} \}$	$\{ +, \$, ) \}$
$T'$	$\{ *, \epsilon \}$	$\{ +, \$, ) \}$
$F$	$\{ (, \text{id} \} \}$	

# Example-3: LL(1) parsing

Step 3: Construct predictive parsing table

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

NT	Input Symbol					
	id	+	*	(	)	\$
E						
E'						
T						
T'						
F						

$E \rightarrow TE'$

$a = FIRST(TE') = \{ (, id \}$

$M[E, ()] = E \rightarrow TE'$

$M[E, id] = E \rightarrow TE'$

Rule: 2  
 $A \rightarrow \alpha$   
 $a = first(\alpha)$   
 $M[A, a] = A \rightarrow \alpha$

NT	First	Follow
E	{ (, id }	{ \$, ) }
E'	{ +, ε }	{ \$, ) }
T	{ (, id }	{ +, \$, ) }
T'	{ *, ε }	{ +, \$, ) }
F	{ (, id }	{ *, +, \$, ) }

# Example-3: LL(1) parsing

Step 3: Construct predictive parsing table

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

NT	Input Symbol					
	id	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'						
T						
T'						
F						

$E' \rightarrow +TE'$

$a = FIRST(+TE') = \{ + \}$

$M[E',+] = E' \rightarrow +TE'$

Rule: 2  
 $A \rightarrow \alpha$   
 $a = first(\alpha)$   
 $M[A,a] = A \rightarrow \alpha$

NT	First	Follow
E	{ (,id }	{ \$,) }
E'	{ +, \epsilon }	{ \$,) }
T	{ (,id }	{ +,\$,) }
T'	{ *, \epsilon }	{ +,\$,) }
F	{ (,id }	{ *,+, \$,) }

# Example-3: LL(1) parsing

Step 3: Construct predictive parsing table

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

NT	Input Symbol					
	id	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$				
T						
T'						
F						

$E' \rightarrow \epsilon$

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

$M[E',\$]=E' \rightarrow \epsilon$

$M[E',)]]=E' \rightarrow \epsilon$

Rule: 3  
 $A \rightarrow \alpha$   
 $b = \text{follow}(A)$   
 $M[A,b] = A \rightarrow \alpha$

NT	First	Follow
E	{ (,id }	{ \$,) }
E'	{ +, \epsilon }	{ \$,) }
T	{ (,id }	{ +,\$,) }
T'	{ *, \epsilon }	{ +,\$,) }
F	{ (,id }	{ *,+, \$,) }

# Example-3: LL(1) parsing

Step 3: Construct predictive parsing table

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

NT	Input Symbol					
	id	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T						
T'						
F						

$T \rightarrow FT'$

$a = FIRST(FT') = \{ (, id \}$

$M[T, ()] = T \rightarrow FT'$

$M[T, id] = T \rightarrow FT'$

Rule: 2  
 $A \rightarrow \alpha$   
 $a = first(\alpha)$   
 $M[A, a] = A \rightarrow \alpha$

NT	First	Follow
E	{ (, id }	{ \$, ) }
E'	{ +, \epsilon }	{ \$, ) }
T	{ (, id }	{ +, \$, ) }
T'	{ *, \epsilon }	{ +, \$, ) }
F	{ (, id }	{ *, +, \$, ) }

# Example-3: LL(1) parsing

Step 3: Construct predictive parsing table

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

NT	Input Symbol					
	id	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'						
F						

$T' \rightarrow *FT'$

$a = FIRST(*FT') = \{ *\}$

$M[T', *] = T' \rightarrow *FT'$

Rule: 2  
 $A \rightarrow \alpha$   
 $a = first(\alpha)$   
 $M[A, a] = A \rightarrow \alpha$

NT	First	Follow
E	{ (, id }	{ \$, ) }
E'	{ +, ε }	{ \$, ) }
T	{ (, id }	{ +, \$, ) }
T'	{ *, ε }	{ +, \$, ) }
F	{ (, id }	{ *, +, \$, ) }

# Example-3: LL(1) parsing

Step 3: Construct predictive parsing table

NT	Input Symbol					
	id	+	*	(	)	\$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'			T' → *FT'			
F						

$T' \rightarrow \epsilon$

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

Rule: 3

A → α

b = follow(A)

M[A,b] = A → α

M[T',+] = T' → ε

M[T',\\$] = T' → ε

M[T',)] = T' → ε

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

NT	First	Follow
E	{ (, id } { \$,) }	
E'	{ +, ε } { \$,) }	
T	{ (, id } { +,\$,) }	
T'	{ *, ε } { +,\$,) }	
F	{ (, id } { *, +,\$,) }	

# Example-3: LL(1) parsing

Step 3: Construct predictive parsing table

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

NT	Input Symbol					
	id	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F						

Rule: 2

$A \rightarrow \alpha$

a = first( $\alpha$ )

$M[A,a] = A \rightarrow \alpha$

$F \rightarrow (E)$

a=FIRST((E))={ ( ) }

$M[F,()] = F \rightarrow (E)$

NT	First	Follow
E	{ (,id }	{ \$,) }
E'	{ +, \epsilon }	{ \$,) }
T	{ (,id }	{ +,\$,) }
T'	{ *, \epsilon }	{ +,\$,) }
F	{ (,id }	{ *,+, \$,) }

# Example-3: LL(1) parsing

Step 3: Construct predictive parsing table

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

NT	Input Symbol					
	id	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F				$F \rightarrow (E)$		

Rule: 2

$A \rightarrow \alpha$

a = first( $\alpha$ )

$M[A,a] = A \rightarrow \alpha$

$F \rightarrow id$

a=FIRST(id)= { id }

$M[F,id] = F \rightarrow id$

NT	First	Follow
E	{ (,id }	{ \$,) }
E'	{ +, \epsilon }	{ \$,) }
T	{ (,id }	{ +,\$,) }
T'	{ *, \epsilon }	{ +,\$,) }
F	{ (,id }	{ *,+, \$,) }

# Example-3: LL(1) parsing

- ▶ Step 4: Make each undefined entry of table be Error

NT	Input Symbol					
	id	+	*	(	)	\$
E	$E \rightarrow TE'$	Error	Error	$E \rightarrow TE'$	Error	Error
E'	Error	$E' \rightarrow +TE'$	Error	Error	$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	Error	Error	$T \rightarrow FT'$	Error	Error
T'	Error	$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$	Error	$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$	Error	Error	$F \rightarrow (E)$	Error	Error

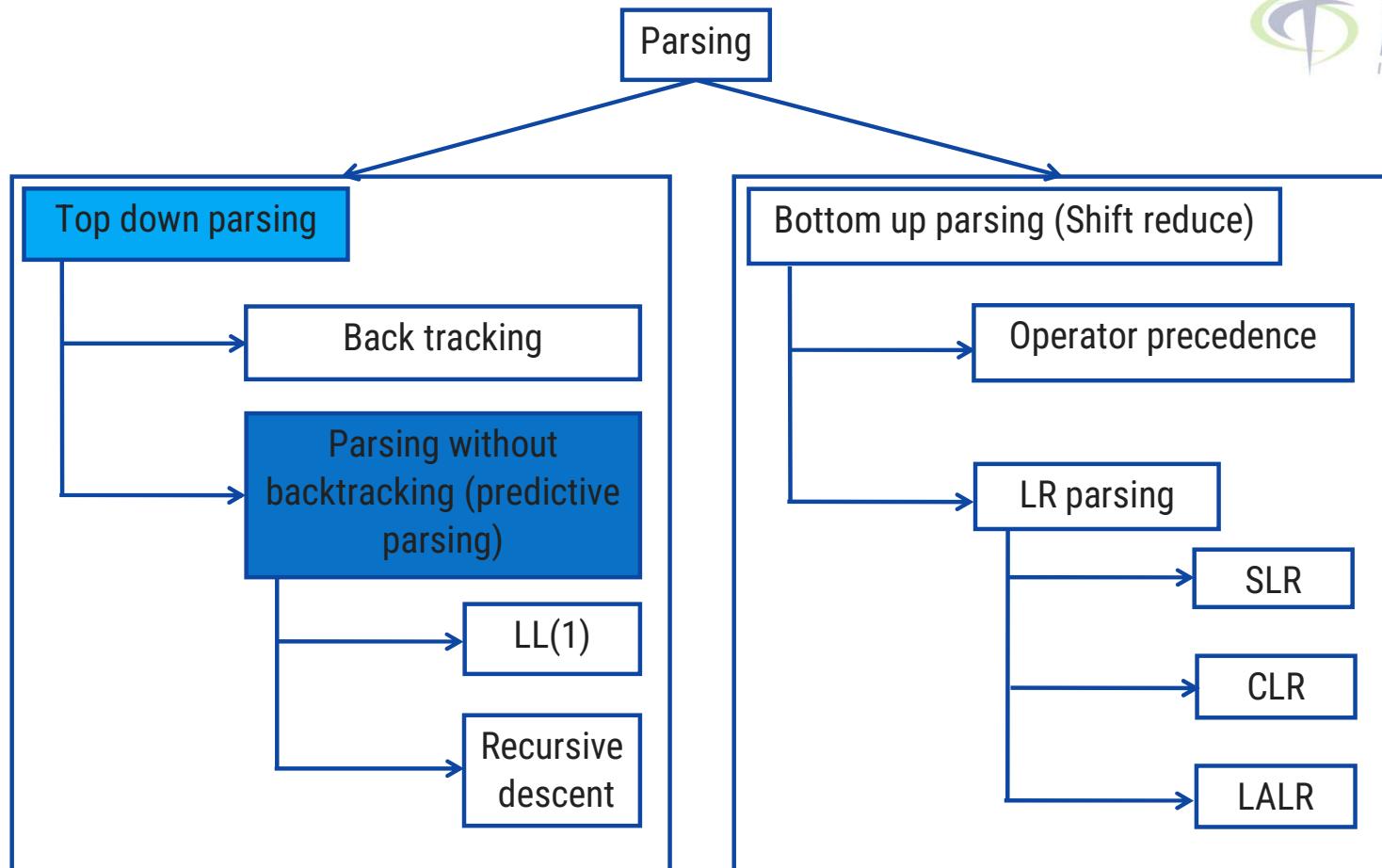
# Example-3: LL(1) parsing

Step 4: Parse the string : id + id \* id \$

STACK	INPUT	OUTPUT
E\$	id+id*id\$	

NT	Input Symbol					
	id	+	*	(	)	\$
E	$E \rightarrow TE'$	Error	Error	$E \rightarrow TE'$	Error	Error
E'	Error	$E' \rightarrow +TE'$	Error	Error	$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	Error	Error	$T \rightarrow FT'$	Error	Error
T'	Error	$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$	Error	$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$	Error	Error	$F \rightarrow (E)$	Error	Error

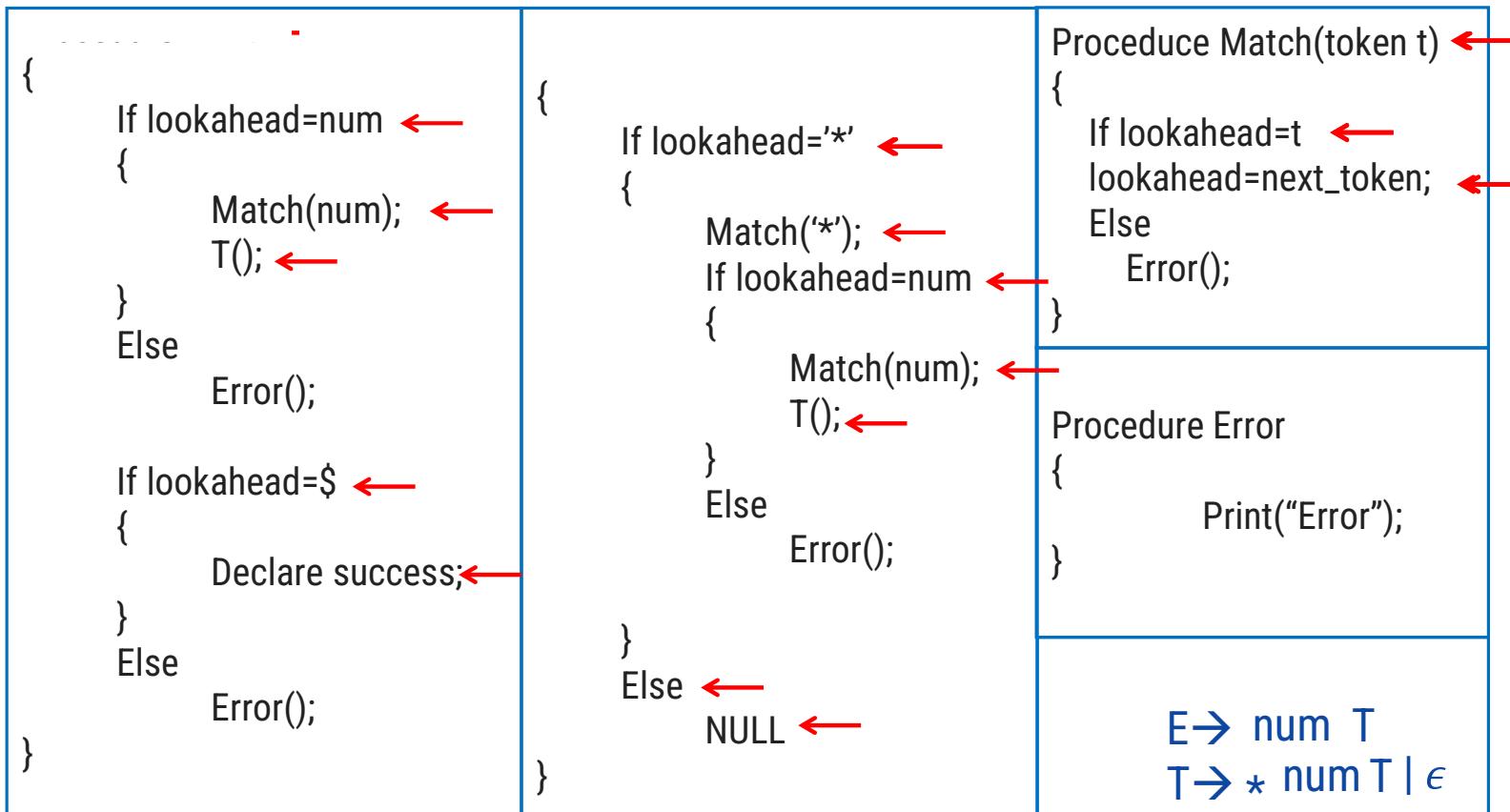

# Parsing methods



# Recursive descent parsing

- ▶ A top down parsing that executes a set of recursive procedure to process the input without backtracking is called recursive descent parser.
- ▶ There is a procedure for each non terminal in the grammar.
- ▶ Consider RHS of any production rule as definition of the procedure.
- ▶ As it reads expected input symbol, it advances input pointer to next position.

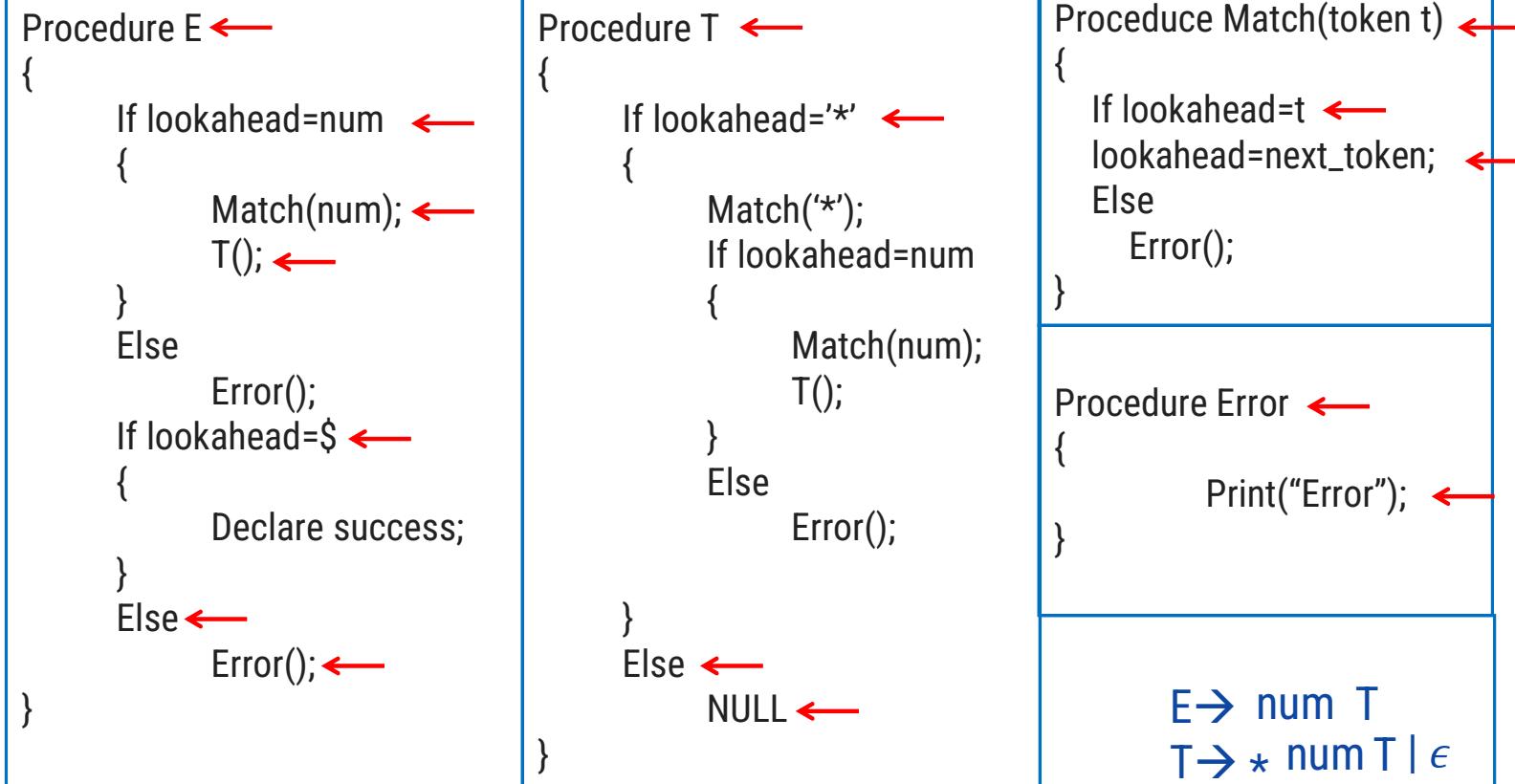
# Example: Recursive descent parsing



3	*	4	\$
---	---	---	----

Success

# Example: Recursive descent parsing



3 \* 4 \$

Success

3 4 \* \$

Error

# Handle & Handle pruning

# Handle & Handle pruning

- ▶ **Handle:** A “handle” of a string is a substring of the string that matches the right side of a production, and whose reduction to the non terminal of the production is one step along the **reverse of rightmost derivation**.
- ▶ **Handle pruning:** The process of discovering a handle and **reducing it to appropriate left hand side non terminal** is known as handle pruning.

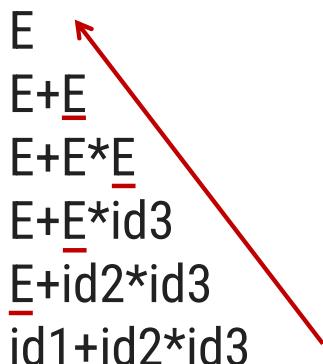
$E \rightarrow E+E$

$E \rightarrow E^*E$       String: id1+id2\*id3

$E \rightarrow id$

**Rightmost Derivation**

E  
E+E  
E+E\*E  
E+E\*id3  
E+id2\*id3  
id1+id2\*id3



Right sentential form	Handle	Production
id1+id2*id3		

# Shift reduce parser

- ▶ The shift reduce parser performs following basic operations:
  1. **Shift:** Moving of the symbols from **input buffer onto the stack**, this action is called shift.
  2. **Reduce:** If handle appears on the top of the stack then **reduction of it by appropriate rule** is done. This action is called reduce action.
  3. **Accept:** If **stack contains start symbol only and input buffer is empty** at the same time then that action is called accept.
  4. **Error:** A situation in which parser **cannot either shift or reduce the symbols**, it cannot even perform accept action then it is called error action.

# Example: Shift reduce parser

## Grammar:

$E \rightarrow E + T \mid T$

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

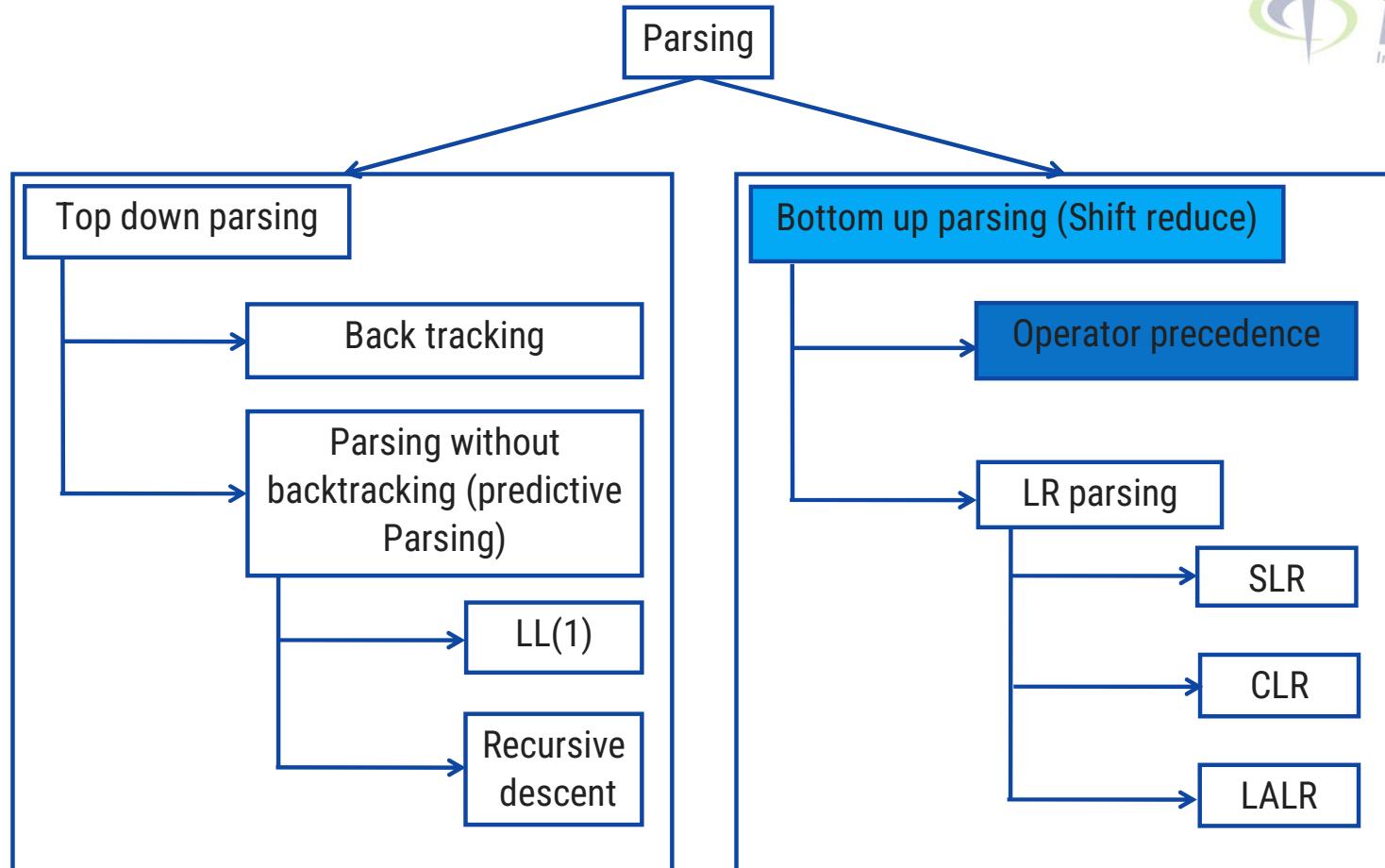
F → id

## String: id+id\*id

# Viable Prefix

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

# Parsing Methods



# Operator precedence parsing

# Operator precedence parsing

- ▶ **Operator Grammar:** A Grammar in which there is no  $\epsilon$  in RHS of any production or no adjacent non terminals is called operator grammar.
- ▶ Example:  $E \rightarrow EAE \mid (E) \mid id$   
 $A \rightarrow + \mid * \mid -$
- ▶ Above grammar is not operator grammar because right side **EAE** has consecutive non terminals.
- ▶ In operator precedence parsing we define following disjoint relations:

Relation	Meaning
$a < b$	a "yields precedence to" b
$a = b$	a "has the same precedence as" b
$a \cdot > b$	a "takes precedence over" b

# Precedence & associativity of operators

Operator	Precedence	Associative
$\uparrow$	1	right
$*, /$	2	left
$+, -$	3	left

# Steps of operator precedence parsing

1. Find Leading and trailing of non terminal
2. Establish relation
3. Creation of table
4. Parse the string

# Leading & Trailing

**Leading:-** Leading of a non terminal is the **first terminal or operator** in production of that non terminal.

**Trailing:-** Trailing of a non terminal is the **last terminal or operator** in production of that non terminal.

Example:

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

Non terminal	Leading	Trailing
E		
T		
F		

# Rules to establish a relation

1. For  $a \doteq b$ ,  $\Rightarrow aAb$ , where  $A$  is  $\epsilon$  or a single non terminal [e.g : (E)]
2.  $a < \cdot b \Rightarrow Op . NT$  then  $Op < . Leading(NT)$  [e.g : +T]
3.  $a \cdot > b \Rightarrow NT . Op$  then  $(Trailing(NT)) \cdot > Op$  [e.g : E+]
4.  $\$ < . Leading$  (start symbol)
5. Trailing (start symbol)  $\cdot > \$$

# Example: Operator precedence parsing

## Step 1: Find Leading & Trailing of NT

Nonterminal	Leading	Trailing
E	{+, *, id}	{+, *, id}
T	{*, id}	{*, id}
F	{id}	{id}

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

## Step 2: Establish Relation

a < $\cdot$ b

$Op \cdot NT$	$Op < \cdot Leading(NT)$
$+T$	$+ < \cdot \{*, id\}$
$*F$	$* < \cdot \{id\}$

## Step3: Creation of Table

	+	*	id	\$
+				
*				
id				
\$				

# Example: Operator precedence parsing

## Step 1: Find Leading & Trailing of NT

Nonterminal	Leading	Trailing
E	{+,*,id}	{+,*,id}
T	{*,id}	{*,id}
F	{id}	{id}

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

## Step2: Establish Relation

a → b

$$\begin{array}{l|l} NT \cdot Op & (Trailing(NT)) \cdot > Op \\ \hline E + & \{+, *, id\} \cdot > + \\ T * & \{*, id\} \cdot > * \end{array}$$

## Step3: Creation of Table

	+	*	id	\$
+		<·	<·	
*			<·	
id				
\$				

# Example: Operator precedence parsing

## Step 1: Find Leading & Trailing of NT

Nonterminal	Leading	Trailing
E	{+,*,id}	{+,*,id}
T	{*,id}	{*,id}
F	{id}	{id}

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

## Step 2: Establish Relation

\$<· Leading (start symbol)

\$ <· {+,\*, id}

Trailing (start symbol) ·> \$

{+,\*, id} ·> \$

## Step 3: Creation of Table

	+	*	id	\$
+	·>	<·	<·	
*	·>	·>	<·	
id	·>	·>		
\$				

# Example: Operator precedence parsing

## Step 4: Parse the string using precedence table

Assign precedence operator between terminals

String: id+id\*id

\$ id+id\*id \$

\$ < id+id\*id \$

\$ < id > + id\*id \$

\$ < id > + < id\*id \$

\$ < id > + < id > \*id \$

\$ < id > + < id > \* < id \$

\$ < id > + < id > \* < id > \$

	+	*	id	\$
+	>	<	<	>
*	>	>	<	>
id	>	>		>
\$	<	<	<	

# Example: Operator precedence parsing

## Step 4: Parse the string using precedence table

1. Scan the input string until first  $\cdot >$  is encountered.
2. Scan backward until  $\cdot \cdot$  is encountered.
3. The handle is string between  $\cdot \cdot$  and  $\cdot >$

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

$\$ \cdot \cdot Id \cdot > + \cdot \cdot Id \cdot > * \cdot \cdot Id \cdot > \$$	
$\$ F + \cdot \cdot Id \cdot > * \cdot \cdot Id \cdot > \$$	
$\$ F + F * \cdot \cdot Id \cdot > \$$	
$\$ F + F * F \$$	
$\$ E + T * F \$$	
$\$ + * \$$	
$\$ \cdot \cdot + \cdot \cdot * \cdot > \$$	
$\$ \cdot \cdot + \cdot > \$$	
$\$ \$$	

	$+$	$*$	$id$	$\$$
$+$	$\cdot >$	$\cdot \cdot$	$\cdot \cdot$	$\cdot >$
$*$	$\cdot >$	$\cdot >$	$\cdot \cdot$	$\cdot >$
$id$	$\cdot >$	$\cdot >$		$\cdot >$
$\$$	$\cdot \cdot$	$\cdot \cdot$	$\cdot \cdot$	

# Operator precedence function

# Operator precedence function

## Algorithm for constructing precedence functions

1. Create functions  $f_a$  and  $g_a$  for each  $a$  that is terminal or \$.
2. Partition the symbols in as many as groups possible, in such a way that  $f_a$  and  $g_b$  are in the same group if  $a = b$ .
3. Create a directed graph whose nodes are in the groups, next for each symbols  $a$  and  $b$  do:
  - a) if  $a < \cdot b$ , place an edge from the group of  $g_b$  to the group of  $f_a$
  - b) if  $a \cdot > b$ , place an edge from the group of  $f_a$  to the group of  $g_b$
4. If the constructed graph has a cycle then no precedence functions exist. When there are no cycles collect the length of the longest paths from the groups of  $f_a$  and  $g_b$  respectively.

# Operator precedence function

1. Create functions  $f_a$  and  $g_a$  for each  $a$  that is terminal or \$.

$E \rightarrow E+T \mid T$   
 $T \rightarrow T^*F \mid F$   
 $F \rightarrow id$

$$a = \{+, *, id\} \text{ or } \$$$

$f_+$

$f_*$

$f_{id}$

$f_{\$}$

$g_+$

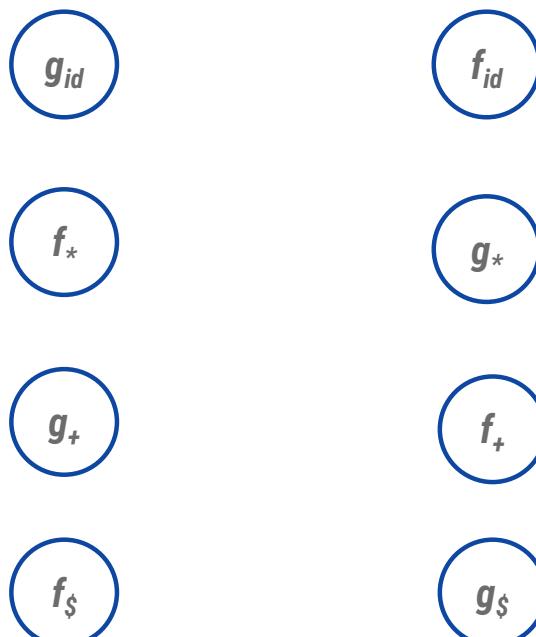
$g_*$

$g_{id}$

$g_{\$}$

# Operator precedence function

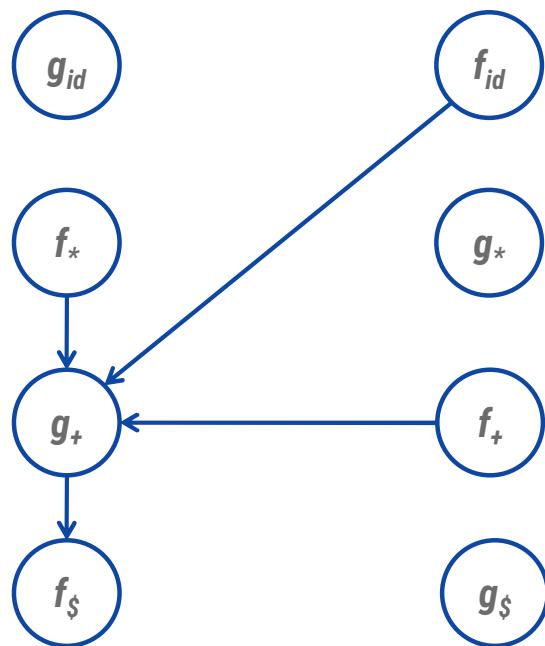
- ▶ Partition the symbols in as many as groups possible, in such a way that  $f_a$  and  $g_b$  are in the same group if  $a \doteq b$ .



	+	*	id	\$
+	>	<·	<·	>
*	>	>	<·	>
id	>	>		>
\$	<·	<·	<·	

# Operator precedence function

3. if  $a < \cdot b$ , place an edge from the group of  $g_b$  to the group of  $f_a$   
 if  $a \cdot > b$ , place an edge from the group of  $f_a$  to the group of  $g_b$

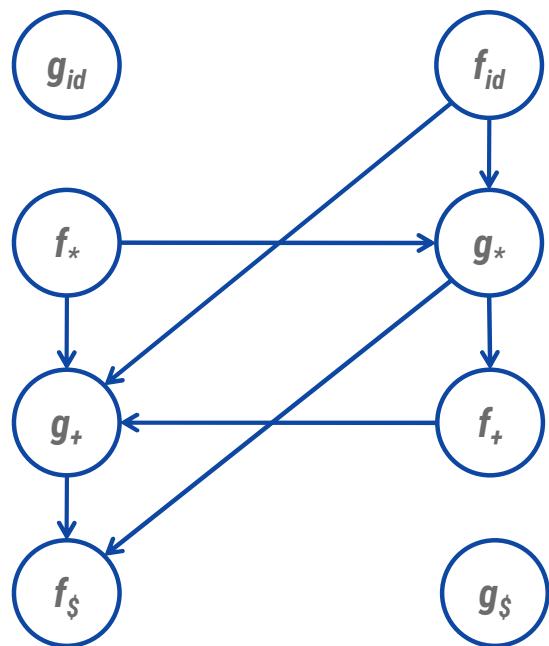


		g			
		+	*	id	\$
f	+	>	<	<	>
	*	>	>	<	>
	id	>	>		>
	\$	<	<	<	

$$\begin{array}{ll}
 f_+ \cdot > g_+ & f_+ \rightarrow g_+ \\
 f_* \cdot > g_+ & f_* \rightarrow g_+ \\
 f_{id} \cdot > g_+ & f_{id} \rightarrow g_+ \\
 f_\$ < \cdot g_+ & f_\$ \leftarrow g_+
 \end{array}$$

# Operator precedence function

3. if  $a \cdot\!< b$ , place an edge from the group of  $g_b$  to the group of  $f_a$   
 if  $a \cdot\!> b$ , place an edge from the group of  $f_a$  to the group of  $g_b$

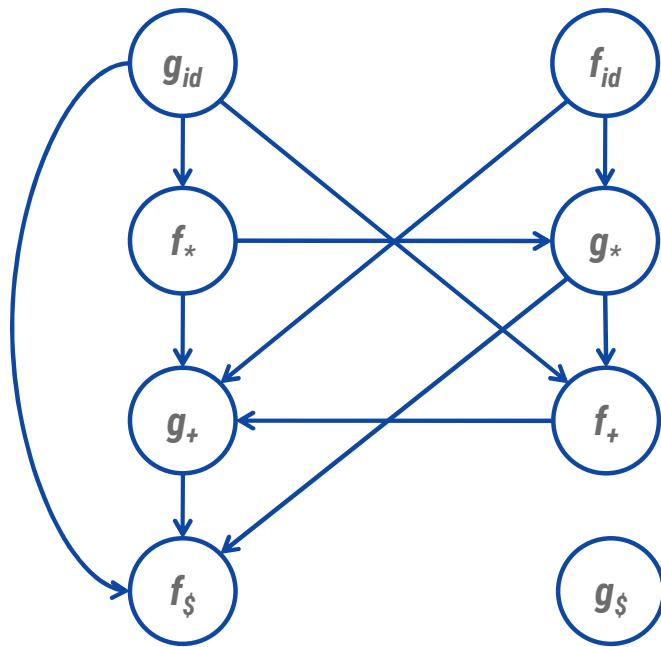


		g			
		+	*	id	\$
f	+	>	<	<	>
	*	>	>	<	>
	id	>	>		>
	\$	<	<	<	

$$\begin{array}{ll}
 f_+ \cdot\!< g_* & f_+ \leftarrow g_* \\
 f_* \cdot\!> g_* & f_* \rightarrow g_* \\
 f_{id} \cdot\!> g_* & f_{id} \rightarrow g_* \\
 f_\$ \cdot\!< g_* & f_\$ \leftarrow g_*
 \end{array}$$

# Operator precedence function

3. if  $a \cdot\!< b$ , place an edge from the group of  $g_b$  to the group of  $f_a$   
 if  $a \cdot\!> b$ , place an edge from the group of  $f_a$  to the group of  $g_b$

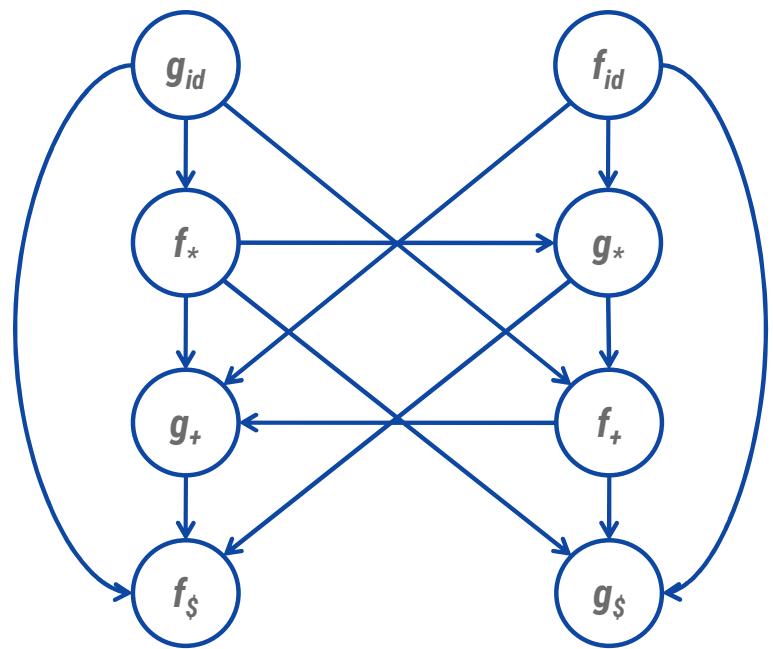


		g			
		+	*	id	\$
f	+	>	<	<	>
	*	>	>	<	>
id	>	>			>
\$	<	<	<	<	

$$\begin{array}{ll}
 f_+ < \cdot g_{id} & f_+ \leftarrow g_{id} \\
 f_* < \cdot g_{id} & f_* \leftarrow g_{id} \\
 f_\$ < \cdot g_{id} & f_\$ \leftarrow g_{id}
 \end{array}$$

# Operator precedence function

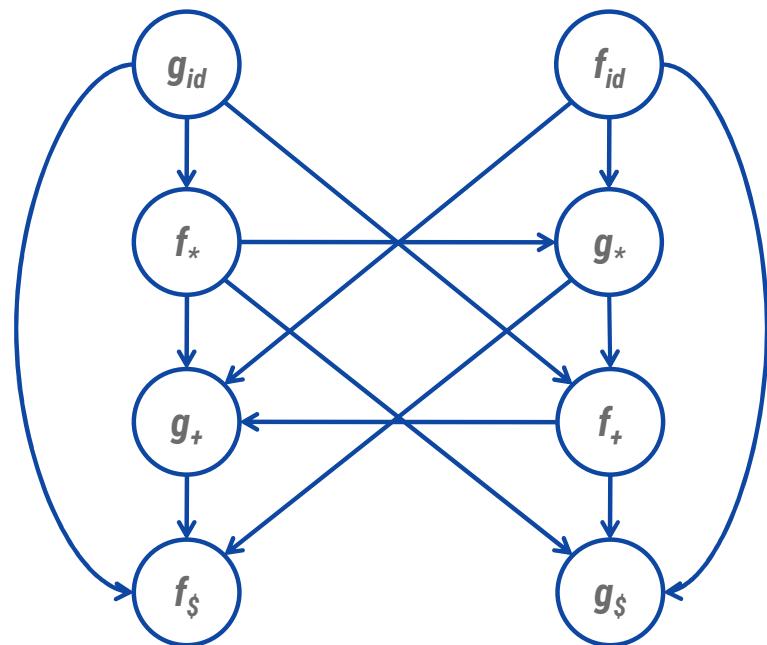
3. if  $a < \cdot b$ , place an edge from the group of  $g_b$  to the group of  $f_a$   
 if  $a \cdot > b$ , place an edge from the group of  $f_a$  to the group of  $g_b$



		g			
		+	*	id	\$
f		>	<	<	>
+		>	<	<	>
*		>	>	<	>
id		>	>		>
\$		<	<	<	

$$\begin{array}{ll}
 f_+ < \cdot g_\$ & f_+ \rightarrow g_\$ \\
 f_* < \cdot g_\$ & f_* \rightarrow g_\$ \\
 f_{id} < \cdot g_\$ & f_{id} \rightarrow g_\$ \\
 \end{array}$$

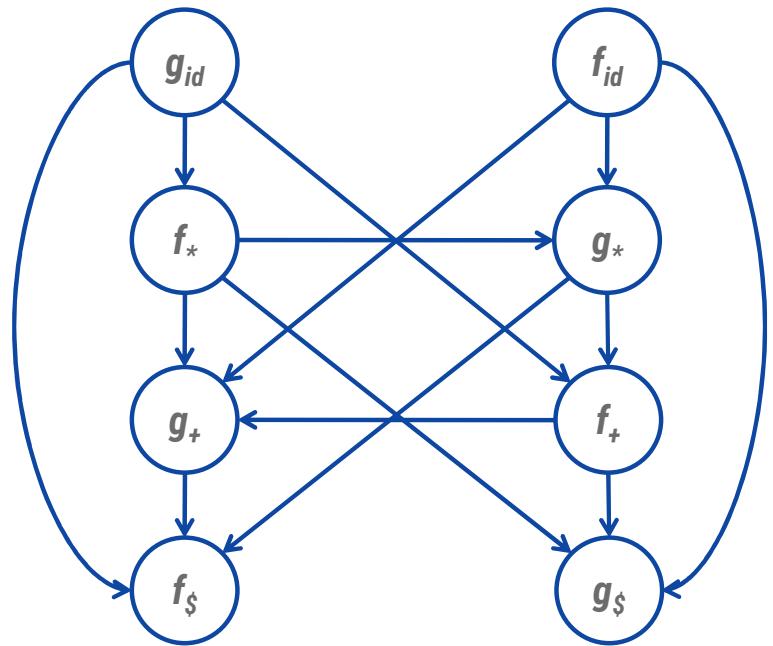
# Operator precedence function



	+	*	id	\$
f				
g				

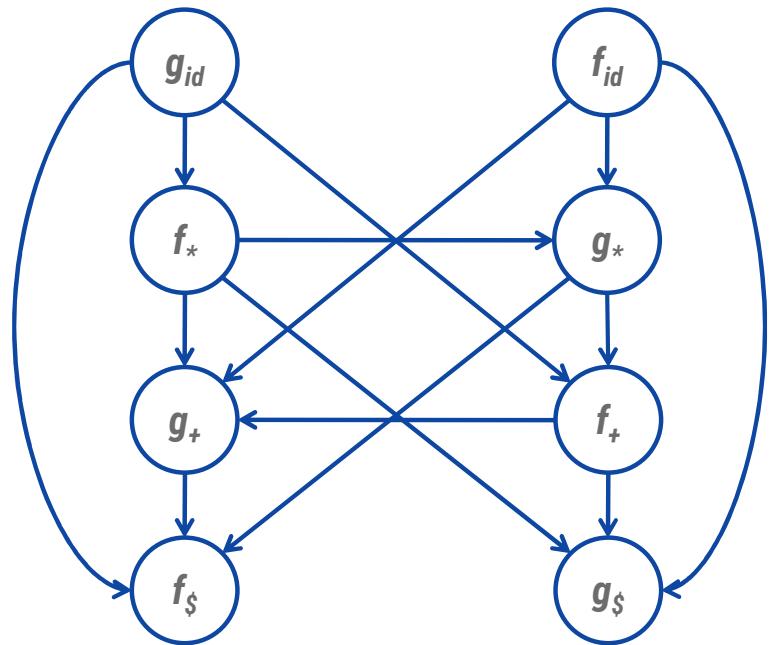
4. If the constructed graph has a cycle then no precedence functions exist. When there are no cycles collect the length of the longest paths from the groups of  $f_a$  and  $g_b$  respectively.

# Operator precedence function



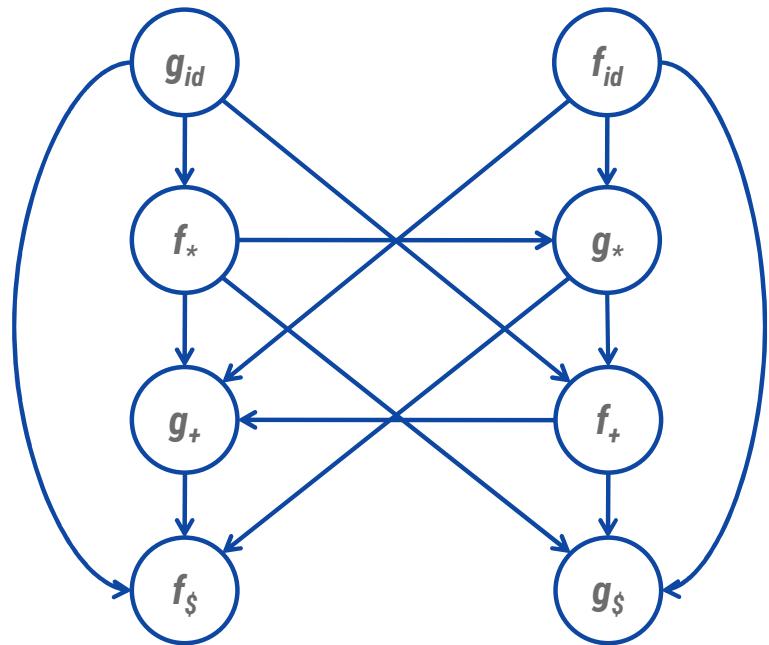
	$+$	$*$	$id$	$\$$
$f$	2			
$g$				

# Operator precedence function



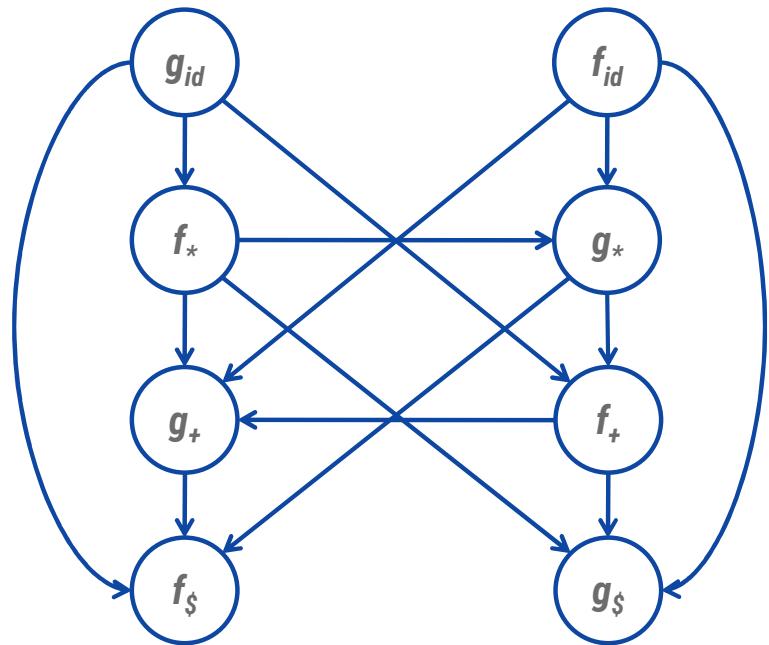
	+	*	id	\$
f	2			
g	1			

# Operator precedence function



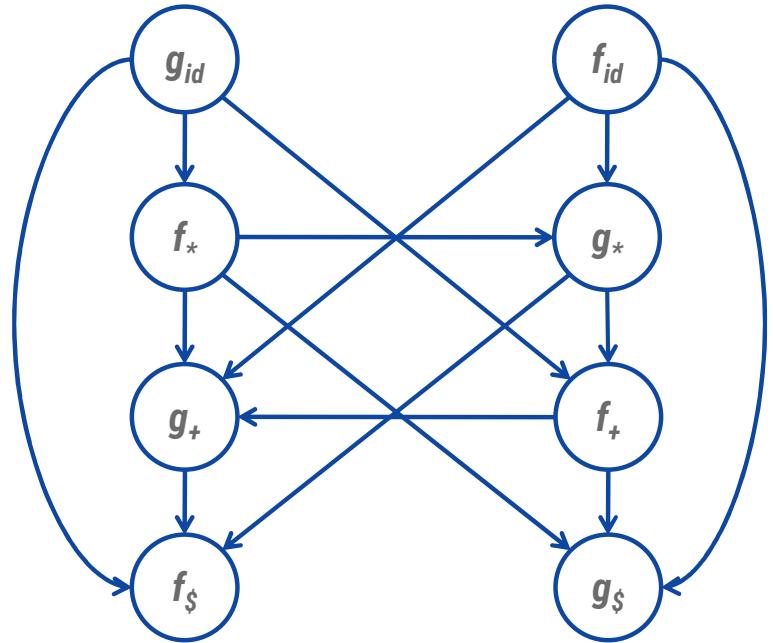
	+	*	id	\$
f	2	4		
g	1			

# Operator precedence function



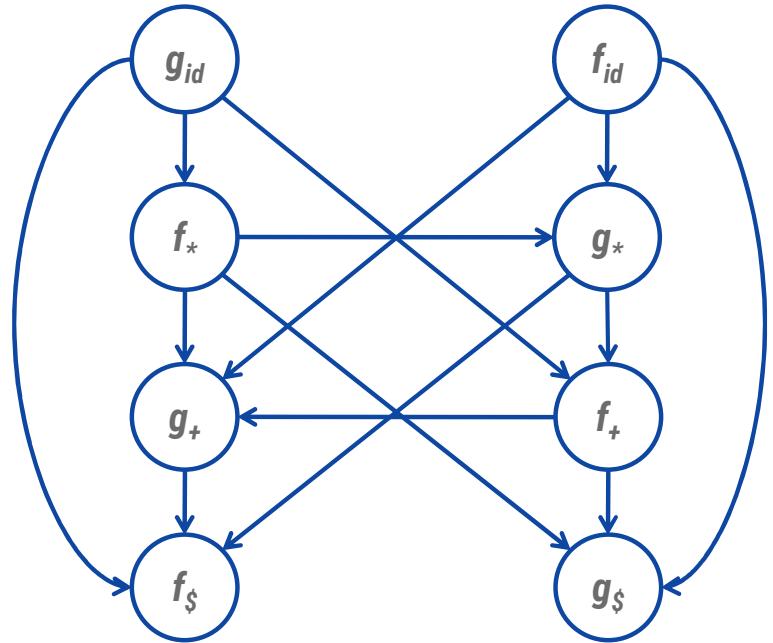
	+	*	id	\$
f	2	4		
g	1	3		

# Operator precedence function



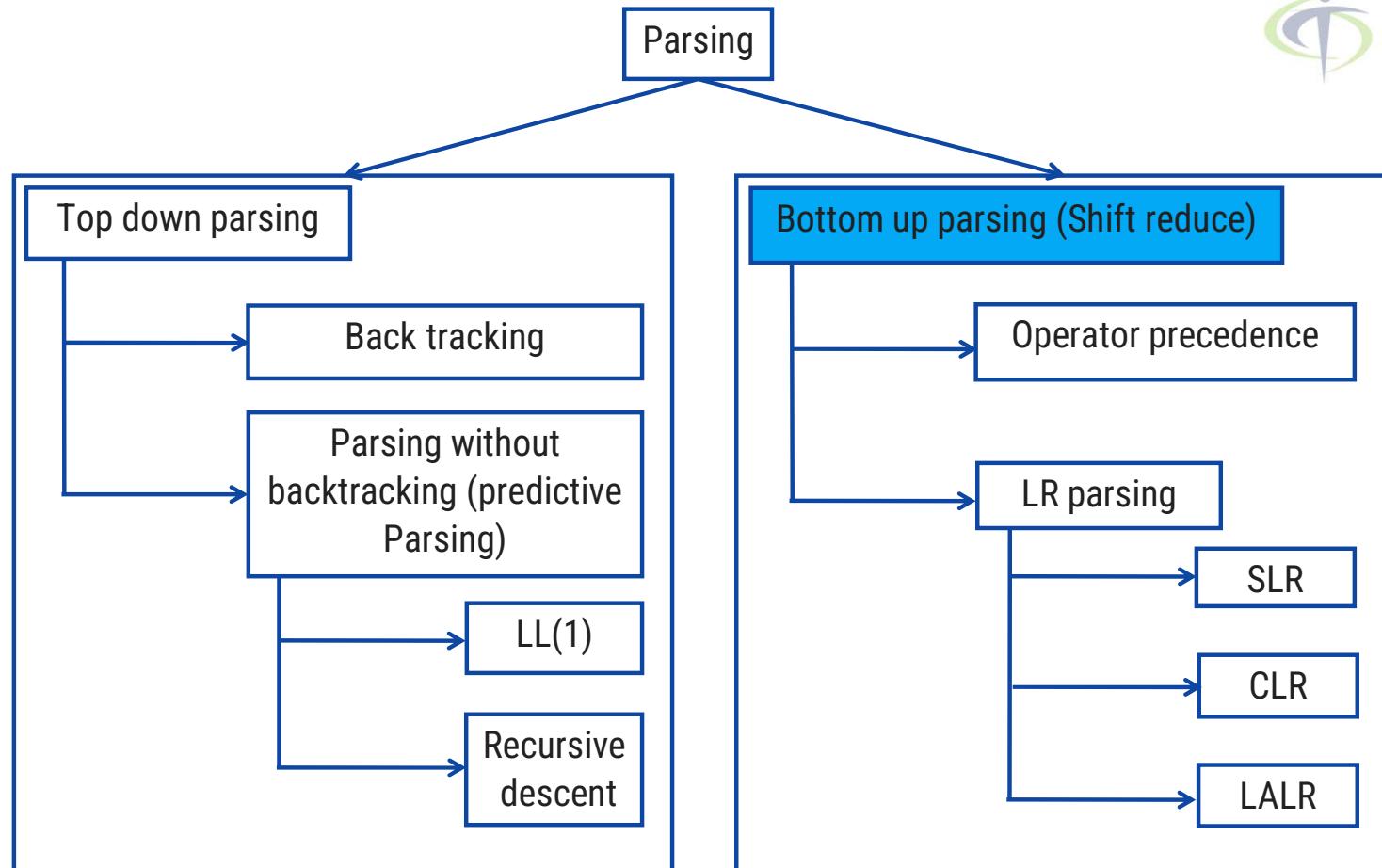
	+	*	id	\$
f	2	4	4	
g	1	3		

# Operator precedence function



	+	*	id	\$
f	2	4	4	
g	1	3	5	

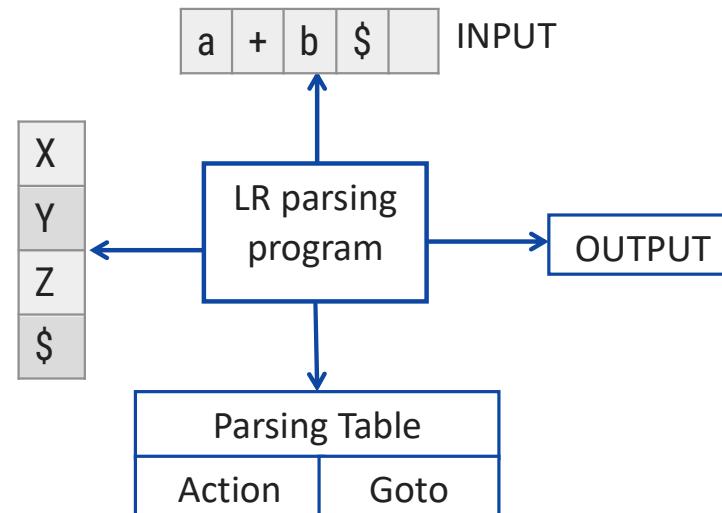
# Parsing Methods



# Introduction to LR Parser

# LR parser

- ▶ LR parsing is most efficient method of bottom up parsing which can be used to parse large class of context free grammar.
- ▶ The technique is called LR(k) parsing:
  1. The “L” is for **left to right** scanning of input symbol,
  2. The “R” for constructing **right most derivation in reverse**,
  3. The “k” for the **number of input symbols** of look ahead that are used in making parsing decision.



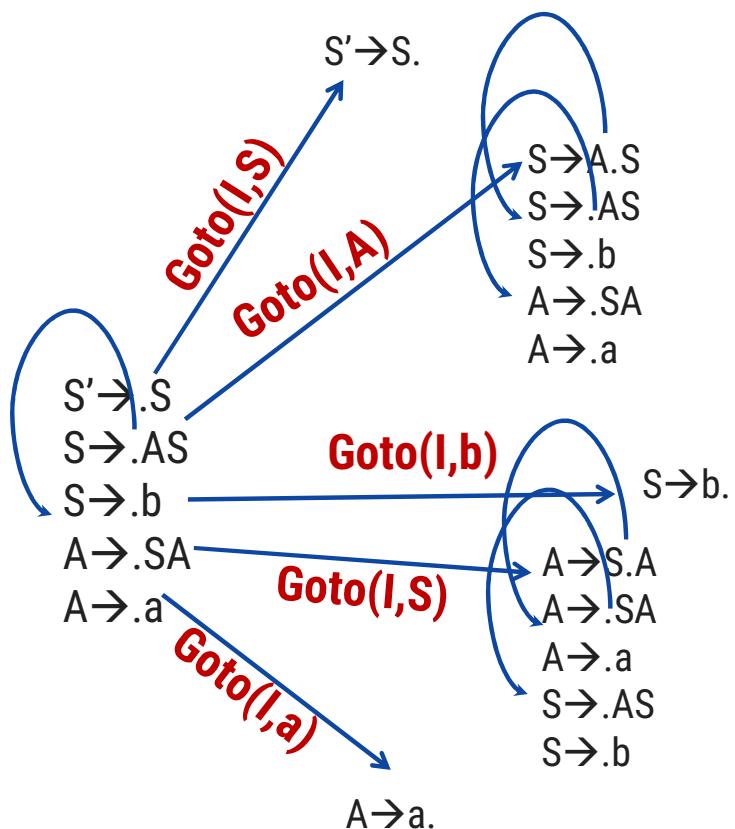
# Closure & goto function

# Computation of closure & goto function

$S \rightarrow AS \mid b$

$A \rightarrow SA \mid a$

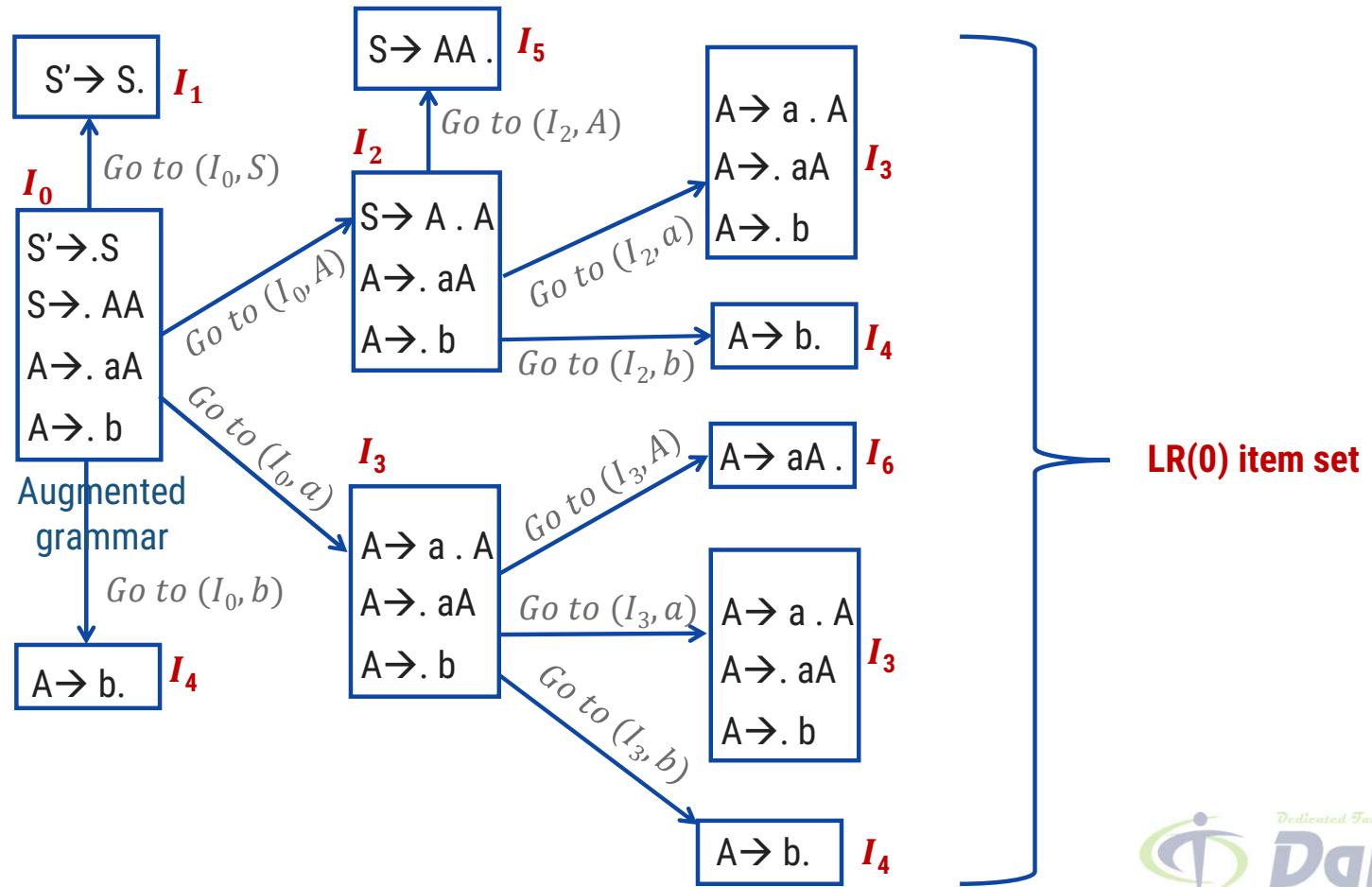
Closure(I):



# SLR Parser

# Example: SLR(1)- simple LR

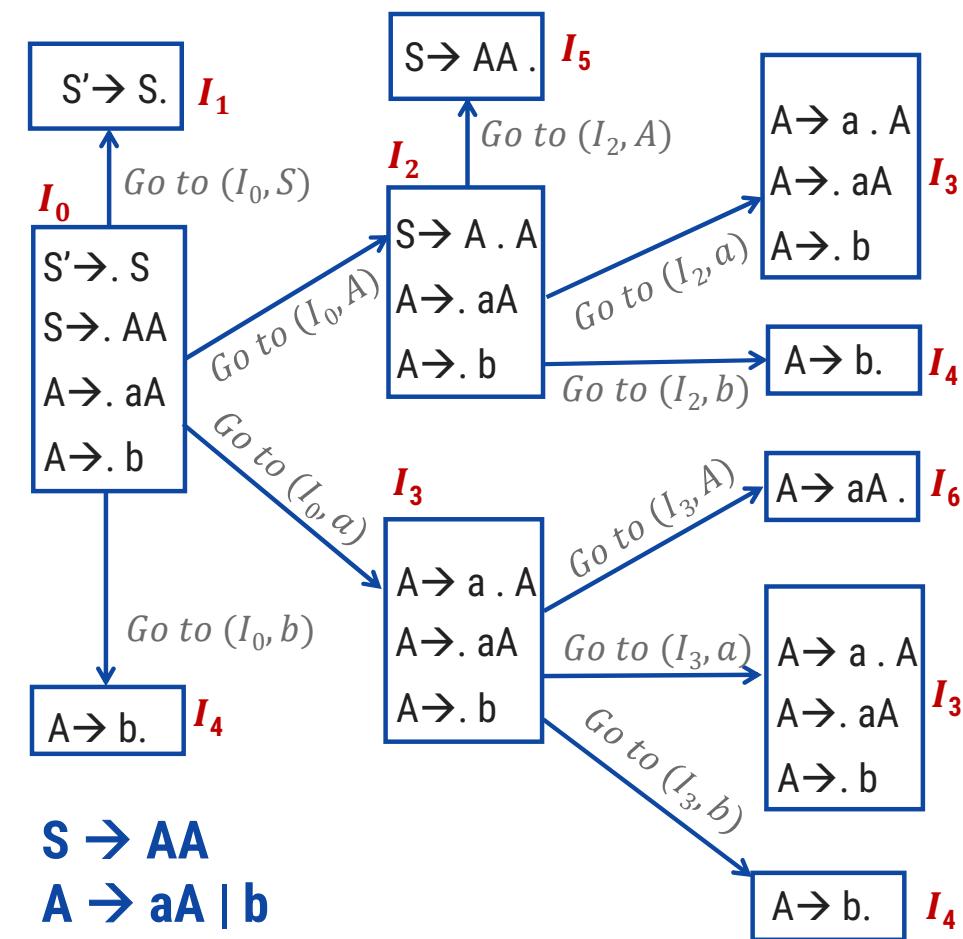
$S \rightarrow AA$   
 $A \rightarrow aA \mid b$



# Rules to construct SLR parsing table

1. Construct  $C = \{I_0, I_1, \dots, I_n\}$ , the collection of sets of LR(0) items for  $G'$ .
2. State  $i$  is constructed from  $I_i$ . The parsing actions for state  $i$  are determined as follow :
  - a) If  $[A \rightarrow \alpha. a\beta]$  is in  $I_i$  and  $\text{GOTO}(I_i, a) = I_j$ , then set  $\text{ACTION}[i, a]$  to "shift j". Here a must be terminal.
  - b) If  $[A \rightarrow \alpha.]$  is in  $I_i$ , then set  $\text{ACTION}[i, a]$  to "reduce  $A \rightarrow \alpha$ " for all  $a$  in  $\text{FOLLOW}(A)$ ; here  $A$  may not be  $S'$ .
  - c) If  $[S \rightarrow S.]$  is in  $I_i$ , then set action  $[i, \$]$  to "accept".
3. The goto transitions for state  $i$  are constructed for all non terminals  $A$  using the if  $\text{GOTO}(I_i, A) = I_j$  then  $\text{GOTO}[i, A] = j$ .
4. All entries not defined by rules 2 and 3 are made error.

# Example: SLR(1)- simple LR



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

Item set	Action			Go to	
	a	b	\$	S	A
0					
1					
2					
3					
4					
5					
6					

# CLR Parser

# How to calculate look ahead?

How to calculate look ahead?

$S \rightarrow CC$

$C \rightarrow cC \mid d$

Closure(I)

$S' \rightarrow .S\$$

$S \rightarrow .CC\$$

$C \rightarrow .cC, c|d$

$C \rightarrow .d, c|d$

<b>S'</b>	<b>→</b>		.	<b>S</b>	,	<b>\$</b>	
<b>A</b>	<b>→</b>	$\alpha$	.	X	$\beta$	,	$a$

Lookahead = First( $\beta a$ )

First(\$)

= \$

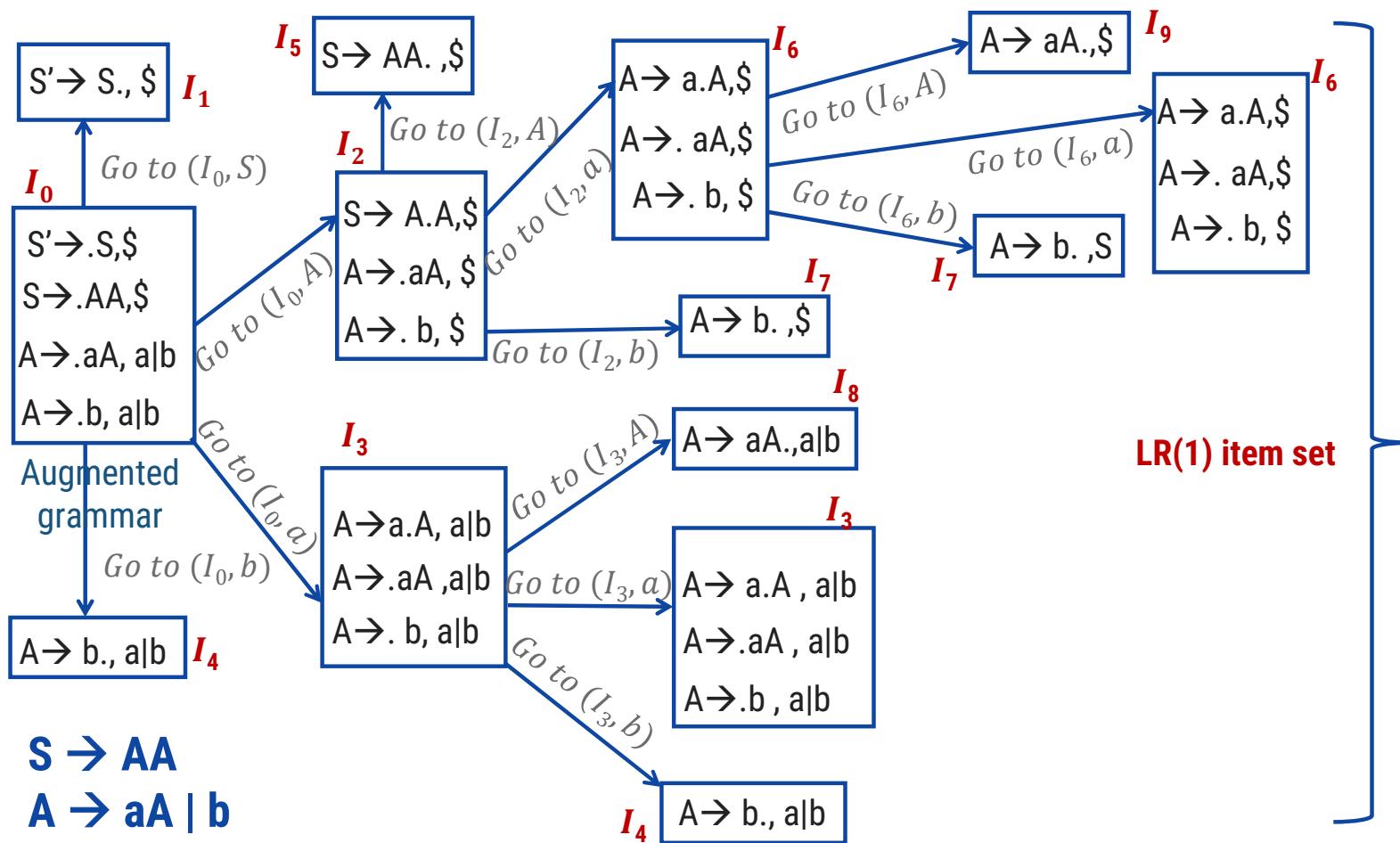
<b>S</b>	<b>→</b>		.	<b>C</b>	<b>C</b>	,	<b>\$</b>
<b>A</b>	<b>→</b>	$\alpha$	.	X	$\beta$	,	$a$

Lookahead = First( $\beta a$ )

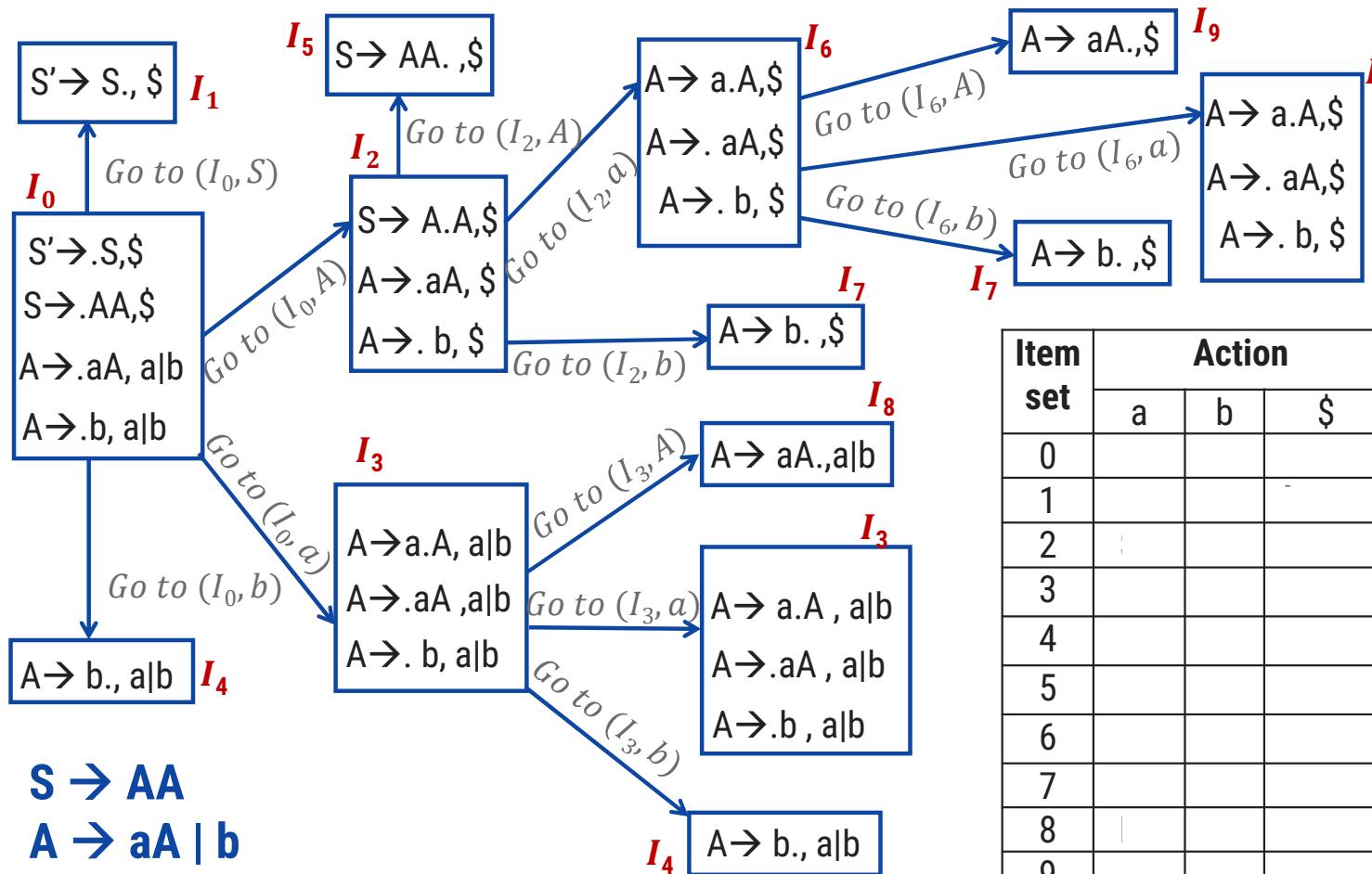
First(C\$)

=  $c, d$

# Example: CLR(1)- canonical LR



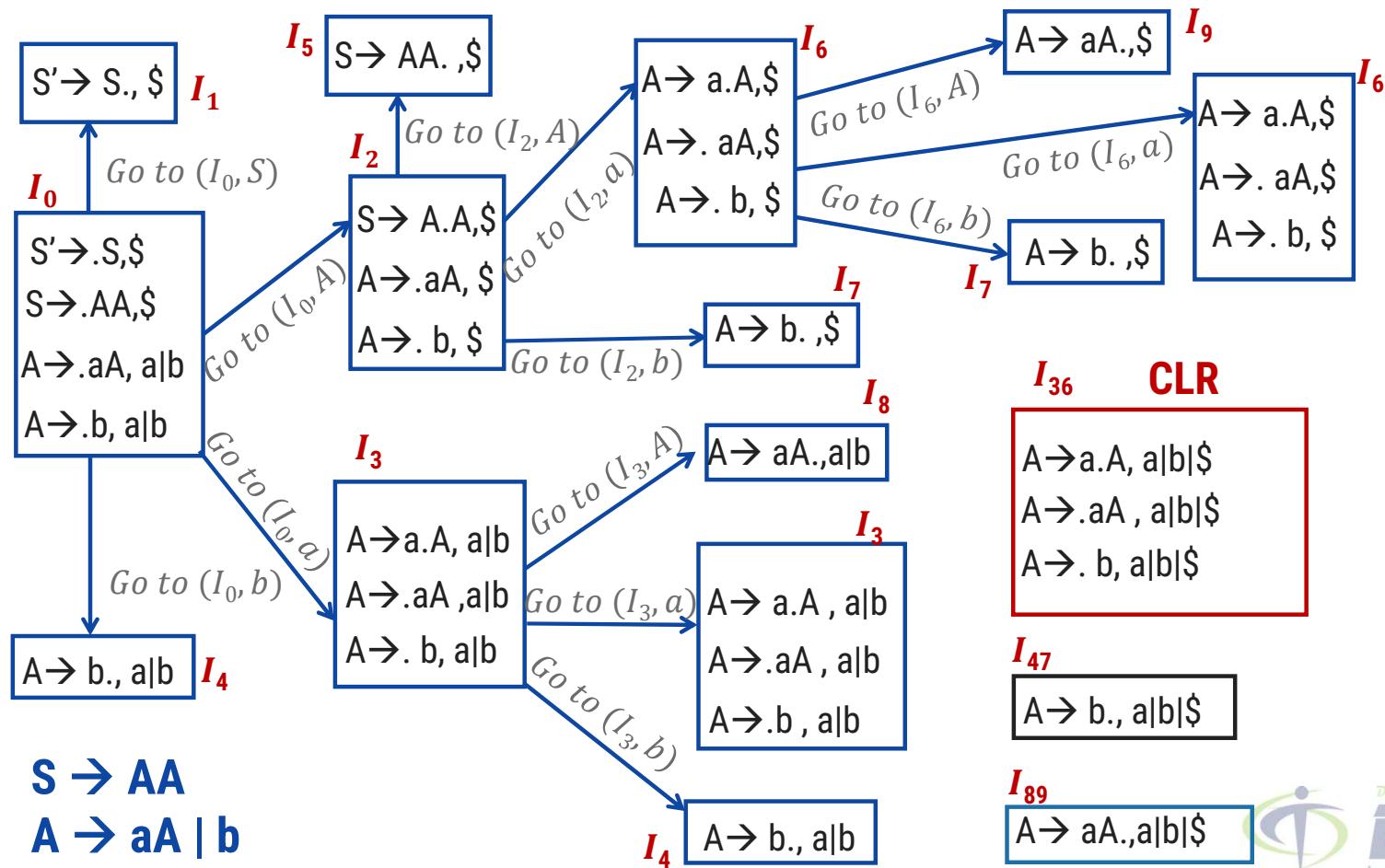
# Example: CLR(1)- canonical LR



Item set	Action			Go to	
	a	b	\$	S	A
0				-	-
1				-	-
2				-	-
3				-	-
4				-	-
5				-	-
6				-	-
7				-	-
8				-	-
9				-	-

# LALR Parser

# Example: LALR(1)- look ahead LR



# Example: LALR(1)- look ahead LR

Item set	Action			Go to	
	a	b	\$	S	A
0	S3	S4		1	2
1			Accept		
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 Parsing Table



Item set	Action			Go to	
	a	b	\$	S	A
0	S36	S47		1	2
1			Accept		
2	S36	S47			5
36	S36	S47			89
47	R3	R3	R3		
5			R1		
89	R2	R2	R2		

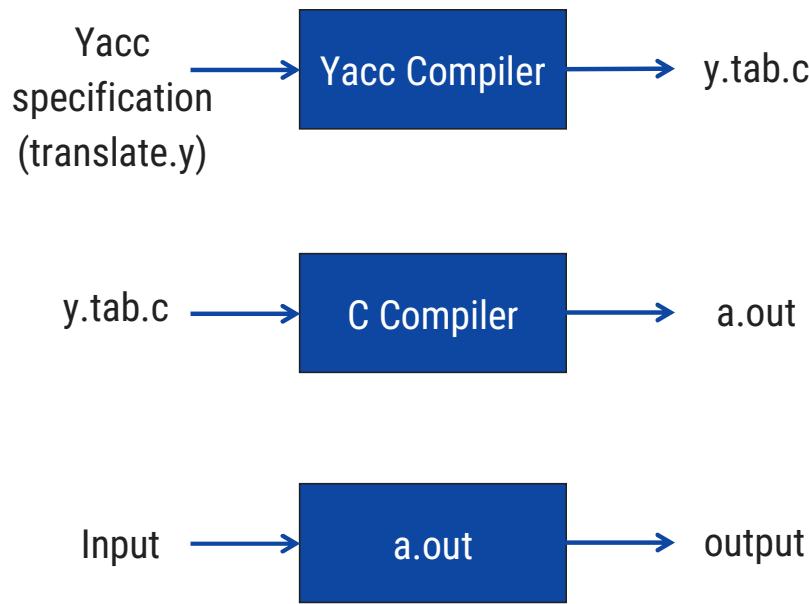
LALR Parsing Table

# Parser Generator

(YACC)

# YACC tool or YACC Parser Generator

- ▶ YACC is a tool which generates the parser.
- ▶ It takes input from the lexical analyzer (tokens) and produces parse tree as an output.



# Structure of Yacc Program

- ▶ Any Yacc program contains mainly three sections
  1. Declaration
  2. Translation rules
  3. Supporting C-routines



## Structure of Program

Declaration →  $\langle \text{left side} \rangle \rightarrow \langle \text{alt 1} \rangle | \langle \text{alt 2} \rangle | \dots | \langle \text{alt n} \rangle$   
%%

%%  
Translation rule →  $\langle \text{left side} \rangle : \langle \text{alt 1} \rangle \{ \text{semantic action 1} \}$   
|  $\langle \text{alt 2} \rangle \{ \text{semantic action 2} \}$   
|  $\langle \text{alt n} \rangle \{ \text{semantic action n} \}$   
%%

Supporting C routines → All the function needed are specified over here.

# Example: Yacc Program

## ► Program: Write Yacc program for simple desk calculator

```
/* Declaration */          /* Translation rule */          /* Supporting C routines*/  
%{                      %%  
    #include <ctype.h>      line   : expr '\n'        {print("%d\n",$1);}      yylex()  
%}                      expr   : expr '+' term   {$$=$1 + $3;}      {  
% token DIGIT           term   : term '*' factor  {$$=$1 * $3;}      int c;  
                           | factor;  
                           factor : '(' expr ')'  {$$=$2;}      c=getchar();  
                           | DIGIT;                %%  
                           }  
                           if(isdigit(c))  
                           {  
                               yylval= c-'0'  
                               return DIGIT  
                           }  
                           }  
                           return c;
```

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

# References

## Books:

### 1. **Compilers Principles, Techniques and Tools, PEARSON Education (Second Edition)**

Authors: Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

### 2. **Compiler Design, PEARSON (for Gujarat Technological University)**

Authors: Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman

# Thank You



# Unit – 3

# Syntax Analysis (II)



**Prof. Dixita B. Kagathara**  
Computer Engineering Department  
Darshan Institute of Engineering & Technology, Rajkot  
✉ dixita.kagathara@darshan.ac.in  
📞 +91 - 97277 47317 (CE Department)

# Topics to be covered

- Syntax directed definitions
- Synthesized attributes
- Inherited attribute
- Dependency graph
- Evaluation order
- Construction of syntax tree
- Bottom up evaluation of S-attributed definitions
- L-Attributed definitions
- Translation scheme

# Syntax directed definitions

# Syntax directed definitions

- ▶ Syntax directed definition is a generalization of context free grammar in which **each grammar symbol has an associated set of attributes**.
- ▶ The attributes can be a number, type, memory location, return type etc....
- ▶ Types of attributes are:
  1. Synthesized attribute
  2. Inherited attribute

## E. Memory Allocation

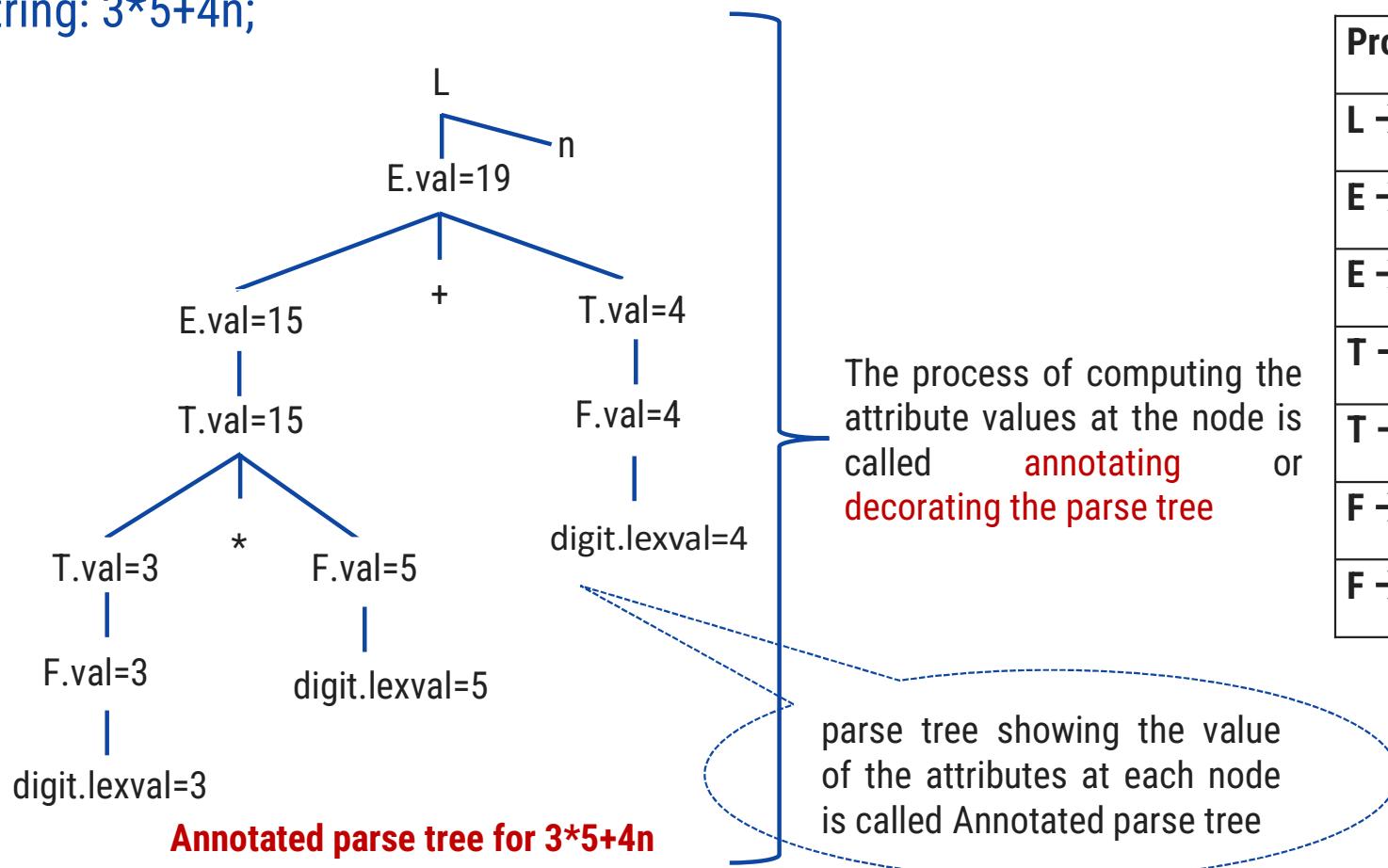
# Synthesized attributes

- ▶ Value of synthesized attribute at a node can be computed from the value of attributes at the **children of that node** in the parse tree.
- ▶ A syntax directed definition that uses synthesized attribute exclusively is said to be **S-attribute definition**.
- ▶ Example: Syntax directed definition of simple desk calculator

Production	Semantic rules
$L \rightarrow E_n$	
$E \rightarrow E_1 + T$	
$E \rightarrow T$	
$T \rightarrow T_1 * F$	
$T \rightarrow F$	
$F \rightarrow (E)$	
$F \rightarrow \text{digit}$	

# Example: Synthesized attributes

String:  $3*5+4n$ ;



Production	Semantic rules
$L \rightarrow E_n$	Print ( $E.val$ )
$E \rightarrow E_1 + T$	$E.Val = E_1.val + T.val$
$E \rightarrow T$	$E.Val = T.val$
$T \rightarrow T_1 * F$	$T.Val = T_1.val * F.val$
$T \rightarrow F$	$T.Val = F.val$
$F \rightarrow (E)$	$F.Val = E.val$
$F \rightarrow \text{digit}$	$F.Val = \text{digit} . lexval$

# Exercise

Draw Annotated Parse tree for following:

1.  $7+3*2n$
2.  $(3+4)*(5+6)n$

# Syntax directed definition to translates arithmetic expressions from infix to prefix notation

Production	Semantic rules
$L \rightarrow E$	
$E \rightarrow E + T$	
$E \rightarrow E - T$	
$E \rightarrow T$	
$T \rightarrow T * F$	
$T \rightarrow T / F$	
$T \rightarrow F$	
$F \rightarrow F^P$	
$F \rightarrow P$	
$P \rightarrow (E)$	
$P \rightarrow \text{digit}$	

# Inherited attribute

- An inherited value at a node in a parse tree is computed from the value of attributes at the **parent and/or siblings** of the node.

Production	Semantic rules
$D \rightarrow T L$	
$T \rightarrow \text{int}$	
$T \rightarrow \text{real}$	
$L \rightarrow L_1, \text{id}$	
$L \rightarrow \text{id}$	

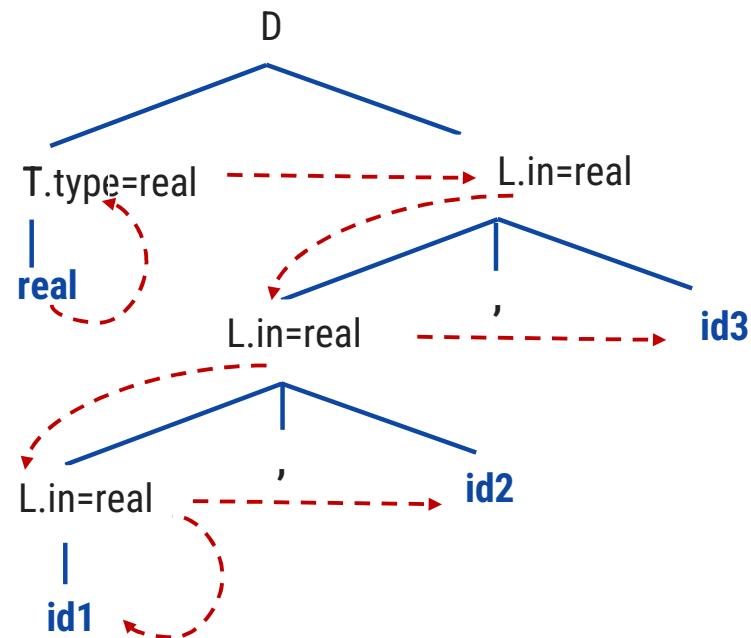
Syntax directed definition with inherited attribute L.in

- Symbol T is associated with a **synthesized attribute type**.
- Symbol L is associated with an **inherited attribute in**.

# Example: Inherited attribute

## Example: Pass data types to all identifier real id1,id2,id3

Production	Semantic rules
$D \rightarrow T L$	$L.in = T.type$
$T \rightarrow int$	$T.type = integer$
$T \rightarrow real$	$T.type = real$
$L \rightarrow L_1 , id$	$L_1.in = L.in$ , $addtype(id.entry, L.in)$
$L \rightarrow id$	$addtype(id.entry, L.in)$



$D \Rightarrow T id, id$

# Dependency graph

# Dependency graph

- ▶ The directed graph that represents the interdependencies between synthesized and inherited attribute at nodes in the parse tree is called dependency graph.
- ▶ For the rule  $X \rightarrow YZ$  the semantic action is given by  $X.x=f(Y.y, Z.z)$  then synthesized attribute  $X.x$  depends on attributes  $Y.y$  and  $Z.z$ .
- ▶ The basic idea behind dependency graphs is for a compiler to look for various kinds of dependency among statements to prevent their execution in wrong order.

# Algorithm : Dependency graph

**for** each node  $n$  in the parse tree **do**

**for** each attribute  $a$  of the grammar symbol at  $n$  **do**

Construct a node in the dependency graph for  $a$ ;

**for** each node  $n$  in the parse tree **do**

**for** each semantic rule  $b=f(c_1, c_2, \dots, c_k)$

associated with the production used at  $n$  **do**

**for**  $i=1$  to  $k$  **do**

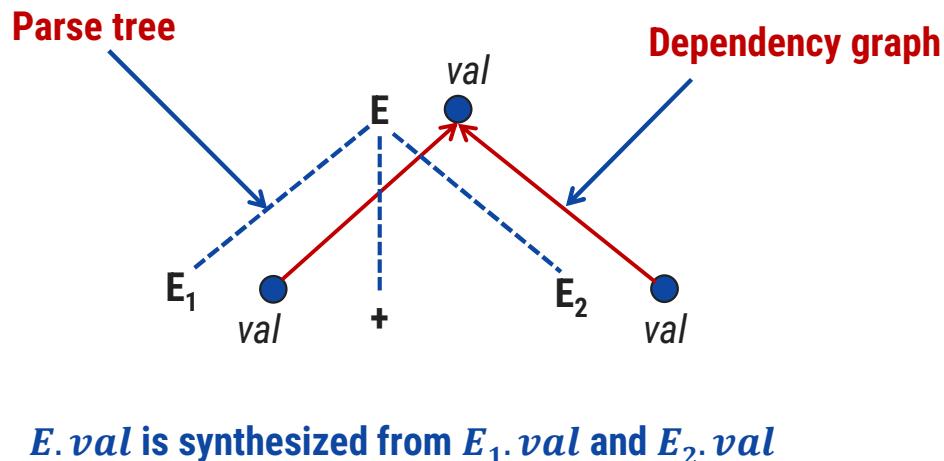
construct an edge from the node for  $C_i$  to the node for  $b$ ;

# Example: Dependency graph

Example:

$$E \rightarrow E_1 + E_2$$

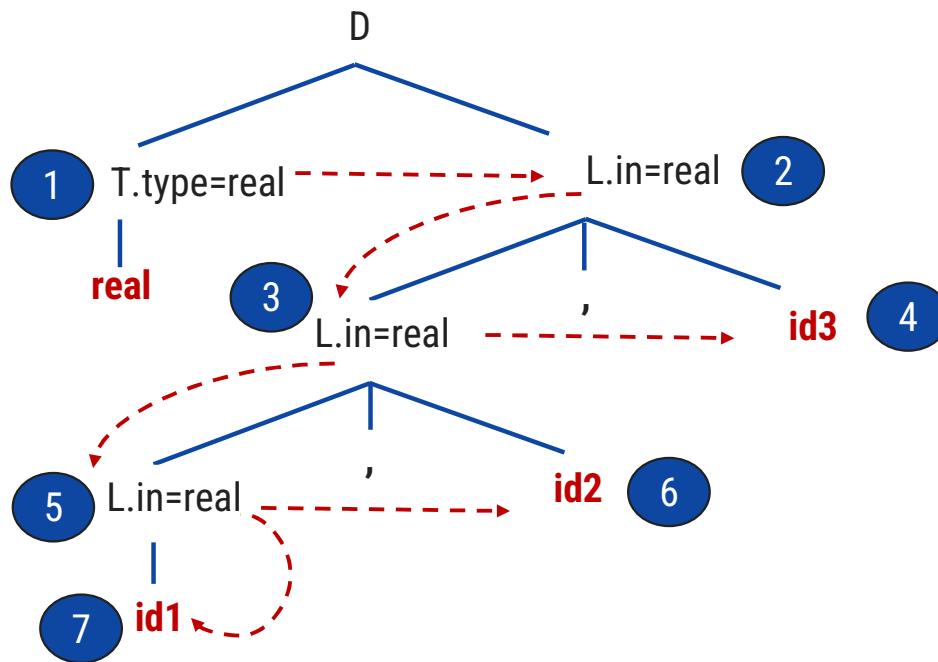
Production	Semantic rules
$E \rightarrow E_1 + E_2$	$E.val = E_1.val + E_2.val$



The edges to E.val from E<sub>1</sub>.val and E<sub>2</sub>.val shows that E.val depends on E<sub>1</sub>.val and E<sub>2</sub>.val

# Evaluation order

- ▶ A topological sort of a directed acyclic graph is any ordering  $m_1, m_2, \dots, m_k$  of the nodes of the graph such that edges go from nodes earlier in the ordering to later nodes.
- ▶ If  $m_i \rightarrow m_j$  is an edge from  $m_i$  to  $m_j$  then  $m_i$  appears before  $m_j$  in the ordering.



# Construction of syntax tree

# Construction of syntax tree

► Following functions are used to create the nodes of the syntax tree.

1. **Mknode (op, left, right)**: creates an operator node with label **op** and two fields containing pointers to **left** and **right**.
2. **Mkleaf (id, entry)**: creates an identifier node with label **id** and a field containing **entry**, a pointer to the symbol table **entry** for the identifier.
3. **Mkleaf (num, val)**: creates a number node with label **num** and a field containing **val**, the value of the number.

# Construction of syntax tree for expressions

**Example:** construct syntax tree for  $a - 4 + c$

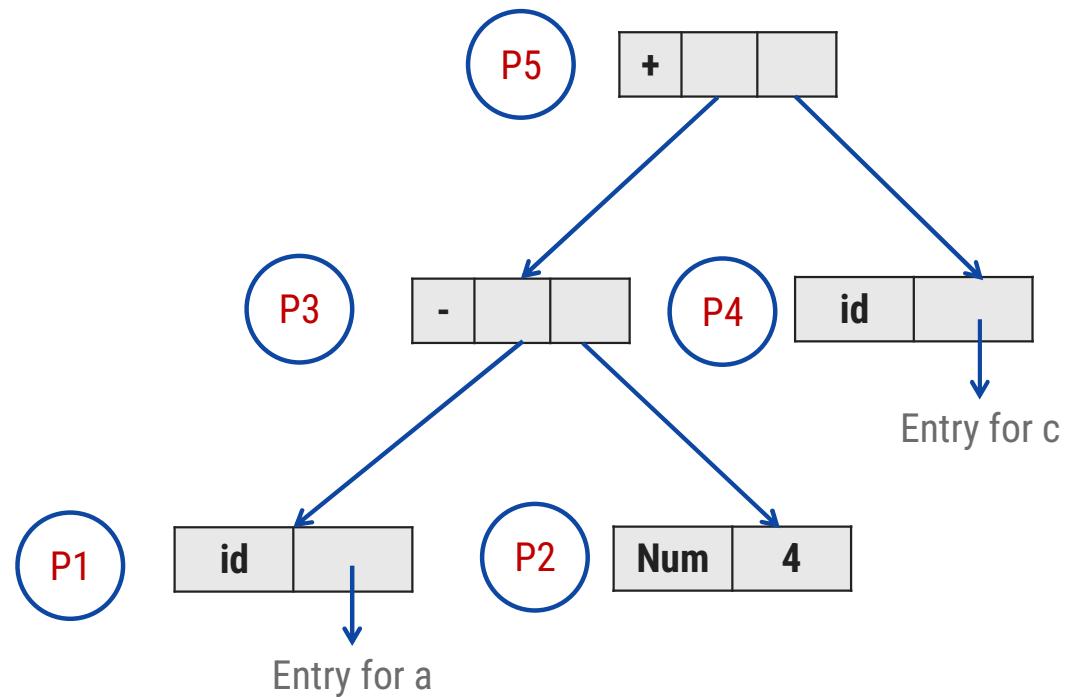
P1: mkleaf(id, entry for a);

P2: mkleaf(num, 4);

P3: mknod(-,p1,p2);

P4: mkleaf(id, entry for c);

P5: mknod(+,p3,p4);



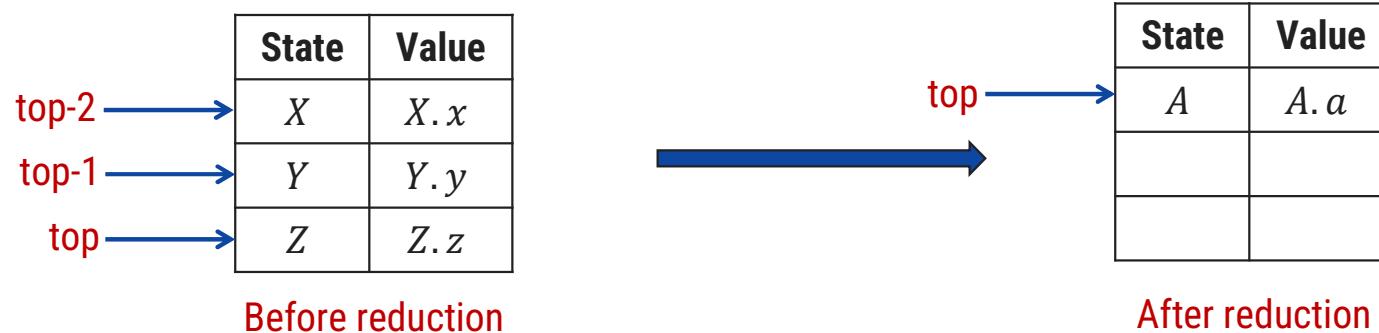
# **Bottom up evaluation of S-attributed definitions**

# Bottom up evaluation of S-attributed definitions

- ▶ S-attributed definition is one such class of syntax directed definition with synthesized attributes only.
- ▶ Synthesized attributes can be evaluated using bottom up parser only.

## Synthesized attributes on the parser stack

- ▶ Consider the production  $A \rightarrow XYZ$  and associated semantic action is  $A.a=f(X.x, Y.y, Z.z)$



# Bottom up evaluation of S-attributed definitions

Production	Semantic rules
$L \rightarrow E_n$	Print (val[top])
$E \rightarrow E_1 + T$	$val[top] = val[top-2] + val[top]$
$E \rightarrow T$	
$T \rightarrow T_1 * F$	$val[top] = val[top-2] * val[top]$
$T \rightarrow F$	
$F \rightarrow (E)$	$val[top] = val[top-2] - val[top]$
$F \rightarrow \text{digit}$	

Implementation of a desk calculator  
with bottom up parser

Input	State	Val	Production Used
3*5n	-	-	

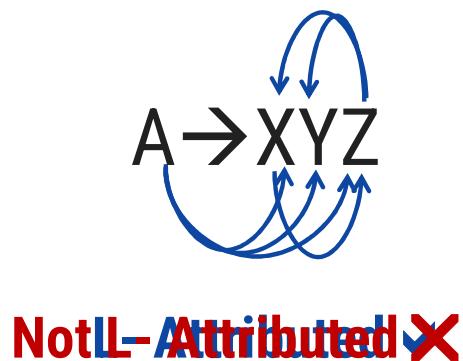
Move made by translator

# L-Attributed definitions

# L-Attributed definitions

- ▶ A syntax directed definition is L-attributed if each inherited attribute of  $X_j$ ,  $1 \leq j \leq n$ , on the right side of  $A \rightarrow X_1, X_2 \dots X_n$  depends only on:
  1. The attributes of the symbols  $X_1, X_2, \dots X_{j-1}$  to the left of  $X_j$  in the production and
  2. The inherited attribute of A.

- ▶ Example:



Production	Semantic Rules
$A \rightarrow LM$	$L.i := l(A.i)$ $M.i = m(L.s)$ $A.s = f(M.s)$
$A \rightarrow QR$	$R.i = r(A.i)$ $Q.i = q(R.s)$ $A.s = f(Q.s)$

- ▶ Above syntax directed definition is *not L-attributed* because the inherited attribute Q.i of the grammar symbol Q depends on the attribute R.s of the grammar symbol to its right.

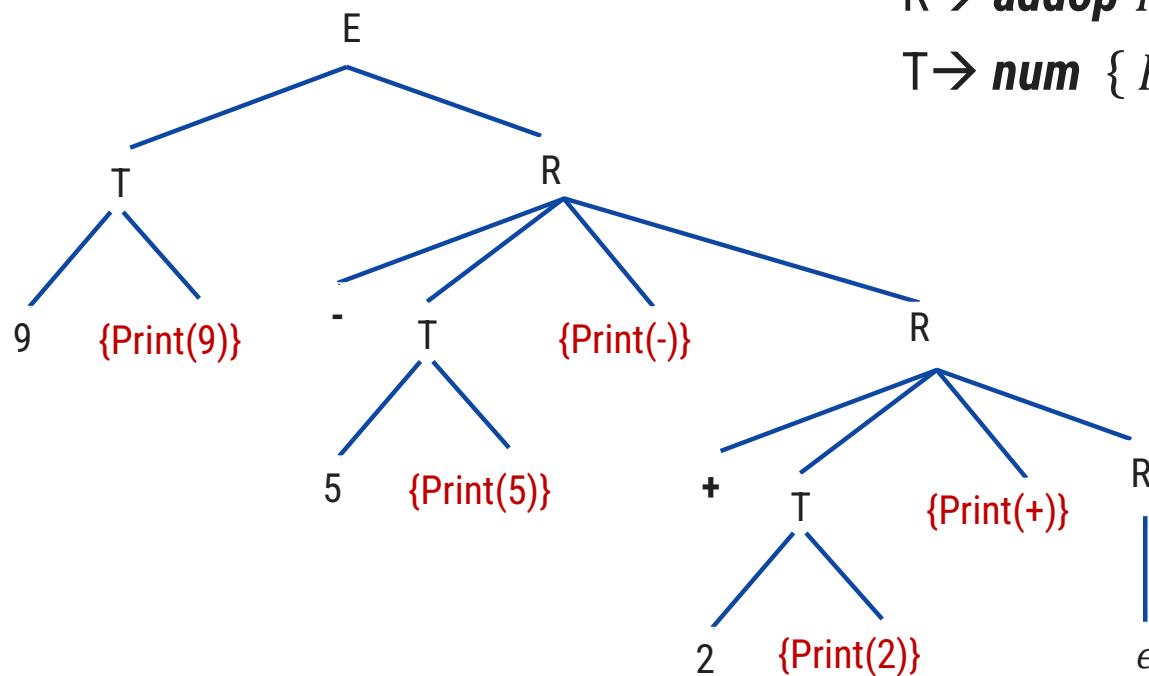
# Translation scheme

# Translation Scheme

- ▶ Translation scheme is a context free grammar in which attributes are associated with the grammar symbols and semantic actions enclosed between braces { } are inserted within the right sides of productions.
- ▶ Attributes are used to evaluate the expression along the process of parsing.
- ▶ During the process of parsing the evaluation of attribute takes place by consulting the semantic action enclosed in {}.
- ▶ A translation scheme generates the output by executing the semantic actions in an ordered manner.
- ▶ This process uses the depth first traversal.

# Example: Translation scheme (Infix to postfix notation)

String: 9-5+2



$E \rightarrow TR$

$R \rightarrow addop\ T\ \{Print(addop.\text{lexeme})\}\ R1\ |\epsilon$

$T \rightarrow num\ \{Print(num.\text{val})\}$

Now, Perform Depth first traversal

Postfix=95-2+

# References

## Books:

### 1. **Compilers Principles, Techniques and Tools, PEARSON Education (Second Edition)**

Authors: Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

### 2. **Compiler Design, PEARSON (for Gujarat Technological University)**

Authors: Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman

# Thank You



# Unit – 4

# Error Recovery



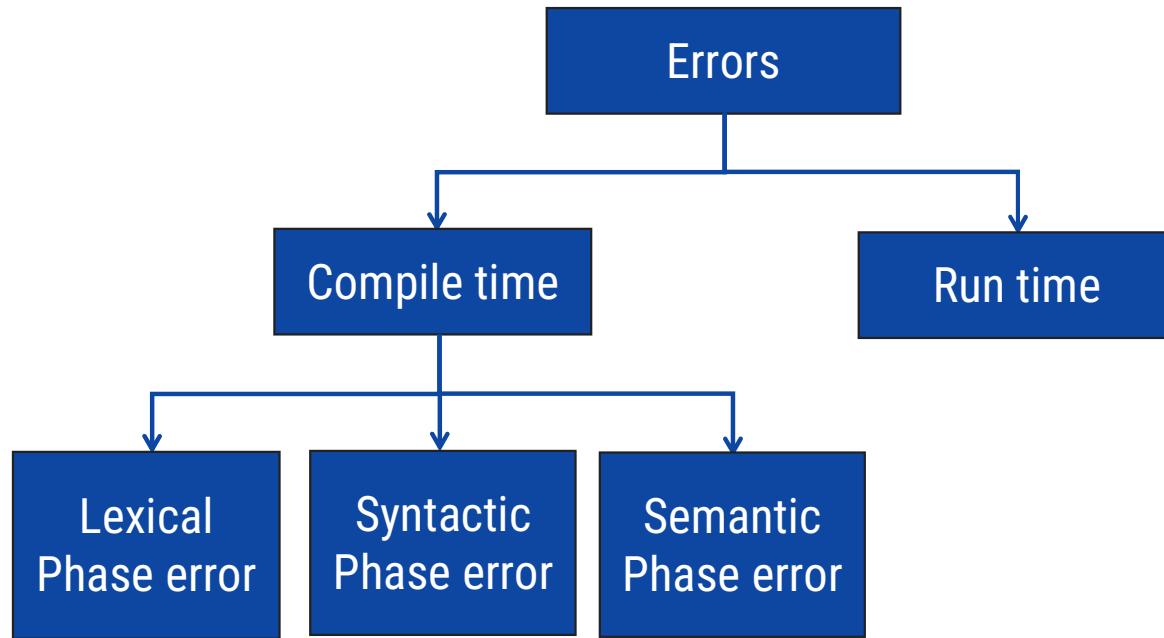
**Prof. Dixita B. Kagathara**  
Computer Engineering Department  
Darshan Institute of Engineering & Technology, Rajkot  
✉ dixita.kagathara@darshan.ac.in  
📞 +91 - 97277 47317 (CE Department)

## Topics to be covered

- Types of errors
- Error recovery strategies

# Types of errors

# Types of Errors



# Lexical error

- ▶ Lexical errors can be detected during lexical analysis phase.
- ▶ Typical lexical phase errors are:
  1. Spelling errors
  2. Exceeding length of identifier or numeric constants
  3. Appearance of illegal characters

- ▶ Example:

```
fi ()  
{  
}
```

- ▶ In above code '**fi**' cannot be recognized as a **misspelling of keyword if** rather lexical analyzer will understand that it is an identifier and will **return it as valid identifier**.
- ▶ Thus misspelling causes errors in token formation.

# Syntax error

- ▶ Syntax error appear during syntax analysis phase of compiler.
- ▶ Typical syntax phase errors are:
  1. Errors in structure
  2. Missing operators
  3. Unbalanced parenthesis
- ▶ The parser demands for tokens from lexical analyzer and if the tokens do not satisfies the grammatical rules of programming language then the syntactical errors get raised.
- ▶ Example:

`printf("Hello World !!!") ← Error: Semicolon missing`

# Semantic error

- ▶ Semantic error detected during semantic analysis phase.
- ▶ Typical semantic phase errors are:
  1. Incompatible types of operands
  2. Undeclared variable
  3. Not matching of actual argument with formal argument
- ▶ Example:

id1=id2+id3\*60 (Note: id1, id2, id3 are real)  
(Directly we can not perform multiplication due to incompatible types of variables)

# Error recovery strategies (Ad-Hoc & systematic methods)

# Error recovery strategies (Ad-Hoc & systematic methods)

► There are mainly four error recovery strategies:

1. Panic mode
2. Phrase level recovery
3. Error production
4. Global generation

# Panic mode

- ▶ In this method on discovering error, the parser discards input symbol one at a time. This process is continued until one of a **designated set of synchronizing tokens** is found.
- ▶ Synchronizing tokens are **delimiters** such as semicolon or end.
- ▶ These tokens **indicate an end** of the statement.
- ▶ If there is less number of errors in the same statement then this strategy is best choice.

fi () ← Scan entire line otherwise scanner will return fi as valid identifier

{  
}

# Phrase level recovery

- ▶ In this method, on discovering an error parser **performs local correction** on remaining input.
- ▶ The local correction can be:
  1. Replacing comma by semicolon
  2. Deletion of semicolons
  3. Inserting missing semicolon
- ▶ This type of local correction is decided by compiler designer.
- ▶ This method is used in many error-repairing compilers.

# Error production

- ▶ If we have good knowledge of common errors that might be encountered, then we can augment the grammar for the corresponding language with **error productions that generate the erroneous constructs.**
- ▶ Then we use the grammar augmented by these error production to construct a parser.
- ▶ If error production is used then, during parsing we can generate appropriate error message and parsing can be continued.

# Global correction

- ▶ Given an incorrect input **string x and grammar G**, the algorithm will find a parse tree for a **related string y**, such that number of insertions, deletions and changes of token require **to transform x into y is as small as possible.**
- ▶ Such methods increase time and space requirements at parsing time.
- ▶ Global correction is thus simply a theoretical concept.

# References

## Books:

### 1. **Compilers Principles, Techniques and Tools, PEARSON Education (Second Edition)**

Authors: Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

### 2. **Compiler Design, PEARSON (for Gujarat Technological University)**

Authors: Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman

# Thank You



## Unit – 5

# Intermediate Code Generation



**Prof. Dixita B. Kagathara**

Computer Engineering Department

Darshan Institute of Engineering & Technology, Rajkot

✉ dixita.kagathara@darshan.ac.in

📞 +91 - 97277 47317 (CE Department)

## Topics to be covered

- Different intermediate forms
- Different representation of Three Address code

# Different intermediate forms

# Different intermediate forms

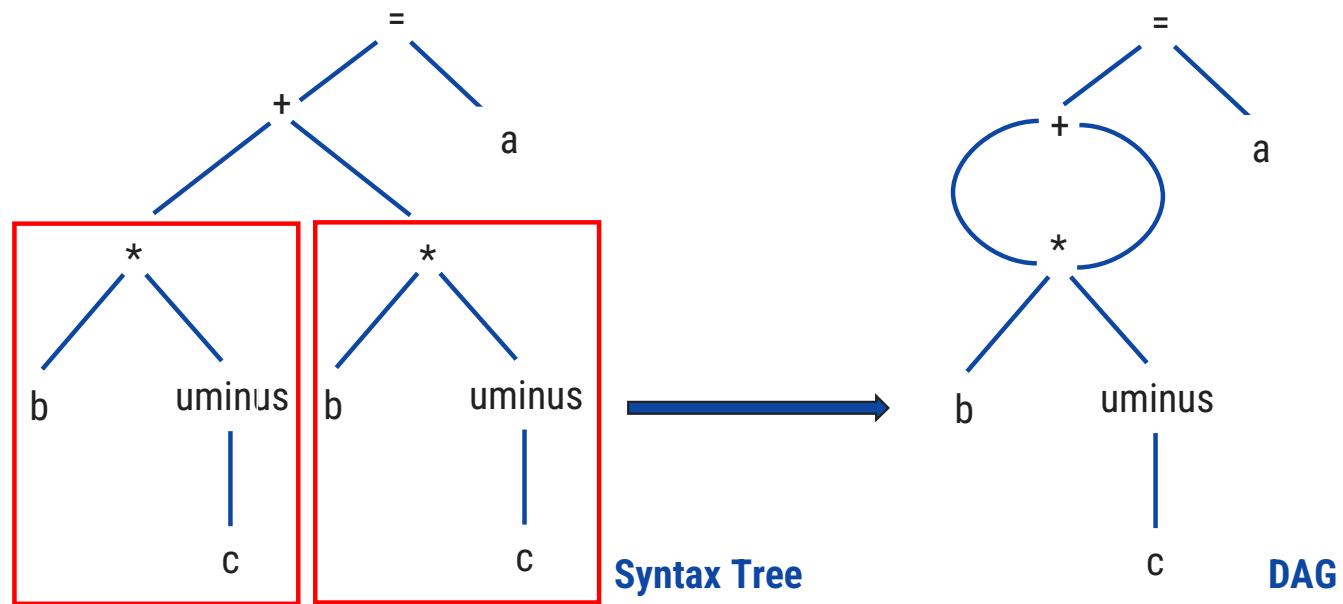
► Different forms of intermediate code are:

1. Abstract syntax tree
2. Postfix notation
3. Three address code



# Abstract syntax tree & DAG

- ▶ A syntax tree depicts the natural hierarchical structure of a source program.
- ▶ A DAG (Directed Acyclic Graph) gives the same information but in a more **compact** way because **common sub-expressions** are identified.
- ▶ Ex:  $a = b * -c + b * -c$



# Postfix Notation

- ▶ Postfix notation is a linearization of a syntax tree.
- ▶ In postfix notation the operands occurs first and then operators are arranged.
- ▶ Ex: **(A + B) \* (C + D)**

**Postfix notation: A B + C D + \***

- ▶ Ex: **(A + B) \* C**

**Postfix notation: A B + C \***

- ▶ Ex: **(A \* B) + (C \* D)**

**Postfix notation: A B \* C D \* +**

# Three address code

- ▶ Three address code is a sequence of statements of the general form,

$$a := b \text{ op } c$$

- ▶ Where **a**, **b** or **c** are the operands that can be names or constants and **op** stands for any operator.
- ▶ Example:  $a = b + c + d$

$$t_1 = b + c$$

$$t_2 = t_1 + d$$

$$a = t_2$$

- ▶ Here  $t_1$  and  $t_2$  are the temporary names generated by the compiler.
- ▶ There are **at most three addresses allowed** (two for operands and one for result). Hence, this representation is called three-address code.

# Different Representation of Three Address Code

# Different Representation of Three Address Code

- ▶ There are three types of representation used for three address code:

1. Quadruples
2. Triples
3. Indirect triples

- ▶ Ex:  $x = -a * b + -a * b$

$$\begin{aligned} t_1 &= -a \\ t_2 &= t_1 * b \\ t_3 &= -a \\ t_4 &= t_3 * b \\ t_5 &= t_2 + t_4 \\ x &= t_5 \end{aligned}$$

Three Address Code

# Quadruple

- ▶ The quadruple is a structure with at the most four fields such as op, arg1, arg2 and result.
- ▶ The op field is used to represent the internal code for operator.
- ▶ The arg1 and arg2 represent the two operands.
- ▶ And result field is used to store the result of an expression.

$x = -a * b + -a * b$   
 $t_1 = -a$   
 $t_2 = t_1 * b$   
 $t_3 = -a$   
 $t_4 = t_3 * b$   
 $t_5 = t_2 + t_4$   
 $x = t_5$

Quadruple				
No.	Operator	Arg1	Arg2	Result
(0)				
(1)				
(2)				
(3)				
(4)				
(5)				

# Triple

- ▶ To avoid entering temporary names into the symbol table, we might refer a temporary value by the position of the statement that computes it.
- ▶ If we do so, three address statements can be represented by records with only three fields: op, arg1 and arg2.

**Quadruple**

No.	Operator	Arg1	Arg2	Result
(0)	uminus	a		t <sub>1</sub>
(1)	*	t <sub>1</sub>	b	t <sub>2</sub>
(2)	uminus	a		t <sub>3</sub>
(3)	*	t <sub>3</sub>	b	t <sub>4</sub>
(4)	+	t <sub>2</sub>	t <sub>4</sub>	t <sub>5</sub>
(5)	=	t <sub>5</sub>		x

**Triple**

No.	Operator	Arg1	Arg2
(0)			
(1)			
(2)			
(3)			
(4)			
(5)			

# Indirect Triple

- ▶ In the indirect triple representation the listing of triples has been done. And listing pointers are used instead of using statement.
- ▶ This implementation is called indirect triples.

**Triple**

No.	Operator	Arg1	Arg2
(0)	uminus	a	
(1)	*	(0)	b
(2)	uminus	a	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	x	(4)

**Indirect Triple**

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

No.	Operator	Arg1	Arg2
(0)	uminus	a	
(1)	*		b
(2)	uminus	a	
(3)	*		b
(4)	+		
(5)	=	x	

# Exercise

Write quadruple, triple and indirect triple for following:

1.  $-(a*b)+(c+d)$
2.  $a*-(b+c)$
3.  $x=(a+b*c)^(d*e)+f*g^h$
4.  $g+a*(b-c)+(x-y)*d$

# References

## Books:

### 1. **Compilers Principles, Techniques and Tools, PEARSON Education (Second Edition)**

Authors: Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

### 2. **Compiler Design, PEARSON (for Gujarat Technological University)**

Authors: Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman

# Thank You



# Unit – 6

# Run-Time Environments



**Prof. Dixita B. Kagathara**  
Computer Engineering Department  
Darshan Institute of Engineering & Technology, Rajkot  
  
✉ dixita.kagathara@darshan.ac.in  
📞 +91 - 97277 47317 (CE Department)

## Topics to be covered

- Source language issues
- Storage organization
- Storage allocation strategies

# Source language issues

# Run-Time Environments

- ▶ As execution proceeds, the same name in the source text can denote different data objects in the target machine.
- ▶ Each execution of a procedure is referred to as an activation of the procedure.
- ▶ If the procedure is recursive, several of its activations may be alive at same time.

## Source language issues:

1. Procedures
2. Activation tree
3. Control stack
4. The Scope of a declaration
5. Binding of names

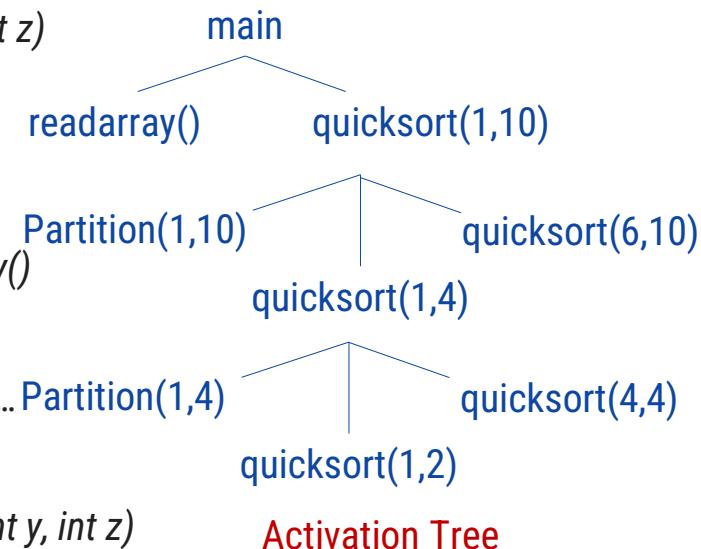
# Procedures

- ▶ A program is made up of procedures.
- ▶ Procedure: declaration that associate an identifier with a statement.
- ▶ Identifier: procedure name
- ▶ Statement: procedure body
- ▶ Procedure call: procedure name appears within an executable statement.
- ▶ Example:

```
main()           void quicksort(int m, int n)           void readarray()
{               {                               {
    int n;          int i=partition(m, n);           .....
    readarray();    quicksort(m, i-1);             }
    quicksort(1, n);   quicksort(i+1, n);           int partition(int y, int z)
}               }                               {
                                .....                         }
}                           }
```

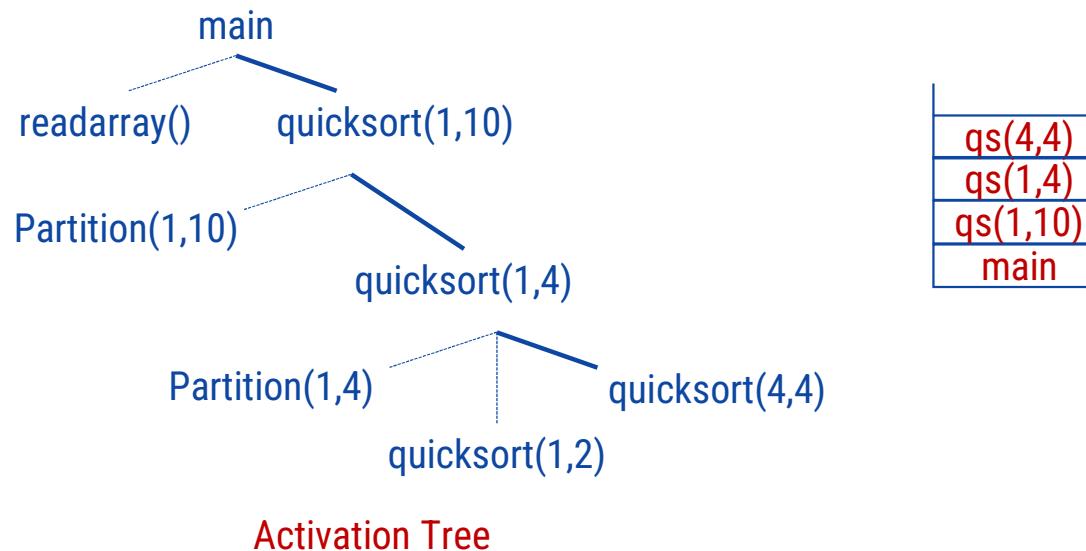
# Activation Tree

- ▶ If 'a' and 'b' are two procedures, their activations will be:
  - ↳ Non-overlapping: when one is called after other
  - ↳ Nested procedure     $\text{int } i = \text{partition}(m, n);$
  - ↳ Recursive procedure: new quicksort(i-1) begins before an earlier activation of the same procedure has ended.
- ▶ An activation tree shows the way control enters and leaves activations.
- ▶ Properties of activation trees are :-
  1. Each node represents an activation of a procedure.
  2. The root shows the activation of the main function.
  3. The node for procedure 'a' is the parent of node for procedure 'b' if and only if the control flows from procedure a to procedure b.
  4. If node 'a' is left of the node 'b' if and only if the lifetime of a occurs before the lifetime of b.



# Control stack

- ▶ Control stack or runtime stack is used to keep track of the live procedure activations i.e the procedures whose execution have not been completed.
- ▶ A procedure name is pushed on to the stack when it is called (activation begins) and it is popped when it returns (activation ends).
- ▶ Information needed by a single execution of a procedure is managed using an activation record or frame. When a procedure is called, an activation record is pushed into the stack and as soon as the control returns to the caller function the activation record is popped.

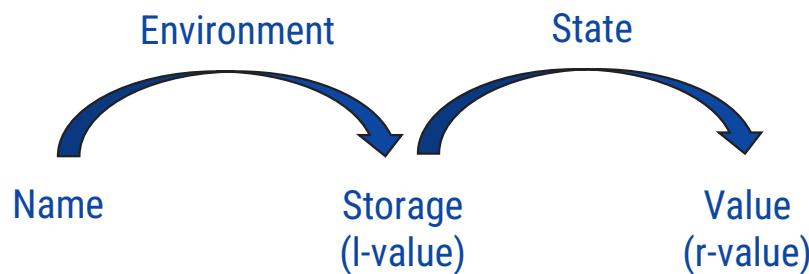


# Scope of a declaration

- ▶ A declaration in a language is a syntactic construct that associate information with a name.
  - Var i: integer;
- ▶ There may be declaration of the same name in different parts of a program.
- ▶ The scope rules of a language determine which declaration of a name applies when the name appears in the text of a program.
- ▶ The portion of the program, to which declaration applies is called the scope of the declaration.
- ▶ An occurrence of a name in a procedure is said to be local to the procedure if it is in the scope of a declaration within the procedure; otherwise the occurrence is said to be nonlocal.
- ▶ The distinction between local and non local names carries over to any syntactic construct that can have declaration within it.

# Binding of names

- ▶ Environment: function that maps a name to a storage location.
- ▶ State: function that maps a storage location to the value held there.



- ▶ When an environment associates storage location s with a name  $x \rightarrow x$  is bound to s.

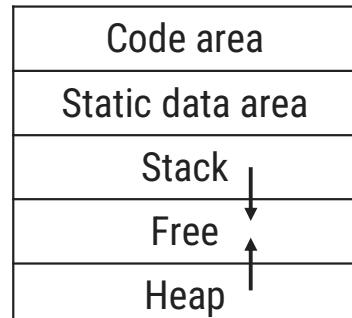
# Storage Organization

# Storage organization

- ▶ The executing target program runs in it's own **logical address** space in which each program value has a location.
- ▶ The management and organization of this logical address space is shared between the **compiler, operating system** and **target machine**.
- ▶ The operating system maps the logical address into the physical address, which are usually spread throughout memory.

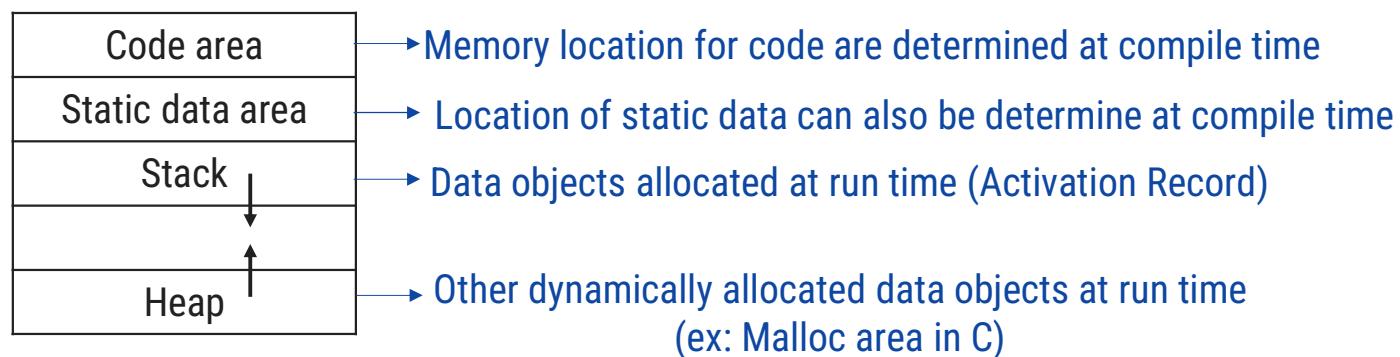
# Subdivision of Runtime Memory

- ▶ The compiler demands for a block of memory to operating system.
- ▶ The compiler utilizes this block of memory executing the compiled program. This block of memory is called **run time storage**.
- ▶ The run time storage is subdivided to hold code and data such as, the generated target code and data objects.
- ▶ The size of generated code is fixed. Hence the target code occupies the determined area of the memory.



# Subdivision of Runtime Memory

- ▶ Code block consisting of a memory location for code.
- ▶ The amount of memory required by the data objects is known at the compiled time and hence data objects also can be placed at the statically determined area of the memory.
- ▶ Stack is used to manage the active procedure.
- ▶ Managing of active procedures means when a call occurs then execution of activation is interrupted and information about status of the stack is saved on the stack.
- ▶ Heap area is the area of run time storage in which the other information is stored.



# Activation Record

- ▶ Control stack is a run time stack which is used to keep track of the live procedure activations i.e. it is used to find out the procedures whose execution have not been completed.
- ▶ When it is called (activation begins) then the procedure name will push on to the stack and when it returns (activation ends) then it will popped.
- ▶ Activation record is used to manage the information needed by a single execution of a procedure.
- ▶ An activation record is pushed into the stack when a procedure is called and it is popped when the control returns to the caller function.

# Activation Record

- ▶ The **execution of a procedure is called its activation.**
- ▶ An activation record contains all the necessary information required to call a procedure.
- ▶ **Return value:** used by the called procedure to return a value to calling procedure.
- ▶ **Actual parameters:** This field holds the information about the actual parameters.
- ▶ **Control link (optional):** points to activation record of caller.
- ▶ **Access link (optional):** refers to non-local data held in other activation records.
- ▶ **Machine status:** holds the information about status of machine just before the function call.
- ▶ **Local variables:** hold the data that is local to the execution of the procedure.
- ▶ **Temporary values:** stores the values that arise in the evaluation of an expression.

Return value
Actual parameters
Control link
Access link
Machine status
Local variable
Temporary values

# Compile-Time Layout of Local Data

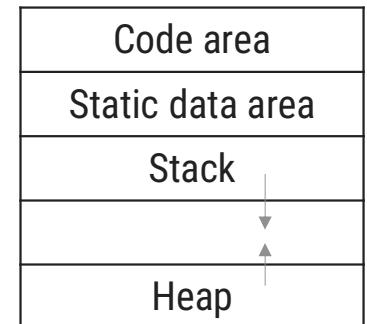
- ▶ The **amount of storage** needed for a name is determined **from its type**. (e.g.: int, char, float...)
- ▶ Storage for an aggregate, such as an **array or record**, must be large enough to hold all it's components.
- ▶ The field of local data is laid out as the declarations in a procedure are examined at **compile time**.
- ▶ We keep a **count** of the memory locations that have been **allocated for previous declarations**.
- ▶ From the count we determine a relative address of the storage for a local with respect to some position such as the beginning of the activation record.

# Storage allocation strategies

# Storage allocation strategies

The different storage allocation strategies are:

- ▶ **Static allocation:** lays out storage for all data objects at compile time.
- ▶ **Stack allocation:** manages the run-time storage as a stack.
- ▶ **Heap allocation:** allocates and de-allocates storage as needed at run time from a data area known as heap.



# Static allocation

- ▶ In static allocation, **names are bound to storage as the program is compiled**, so there is no need for a run-time support package.
- ▶ Since the bindings do not change at run-time, every time a procedure is activated, its names are bounded to the same storage location.

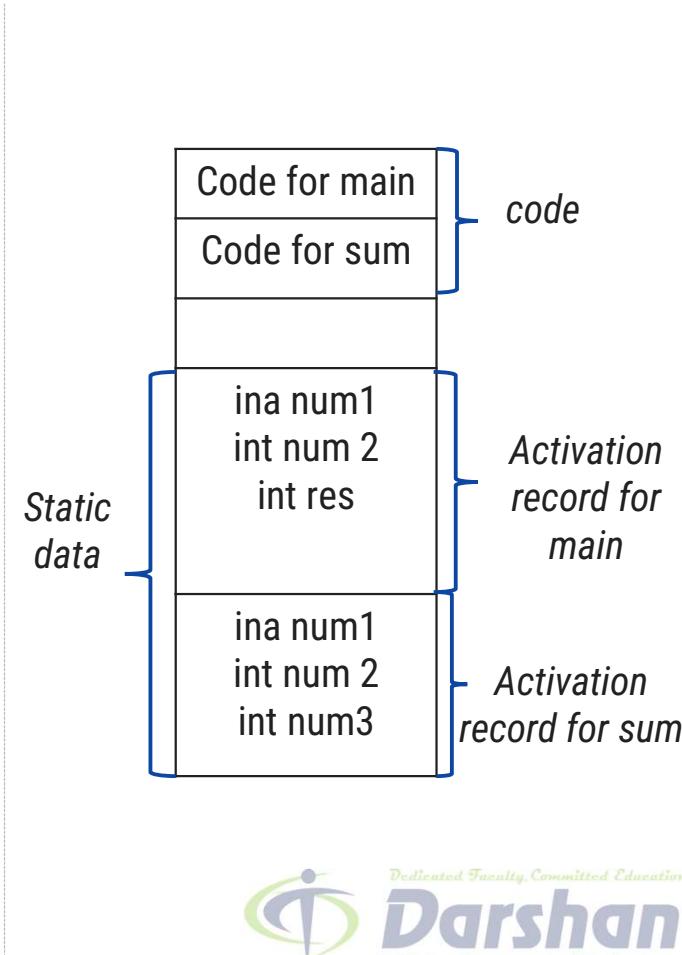
## Limitation

1. Size of data object must be known at compile time.
2. Recursive procedures are restricted.
3. Data structure can't be created dynamically.

Example:

```
int main()
{
    int num1=100, num2=200, res;
    res = sum(num1, num2);
    printf("\n Addition is %d : ",res);
    return (0);
}

int sum(int num1, int num2)
{
    int num3;
    num3 = num1 + num2;
    return (num3);
}
```



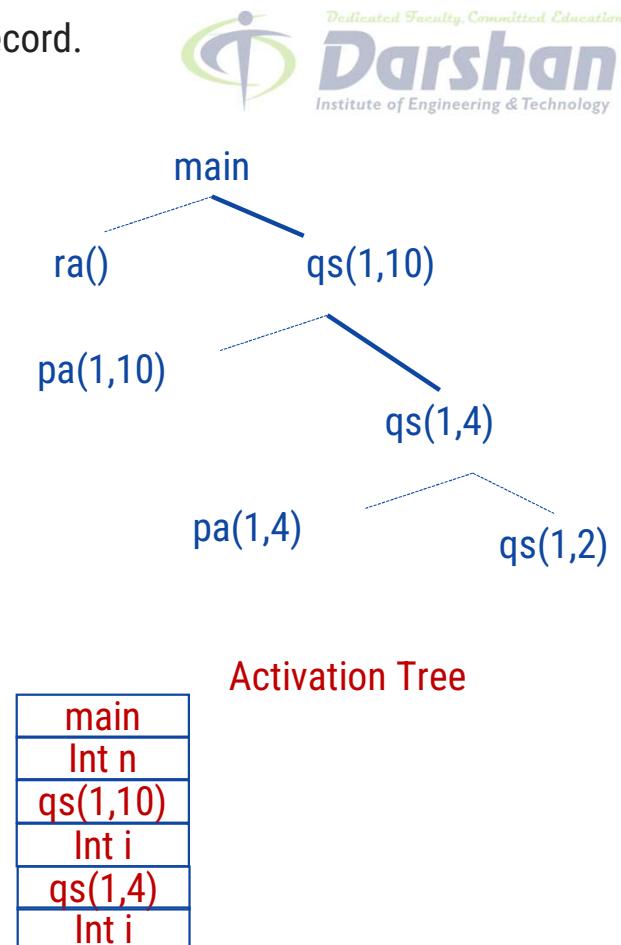
# Stack allocation

- ▶ All compilers for languages that use procedures, functions or methods as units of user define actions manage at least part of their run-time memory as a stack.
- ▶ Each time a procedure is called, **space for its local variables is pushed onto a stack, and when the procedure terminates, the space is popped off the stack.**
- ▶ Locals are bound to fresh storage in each activations.
- ▶ Locals are deleted when the activation ends.

# Stack allocation

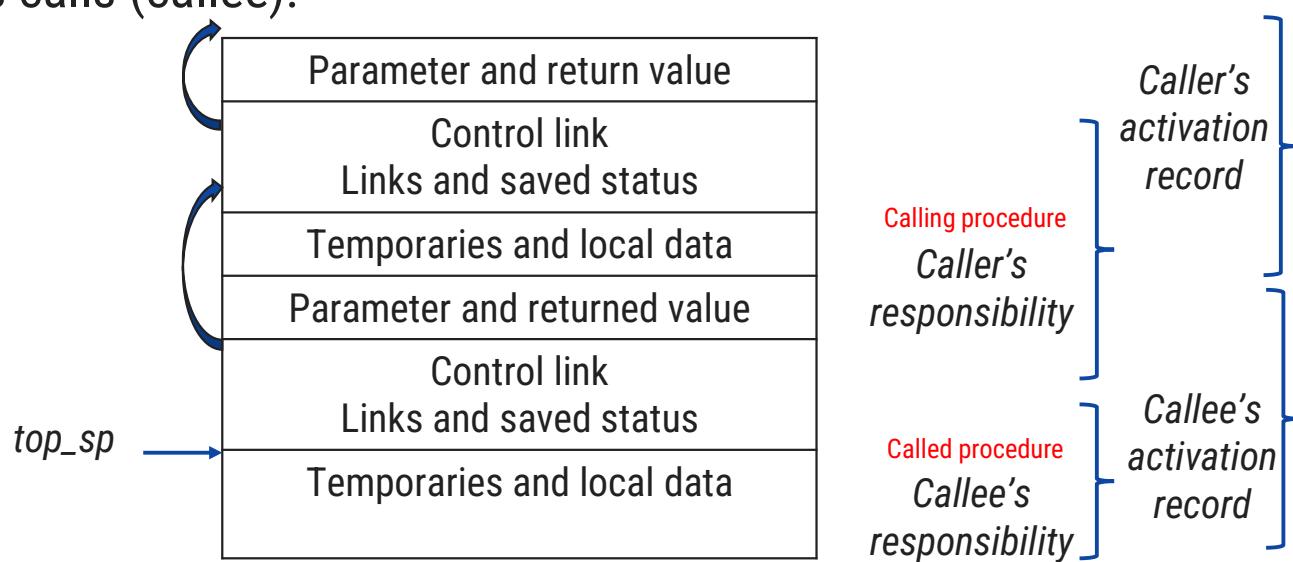
- At run time, an activation record can be allocated by incrementing 'top' by the size of the record.
- Deallocated by decrementing 'top' by the size of record.

Position in activation tree	Activation record on the stack	Remarks				
main	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>main</td></tr> <tr><td>Int n</td></tr> </table>	main	Int n	Frame for main		
main						
Int n						
main ra()	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>main</td></tr> <tr><td>Int n</td></tr> <tr><td>ra</td></tr> <tr><td>Int i</td></tr> </table>	main	Int n	ra	Int i	Ra is activated
main						
Int n						
ra						
Int i						
main ra() qs(1,10)	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>main</td></tr> <tr><td>Int n</td></tr> <tr><td>qs(1,10)</td></tr> <tr><td>Int i</td></tr> </table>	main	Int n	qs(1,10)	Int i	Frame ra has been popped and qs(1,10) is pushed.
main						
Int n						
qs(1,10)						
Int i						



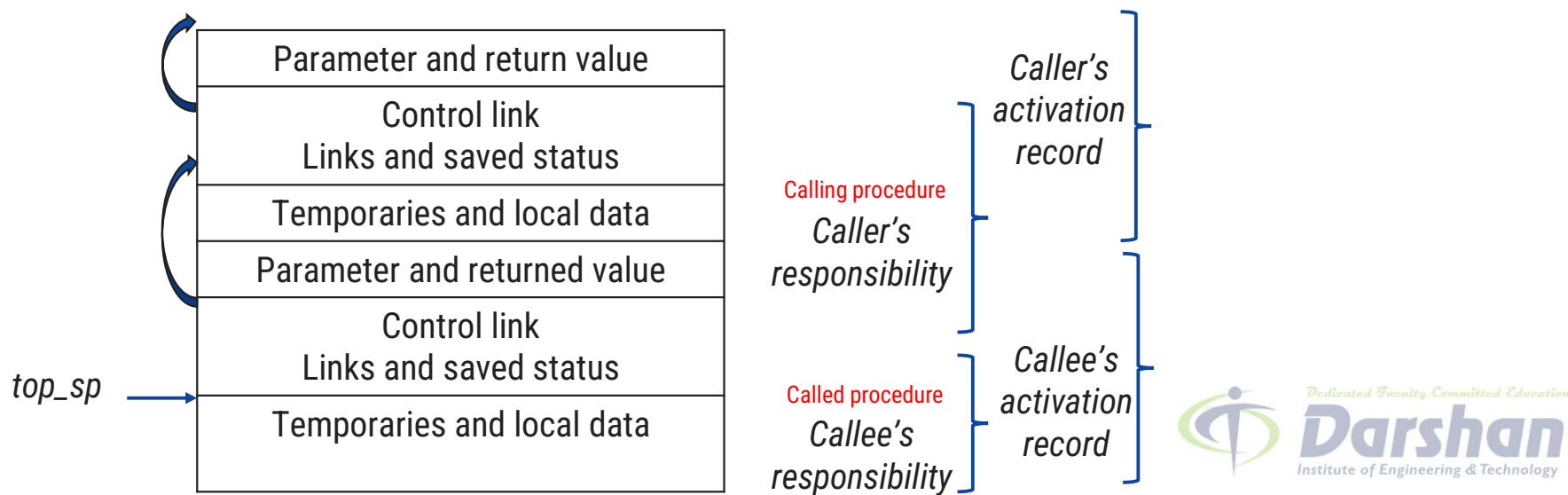
# Stack allocation: Calling Sequences

- ▶ Procedures calls are implemented by generating what are known as **calling sequences in the target code**.
- ▶ A call sequence **allocates an activation record** and **enters the information** into its fields.
- ▶ A Return sequence restore the state of machine so the calling procedure can continue its execution.
- ▶ The code is calling sequence of often divided between the calling procedure (caller) and procedure is calls (callee).



# Stack allocation: Calling Sequences

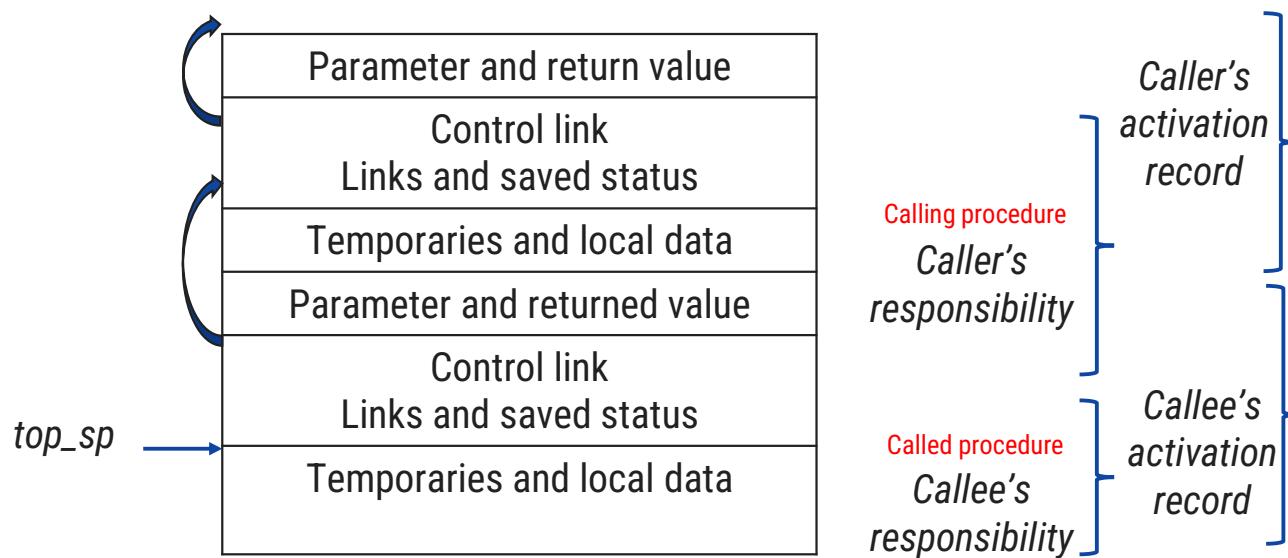
- ▶ The calling sequence and its division between caller and callee are as follows:
  1. The caller (**Calling procedure**) evaluates the actual parameters.
  2. The caller stores a return address and the old value of *top\_sp* into the callee's activation record. The caller then increments the *top\_sp* to the respective positions.
  3. The callee (**Called procedure**) saves the register values and other status information.
  4. The callee initializes its local data and begins execution.



# Stack allocation: Calling Sequences

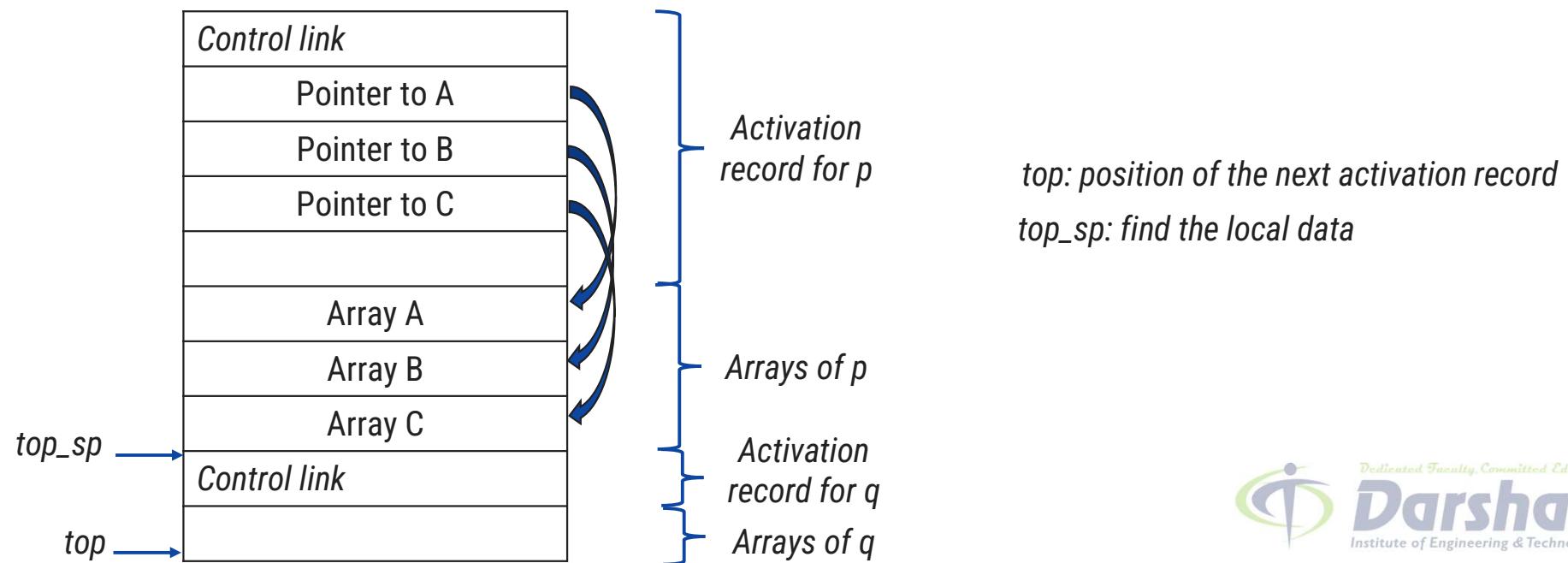
► A suitable, corresponding return sequence is:

1. The callee places the return value next to the parameters.
2. Using the information in the machine status field, the callee restores *top\_sp* and other registers, and then branches to the return address that the caller placed in the status field.
3. Although *top\_sp* has been decremented, the caller knows where the return value is, relative to the current value of *top\_sp*; the caller therefore may use that value.



# Stack allocation: Variable length data on stack

- ▶ The run time memory management system must deal frequently with the allocation of objects, the sizes of which are not known at the compile time, but which are local to a procedure and thus may be allocated on the stack.
- ▶ The same scheme works for objects of any type if they are local to the procedure called have a size that depends on the parameter of the call.



# Stack allocation: Dangling Reference

- ▶ When there is a reference to storage that has been deallocated.
- ▶ It is a logical error to use dangling reference, since, the value of de-allocated storage is undefined according to the semantics of most languages.

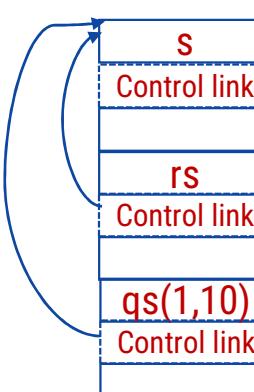
```
main()
{
    int *p;
    p=dangle();
}

int *dangle()
{
    int i=10;
    return &i;
}
```

# Heap Allocation

The stack allocation strategy can not be used for following condition:

1. The values of the local names must be retained when an activation ends.
2. A called activation outlives the called.

Position in activation tree	Activation record on the heap	Remarks
main ra() qs(1,10)		Retains activation record for ra

- Records for the live activations need not be adjacent in a heap.

# References

## Books:

### 1. **Compilers Principles, Techniques and Tools, PEARSON Education (Second Edition)**

Authors: Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

### 2. **Compiler Design, PEARSON (for Gujarat Technological University)**

Authors: Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman

# Thank You



## Unit - 7

# Code Generation & Optimization



**Prof. Dixita B. Kagathara**

Computer Engineering Department

Darshan Institute of Engineering & Technology, Rajkot

✉ dixita.kagathara@darshan.ac.in

📞 +91 - 97277 47317 (CE Department)

# Topics to be covered

- Issues in the design of a code generator
- The Target machine
- Basic block and flow-graph
- Transformation on basic block
- A simple code generator
- Code optimization

# Issues in the design of a code generator

# Issues in design of Code Generator

## ► Issues in Code Generation are:

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Choice of evaluation
7. Approaches to code generation

# Input to code generator

- ▶ Input to the code generator consists of the intermediate representation of the source program.
- ▶ Types of intermediate language are:
  1. Postfix notation
  2. Three address code
  3. Syntax trees or DAGs
- ▶ The detection of semantic error should be done before submitting the input to the code generator.
- ▶ The code generation phase requires complete error free intermediate code as an input.

# Target program

► The output may be in form of:

1. **Absolute machine language:** Absolute machine language program can be placed in a memory location and immediately execute.
2. **Relocatable machine language:** The subroutine can be compiled separately. A set of relocatable object modules can be linked together and loaded for execution.
3. **Assembly language:** Producing an assembly language program as output makes the process of code generation easier, then assembler is required to convert code in binary form.

# Memory management

- ▶ **Mapping names** in the source program **to addresses of data objects** in run time memory is done cooperatively by the front end and the code generator.
- ▶ We assume that a name in a three-address statement refers to a symbol table entry for the name.
- ▶ From the symbol table information, a relative address can be determined for the name in a data area.

# Instruction selection

- ▶ Example: the sequence of statements

a := b + c

d := a + e

- ▶ would be translated into

MOV b, R0

ADD c, R0

MOV R0, a

**MOV a, R0**

ADD e, R0

MOV R0, d

MOV b, R0

ADD c, R0

ADD e, R0

MOV R0, d

- ▶ So, we can eliminate redundant statements.

# Register allocation

- ▶ The use of registers is often subdivided into two sub problems:
- ▶ During **register allocation**, we select the **set of variables** that will reside in registers at a point in the program.
- ▶ During a subsequent **register assignment** phase, we pick the **specific register** that a variable will reside in.
- ▶ Finding an optimal assignment of registers to variables is difficult, even with single register value.
- ▶ Mathematically the problem is **NP-complete**.

# Choice of evaluation

- ▶ The **order in which computations are performed** can affect the efficiency of the target code.
- ▶ Some computation orders require fewer registers to hold intermediate results than others.
- ▶ Picking a best order is another difficult, **NP-complete problem**.

# Approaches to code generation

- ▶ The most important criterion for a code generator is that it produces correct code.
- ▶ The design of code generator should be in such a way so it can be implemented, tested, and maintained easily.

# The Target Machine

# Target machine

- ▶ We will assume our target computer models a three-address machine with:
  1. load and store operations
  2. computation operations
  3. jump operations
  4. conditional jumps
- ▶ The underlying computer is a byte-addressable machine with  $n$  general-purpose registers,  $R_0, R_1, \dots, R_n$

# Addressing Modes

Mode	Form	Address	Extra cost
Absolute	M	M	1
Register	R	R	0
Indexed	k(R)	k +contents(R)	1
Indirect register	*R	contents(R)	0
Indirect indexed	*k(R)	contents(k + contents(R))	1

# Instruction Cost

Mode	Form	Address	Extra cost
Absolute	M	M	1
Register	R	R	0
Indexed	k(R)	k +contents(R)	1
Indirect register	*R	contents(R)	0
Indirect indexed	*k(R)	contents(k + contents(R))	1

▶ Calculate cost for following:

MOV B,R0 ADD C,R0 MOV R0,A	

# Instruction Cost

Mode	Form	Address	Extra cost
Absolute	M	M	1
Register	R	R	0
Indexed	k(R)	k +contents(R)	1
Indirect register	*R	contents(R)	0
Indirect indexed	*k(R)	contents(k + contents(R))	1

▶ Calculate cost for following:

MOV *R1 ,*R0 MOV *R1 ,*R0	

# Basic blocks and flow graph

# Basic Blocks

- ▶ A basic block is a **sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end** without halt or possibility of branching except at the end.
- ▶ The following sequence of three-address statements forms a basic block:

t1 := a\*a

t2 := a\*b

t3 := 2\*t2

t4 := t1+t3

t5 := b\*b

t6 := t4+t5

# Algorithm: Partition into basic blocks

**Input:** A sequence of three-address statements.

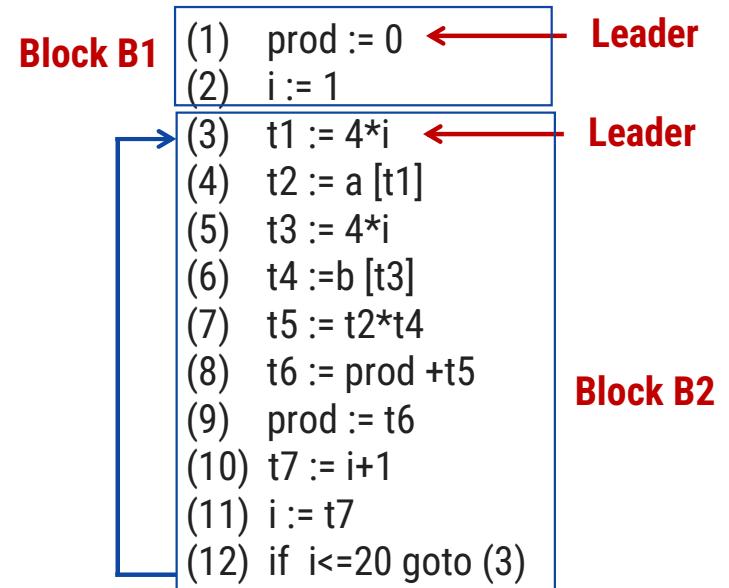
**Output:** A list of basic blocks with each three-address statement in exactly one block.

**Method:**

1. We first determine the set of **leaders**, for that we use the following rules:
  - I. The first statement is a leader.
  - II. Any statement that is the target of a conditional or unconditional goto is a leader.
  - III. Any statement that immediately follows a goto or conditional goto statement is a leader.
2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

# Example: Partition into basic blocks

```
begin
    prod := 0;
        i := 1;
    do
        prod := prod + a[t1] * b[t2];
        i := i+1;
    while i<= 20
end
```



Three Address Code

# Transformation on Basic Blocks

Optimization of Basic block

# Transformation on Basic Blocks

- ▶ A number of transformations can be applied to a basic block without changing the set of expressions computed by the block.
- ▶ Many of these **transformations** are useful for **improving the quality of the code**.
- ▶ Types of transformations are:
  1. Structure preserving transformation
  2. Algebraic transformation

# Structure Preserving Transformations

- ▶ Structure-preserving transformations on basic blocks are:
  1. Common sub-expression elimination
  2. Dead-code elimination
  3. Renaming of temporary variables
  4. Interchange of two independent adjacent statements

# Common sub-expression elimination

- ▶ Consider the basic block,

a:= b+c

b:= a-d

c:= b+c

d:= a-d

- ▶ The second and fourth statements compute the same expression, hence this basic block may be transformed into the equivalent block:

a:= b+c

b:= a-d

c:= b+c

d:= b

# Dead-code elimination

- ▶ Suppose  $x$  is dead, that is, never subsequently used, at the point where the statement  $x := y + z$  appears in a basic block.
- ▶ Above statement may be safely removed without changing the value of the basic block.

# Renaming of temporary variables

- ▶ Suppose we have a statement

$t := b + c$ , where  $t$  is a temporary variable.

- ▶ If we change this statement to

$u := b + c$ , where  $u$  is a new temporary variable,

- ▶ Change all uses of this instance of  $t$  to  $u$ , then the value of the basic block is not changed.
- ▶ In fact, we can always transform a basic block into an equivalent block in which each statement that defines a temporary defines a new temporary.
- ▶ We call such a basic block a *normal-form* block.

# Interchange of two independent adjacent statements

- ▶ Suppose we have a block with the two adjacent statements,

$t1 := b+c$

$t2 := x+y$

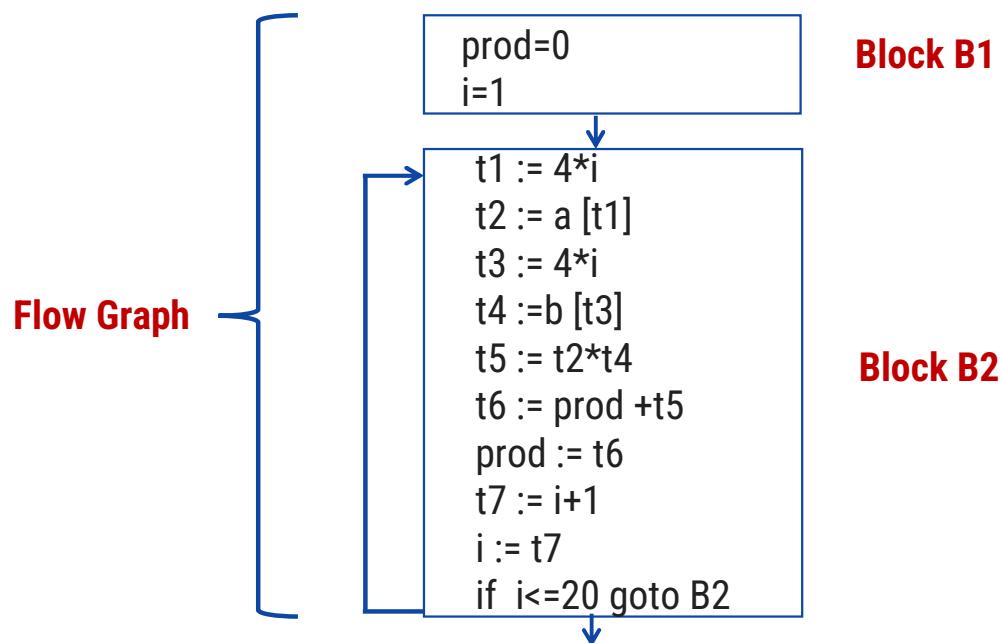
- ▶ Then we can interchange the two statements without affecting the value of the block if and only if neither  $x$  nor  $y$  is  $t1$  and neither  $b$  nor  $c$  is  $t2$ .
- ▶ A normal-form basic block permits all statement interchanges that are possible.

# Algebraic Transformation

- ▶ Countless algebraic transformation can be used to change the set of expressions computed by the basic block into an algebraically equivalent set.
- ▶ The useful ones are those that **simplify expressions or replace expensive operations by cheaper one.**
- ▶ Example: **x=x+0 or x=x\*1** can be eliminated.

# Flow Graph

- ▶ We can add flow-of-control information to the set of basic blocks making up a program by constructing a direct graph called a **flow graph**.
- ▶ Nodes in the flow graph represent computations, and the edges represent the flow of control.
- ▶ Example of flow graph for following three address code:



# A simple code generator

# A simple code generator

- ▶ The code generation strategy generates target code for a sequence of three address statement.
- ▶ It uses function **getReg()** to assign register to variable.
- ▶ The code generator algorithm uses descriptors to keep track of register contents and addresses for names.
- ▶ **Address descriptor** stores the location where the current value of the name can be found at run time. The information about locations can be stored in the symbol table and is used to access the variables.
- ▶ **Register descriptor** is used to keep track of what is currently in each register. The register descriptor shows that initially all the registers are empty. As the generation for the block progresses the registers will hold the values of computation.

# A Code Generation Algorithm

- ▶ The algorithm takes a sequence of three-address statements as input. For each three address statement of the form  $x := y \text{ op } z$  perform the various actions. Assume L is the location where the output of operation  $y \text{ op } z$  is stored.
1. Invoke a function `getReg()` to find out the location L where the result of computation  $y \text{ op } z$  should be stored.
  2. Determine the present location of 'y' by consulting address description for y if y is not present in location L then generate the instruction **MOV y', L** to place a copy of y in L
  3. Present location of z is determined using step 2 and the instruction is generated as **OP z', L**
  4. If L is a register then update it's descriptor that it contains value of x. update the address descriptor of x to indicate that it is in L.
  5. If the current value of y or z have no next uses or not live on exit from the block or in register then alter the register descriptor to indicate that after execution of  $x := y \text{ op } z$  those register will no longer contain y or z.

# Generating a code for assignment statement

- The assignment statement  $d := (a-b) + (a-c) + (a-c)$  can be translated into the following sequence of three address code:

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

$t := a - b$

$u := a - c$

$v := t + u$

$d := v + u$

# Optimization

Machine independent optimization

# Code Optimization

- ▶ Code Optimization is a program transformation technique which, tries to **improve the code** by **eliminating unnecessary code lines** and arranging the statements in such a sequence that **speed up the execution without wasting the resources**.

## Advantages

1. Faster execution
2. Better performance
3. Improves the efficiency

# Code Optimization techniques (Machine independent techniques)

## Techniques

1. Compile time evaluation
2. Common sub expressions elimination
3. Code Movement or Code Motion
4. Reduction in Strength
5. Dead code elimination

# Compile time evaluation

- ▶ Compile time evaluation means shifting of computations from run time to compile time.
- ▶ There are two methods used to obtain the compile time evaluation.

## Folding

- ▶ In the folding technique the computation of constant is done at compile time instead of run time.

Example :  $\text{length} = (22/7)*d$

- ▶ Here folding is implied by performing the computation of  $22/7$  at compile time.

## Constant propagation

- ▶ In this technique the value of variable is replaced and computation of an expression is done at compilation time.

Example :  $\text{pi} = 3.14; r = 5;$

$\text{Area} = \text{pi} * r * r;$

- ▶ Here at the compilation time the value of pi is replaced by 3.14 and r by 5 then computation of  $3.14 * 5 * 5$  is done during compilation.

# Common sub expressions elimination

- ▶ The common sub expression is an expression appearing repeatedly in the program which is computed previously.
- ▶ If the operands of this sub expression do not get changed at all then result of such sub expression is used instead of re-computing it each time.
- ▶ Example:

t1 := 4 * i
t2 := a + 2
t3 := 4 * j
<del>t4 := 4 * i</del>
t5:= n
t6 := b[ <sup>t1</sup> ] + t5

Before Optimization



# Code Movement or Code Motion

- ▶ Optimization can be obtained by **moving some amount of code outside the loop** and placing it just before entering in the loop.
- ▶ It won't have any difference if it executes inside or outside the loop.
- ▶ This method is also called **loop invariant computation**.
- ▶ Example:

```
While(i<=max-1)
{
    sum=sum+a[i];
}
```



Before Optimization

After Optimization

# Reduction in Strength

- ▶ priority of certain operators is higher than others.
- ▶ For instance strength of \* is higher than +.
- ▶ In this technique the higher strength operators can be replaced by lower strength operators.
- ▶ Example:

A=A\*2

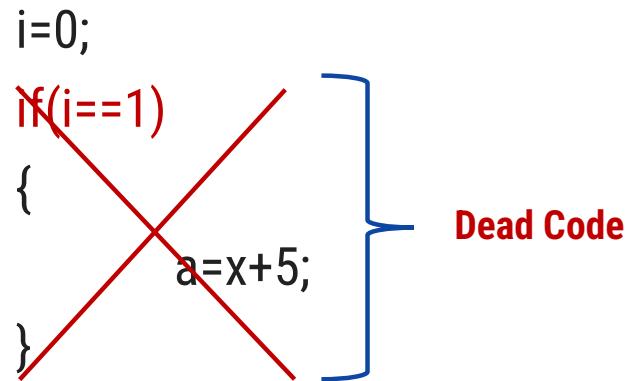
Before Optimization

A=A+A

After Optimization

# Dead code elimination

- ▶ The variable is said to be **dead at a point in a program if the value contained into it is never been used.**
- ▶ The code containing such a variable supposed to be a dead code.
- ▶ Example:



- ▶ If statement is a dead code as this condition will never get satisfied hence, statement can be eliminated and optimization can be done.

# Optimization

Machine dependent optimization

# Machine dependent optimization

- ▶ Machine dependent optimization may vary from machine to machine.
- ▶ Machine-dependent optimization is done after the **target code** has been generated and when the code is transformed according to the target machine architecture.
- ▶ Machine-dependent optimizers put efforts to take maximum **advantage** of the memory hierarchy.

## Techniques

1. Register allocation
2. Use of addressing modes
3. Peephole optimization

Number of register may vary from machine to machine.

Used register may be of 32-bit register or 64 bit register.

Addressing mode also vary from machine to machine.

# Peephole optimization

- ▶ Peephole optimization is a simple and effective technique for locally improving target code.
- ▶ This technique is applied to improve the performance of the target program by examining the short sequence of target instructions (called the peephole) and replacing these instructions by shorter or faster sequence whenever possible.
- ▶ Peephole is a **small, moving window** on the target program.

# Redundant Loads & Stores

- ▶ Especially the redundant loads and stores can be eliminated in following type of transformations.
- ▶ Example:

MOV R0,x

MOV x,R0

- ▶ We can eliminate the second instruction since x is in already R0.

# Flow of Control Optimization

- ▶ The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations.
- ▶ We can replace the jump sequence.

*Goto L1*  
.....  
*L1: goto L2*

- ▶ It may be possible to eliminate the statement **L1: goto L2** provided it is preceded by an unconditional jump. Similarly, the sequence can be replaced by:

*If a<b goto L1*  
.....  
*L1: goto L2*

# Algebraic simplification

- ▶ Peephole optimization is an effective technique for algebraic simplification.
- ▶ The statements such as  $x = x + 0$  or  $x := x * 1$  can be eliminated by peephole optimization.

# Reduction in strength

- ▶ Certain machine instructions are cheaper than the other.
- ▶ In order to improve performance of the intermediate code we can **replace** these **instructions by equivalent cheaper instruction**.
- ▶ For example,  $x^2$  is cheaper than  $x * x$ .
- ▶ Similarly addition and subtraction are cheaper than multiplication and division. So we can add effectively equivalent addition and subtraction for multiplication and division.

# Machine idioms

- ▶ The target instructions have equivalent machine instructions for performing some operations.
- ▶ Hence we can replace these target instructions by equivalent machine instructions in order to improve the efficiency.
- ▶ Example: Some machines have **auto-increment or auto-decrement addressing modes**.  
(Example : INC i)
- ▶ These modes can be used in code for statement like **i=i+1**.

# References

## Books:

### 1. **Compilers Principles, Techniques and Tools, PEARSON Education (Second Edition)**

Authors: Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

### 2. **Compiler Design, PEARSON (for Gujarat Technological University)**

Authors: Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman

# Thank You



## Unit – 8

# Instruction-Level Parallelism



**Prof. Dixita B. Kagathara**

Computer Engineering Department

Darshan Institute of Engineering & Technology, Rajkot

✉ dixita.kagathara@darshan.ac.in

📞 +91 - 97277 47317 (CE Department)

# Topics to be covered

- Processor Architectures
- Code scheduling constraints
- Basic block Scheduling
- Pass structure of Assembler

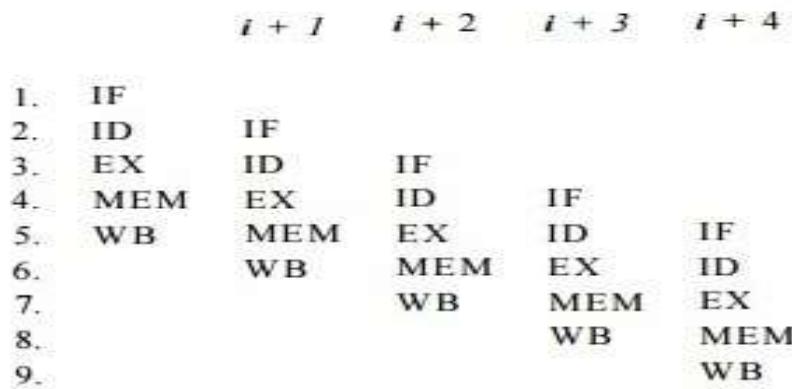
# Processor Architecture

# Processor Architecture

- When we think of instruction-level parallelism, we usually imagine a processor issuing several operations in a single clock cycle. In fact, it is possible for a machine to issue just one operation per clock and yet achieve instruction-level parallelism using the concept of pipelining.

# Instruction Pipelines and Branch Delays

- ▶ Practically every processor, be it a high-performance supercomputer or a standard machine, uses an instruction pipeline. With an instruction pipeline, a new instruction can be fetched every clock while preceding instructions are still going through the pipeline.
- ▶ Shown in below Fig. is a simple 5-stage instruction pipeline: it first fetches the instruction (IF), decodes it (ID), executes the operation (EX), accesses the memory (MEM), and writes back the result (WB).



- ▶ The figure shows how instructions  $i$ ,  $i + 1$ ,  $i + 2$ ,  $i + 3$ , and  $i + 4$  can execute at the same time. Each row corresponds to a clock tick, and each column in the figure specifies the stage each instruction occupies at each clock tick.

# Instruction Pipelines and Branch Delays

- ▶ If the result from an instruction is available by the time the succeeding instruction needs the data, the processor can issue an instruction every clock.
- ▶ Branch instructions are especially problematic because until they are fetched, decoded and executed, the processor does not know which instruction will execute next. Many processors speculatively fetch and decode the immediately succeeding instructions in case a branch is not taken. When a branch is found to be taken, the instruction pipeline is emptied and the branch target is fetched.
- ▶ Thus, taken branches introduce a delay in the fetch of the branch target and introduce "hiccups" in the instruction pipeline. Advanced processors use hard-ware to predict the outcomes of branches based on their execution history and to pre-fetch from the predicted target locations. Branch delays are nonetheless observed if branches are mis-predicted.

# Pipelined Execution



- ▶ Some instructions take several clocks to execute.
- ▶ One common example is the memory-load operation. Even when a memory access hits in the cache, it usually takes several clocks for the cache to return the data.
- ▶ We say that the execution of an instruction is pipelined if succeeding instructions not dependent on the result are allowed to proceed.
- ▶ Thus, even if a processor can issue only one operation per clock, several operations might be in their execution stages at the same time.
- ▶ If the deepest execution pipeline has  $n$  stages, potentially  $n$  operations can be "in flight" at the same time.
- ▶ Note that not all instructions are fully pipelined. While floating-point adds and multiplies often are fully pipelined, floating-point divides, being more complex and less frequently executed, often are not.
- ▶ Most general-purpose processors dynamically detect dependences between consecutive instructions and automatically stall the execution of instructions if their operands are not available. Some processors, especially those embedded in hand-held devices, leave the dependence checking to the software in order to keep the hardware simple and power consumption low. In this case, the compiler is responsible for inserting "noop" instructions in the code if necessary to assure that the results are available when needed.

# Multiple Instruction Issue

- ▶ By issuing several operations per clock, processors can keep even more operations in flight. The largest number of operations that can be executed simultaneously can be computed by multiplying the instruction issue width by the average number of stages in the execution pipeline.
- ▶ Like pipelining, parallelism on multiple-issue machines can be managed either by software or hardware.
- ▶ Machines that rely on software to manage their parallelism are known as VLIW (Very-Long-Instruction-Word) machines, while those that manage their parallelism with hardware are known as superscalar machines.
- ▶ Simple hardware schedulers execute instructions in the order in which they are fetched.
- ▶ If a scheduler comes across a dependent instruction, it and all instructions that follow must wait until the dependences are resolved (i.e., the needed results are available).
- ▶ Such machines obviously can benefit from having a static scheduler that places independent operations next to each other in the order of execution.

# Multiple Instruction Issue

- ▶ More sophisticated schedulers can execute instructions "out of order."
- ▶ Operations are independently stalled and not allowed to execute until all the values they depend on have been produced.
- ▶ Even these schedulers benefit from static scheduling, because hardware schedulers have only a limited space in which to buffer operations that must be stalled. Static scheduling can place independent operations close together to allow better hardware utilization.
- ▶ More importantly, regardless how sophisticated a dynamic scheduler is, it cannot execute instructions it has not fetched. When the processor has to take an unexpected branch, it can only find parallelism among the newly fetched instructions.
- ▶ The compiler can enhance the performance of the dynamic scheduler by ensuring that these newly fetched instructions can execute in parallel.

# Code-Scheduling Constraints

# Code-Scheduling Constraints

- ▶ Code scheduling is a form of program optimization that applies to the machine code that is produced by code generator.
- ▶ Code scheduling subject to three kinds of constraints:
  1. **Control dependence constraints:** All the operations executed in the original program must be executed in the optimized one.
  2. **Data dependence constraints:** The operations in the optimized program must produce the same result as the corresponding ones in the original program.
  3. **Resource constraints:** The schedule must not oversubscribe the resources one the machine.

# Code-Scheduling Constraints

- ▶ These scheduling constraints guarantee that the optimized program produces the same result as the original.
- ▶ However, because code scheduling changes the order in which the operations execute, the state of the memory at any one point may not match any of the memory states in a sequential execution.
- ▶ This situation is a problem if a program's execution is interrupted by, for example, a thrown exception or a user interested breakpoint.
- ▶ Optimized programs are therefore harder to debug.

# Basic-Block Scheduling

# Basic-Block Scheduling

1. Data-Dependence Graphs
2. List Scheduling of Basic Blocks
3. Prioritized Topological Orders

# Data-Dependence Graphs

- ▶ We represent each basic block of machine instructions by a data-dependence graph,  $G = (N, E)$ , having a set of nodes  $N$  representing the operations in the machine instructions in the block and a set of directed edges  $E$  representing the data-dependence constraints among the operations. The nodes and edges of  $G$  are constructed as follows:
  1. Each operation  $n$  in  $N$  has a resource-reservation table  $RT_n$ , whose value is simply the resource-reservation table associated with the operation type of  $n$ .
  2. Each edge  $e$  in  $E$  is labeled with delay  $d_e$  indicating that the destination node must be issued no earlier than  $d_e$  clocks after the source node is issued. Suppose operation  $n_1$  is followed by operation  $n_2$ , and the same location is accessed by both, with latencies  $l_1$  and  $l_2$  respectively. That is, the location's value is produced  $l_1$  clocks after the first instruction begins, and the value is needed by the second instruction  $l_2$  clocks after that instruction begins. Then, there is an edge  $n_1 \rightarrow n_2$  in  $E$  labeled with delay  $l_1 - l_2$ .

# List Scheduling of Basic Blocks

- ▶ The simplest approach to scheduling basic blocks involves visiting each node of the data-dependence graph in "prioritized topological order."
- ▶ Since there can be no cycles in a data-dependence graph, there is always at least one topological order for the nodes. However, among the possible topological orders, some may be preferable to others.
- ▶ There is some algorithm for picking a preferred order.

# Prioritized Topological Orders

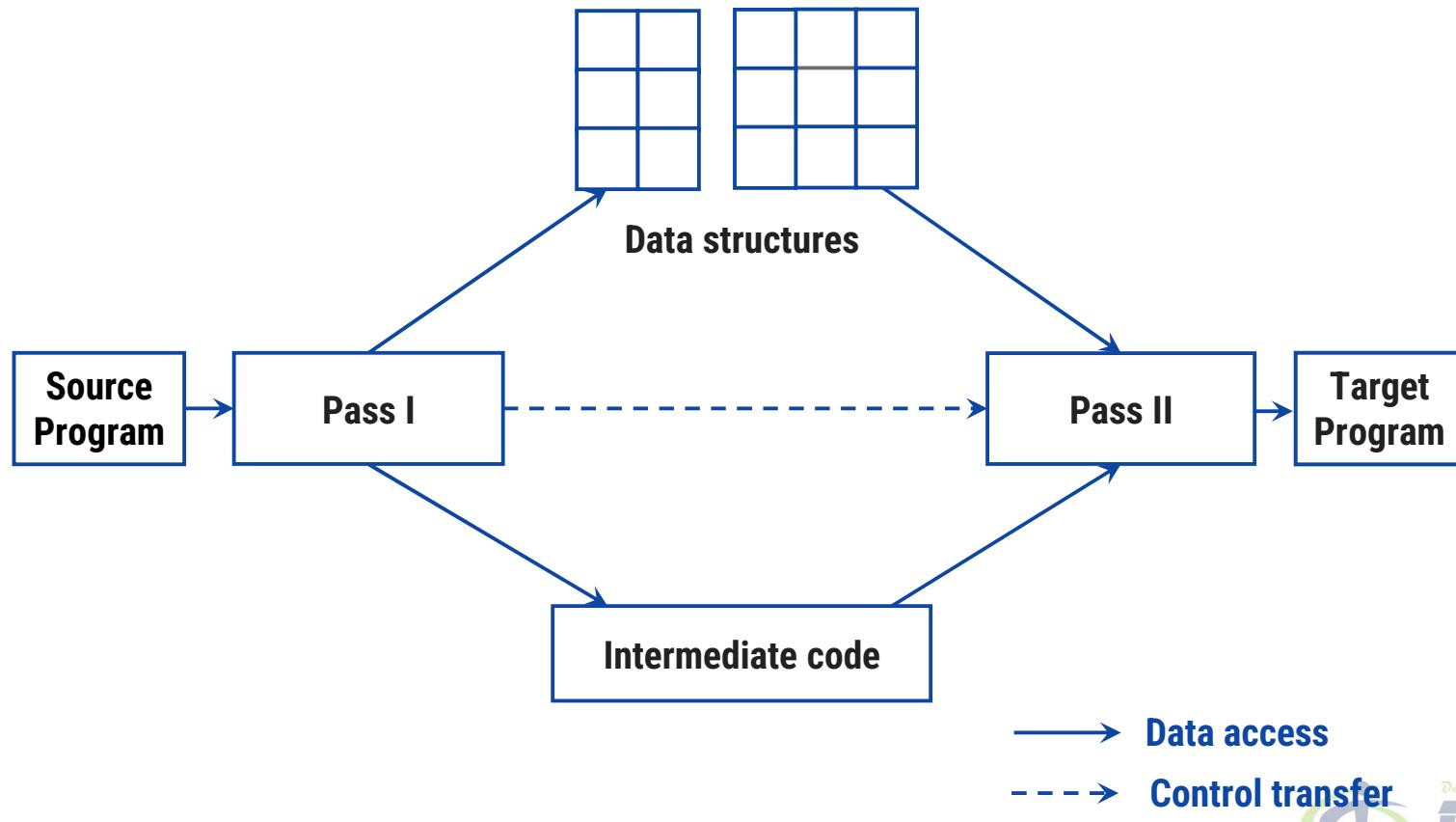
- ▶ List scheduling does not backtrack; it schedules each node once and only once. It uses a heuristic priority function to choose among the nodes that are ready to be scheduled next. Here are some observations about possible prioritized orderings of the nodes:
- ▶ Without resource constraints, the shortest schedule is given by the critical path, the longest path through the data-dependence graph. A metric useful as a priority function is the height of the node, which is the length of a longest path in the graph originating from the node.
- ▶ On the other hand, if all operations are independent, then the length of the schedule is constrained by the resources available. The critical resource is the one with the largest ratio of uses to the number of units of that resource available. Operations using more critical resources may be given higher priority.
- ▶ Finally, we can use the source ordering to break ties between operations; the operation that shows up earlier in the source program should be scheduled first

# Pass structure of assembler

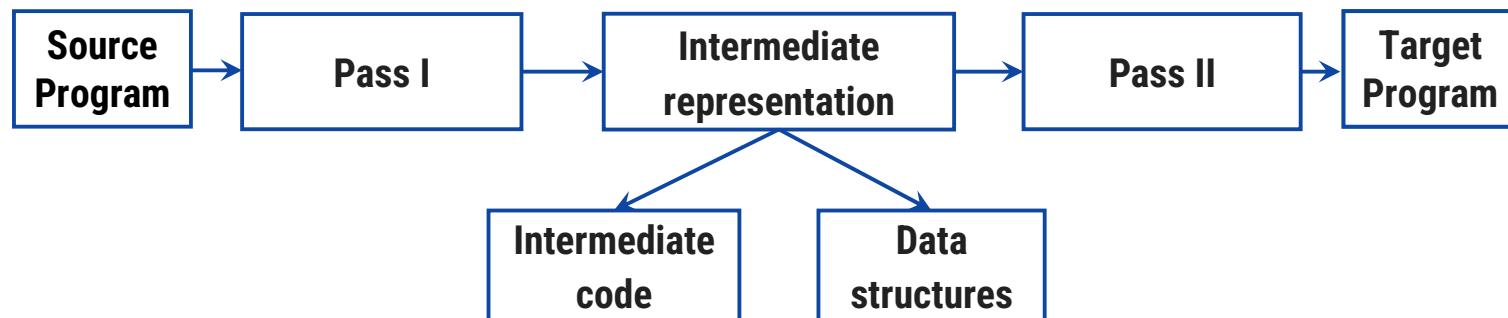
# Pass structure of assembler

- ▶ A complete scan of the program is called pass.
- ▶ Types of assembler are:
  1. Two pass assembler (Two pass translation)
  2. Single pass assembler (Single pass translation)

# Two pass assembler (Two pass translation)

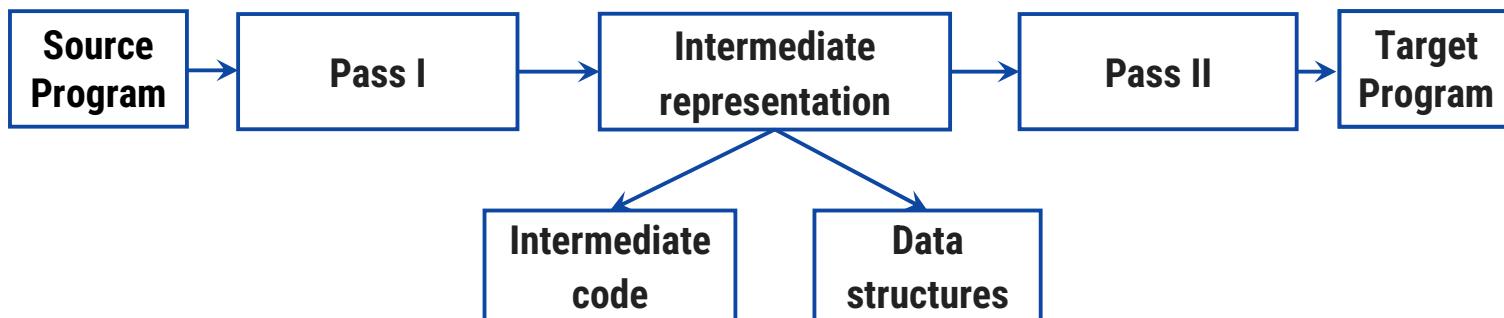


# Two pass assembler (Two pass translation)



- ▶ The first pass performs **analysis of the source program**.
- ▶ The first pass performs **Location Counter processing** and records the addresses of symbols in the symbol table.
- ▶ It **constructs intermediate representation** of the source program.
- ▶ Intermediate representation consists of following two components:
  1. **Intermediate code**
  2. **Data structures**

# Two pass assembler (Two pass translation)



- ▶ The second pass synthesizes the target program by using address information recorded in the symbol table.
- ▶ Two pass translation handles a forward reference to a symbol naturally because the address of each symbol would be known before program synthesis begins.

Use of a Symbol that precedes  
its definition in a program.

# One pass assembler (One pass translation)

- ▶ A one pass assembler **requires one scan** of the source program to generate machine code.
- ▶ Location counter processing, symbol table construction and target code generation proceed in single pass.
- ▶ The issue of **forward references** can be solved using a process called **back patching**.

# References

## Books:

### 1. **Compilers Principles, Techniques and Tools, PEARSON Education (Second Edition)**

Authors: Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

### 2. **Compiler Design, PEARSON (for Gujarat Technological University)**

Authors: Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman

# Thank You

