

# Compiler Design - Introduction

Sitara K.

[sitara@nitt.edu](mailto:sitara@nitt.edu)

# Objectives

- To introduce the major concept areas in compiler design and know the various phases of the compiler
- To understand the various parsing algorithms and comparison of the same
- To provide practical programming skills necessary for designing a compiler
- To gain knowledge about the various code generation principles
- To understand the necessity for code optimization

# Syllabus

## **UNIT I Introduction to Compilation**

Compilers - Analysis of the source program - Phases of a compiler - Cousins of the Compiler - Grouping of Phases - Compiler construction tools - Lexical Analysis - Role of Lexical Analyzer - Input Buffering - Specification of Tokens.

Lab Component: Tutorial on LEX / FLEX tool, Tokenization exercises using LEX.

## **UNIT II Syntax Analysis**

Role of the parser - Writing Grammars - Context-Free Grammars - Top Down parsing - Recursive Descent Parsing - Predictive Parsing - Bottom-up parsing - Shift Reduce Parsing - Operator Precedent Parsing - LR Parsers - SLR Parser - Canonical LR Parser - LALR Parser.

Lab Component: Tutorial on YACC tool, Parsing exercises using YACC tool.

## **UNIT III Intermediate Code Generation**

Intermediate languages - Declarations - Assignment Statements - Boolean Expressions - Case Statements - Back patching - Procedure calls.

Lab Component: A sample language like C-lite is to be chosen. Intermediate code generation exercises for assignment statements, loops, conditional statements using LEX/YACC.

## **UNIT IV Code Optimization and Run Time Environments**

Introduction - Principal Sources of Optimization - Optimization of basic Blocks - DAG representation of Basic Blocks - Introduction to Global Data Flow Analysis - Runtime Environments - Source Language issues - Storage Organization - Storage Allocation strategies - Access to non-local names - Parameter Passing - Error detection and recovery.

Lab Component: Local optimization to be implemented using LEX/YACC for the sample language.

## **UNIT V Code Generation**

Issues in the design of code generator - The target machine - Runtime Storage management - Basic Blocks and Flow Graphs - Next-use Information - A simple Code generator - DAG based code generation - Peephole Optimization.

Lab Component: DAG construction, Simple Code Generator implementation, DAG based code generation using LEX/YACC for the sample language.

# Course Outcomes

- Apply the knowledge of LEX & YACC tool to develop a scanner and parser
- Design and develop software system for backend of the compiler
- Suggest the necessity for appropriate code optimization techniques
- Conclude the appropriate code generator algorithm for a given source language
- Design a compiler for any programming language

# Text Books

- Alfred V. Aho, Jeffrey D Ullman, “Compilers: Principles, Techniques and Tools”, Pearson Education Asia, 2012.
- Jean Paul Tremblay, Paul G Serenson, “The Theory and Practice of Compiler Writing”, BS Publications, 2005.
- Dhamdhere, D. M., “Compiler Construction Principles and Practice”, Second Edition, Macmillan India Ltd., New Delhi, 2008.

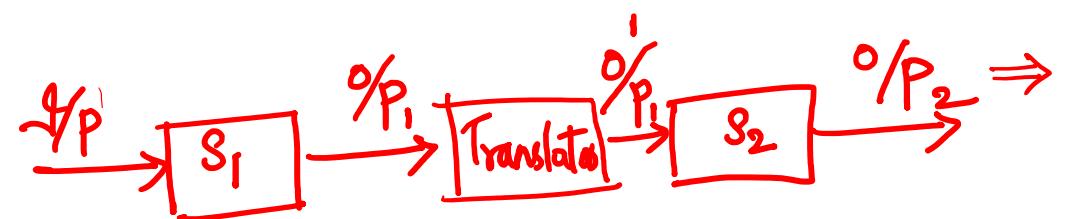
# History

- Software – essential component of the current scenario
- Early software was written in assembly languages
- Drawbacks
  - Very difficult to remember instructions
- The benefits of reusing software on different CPUs became greater than the cost of designing compiler
- Very cumbersome to write
- Need for a software that will understand human language

## Language Processors.



- A **translator** inputs and then converts a **source program** into an **object or target program**.
- **Source program** is written in one language
- **Object program** belongs to an object language
- A translators could be: Assembler, Compiler, Interpreter



# Options

- Design an interpreter / translator to convert human language to machine language
  - Difficult – Parsing, interpreting, ambiguous

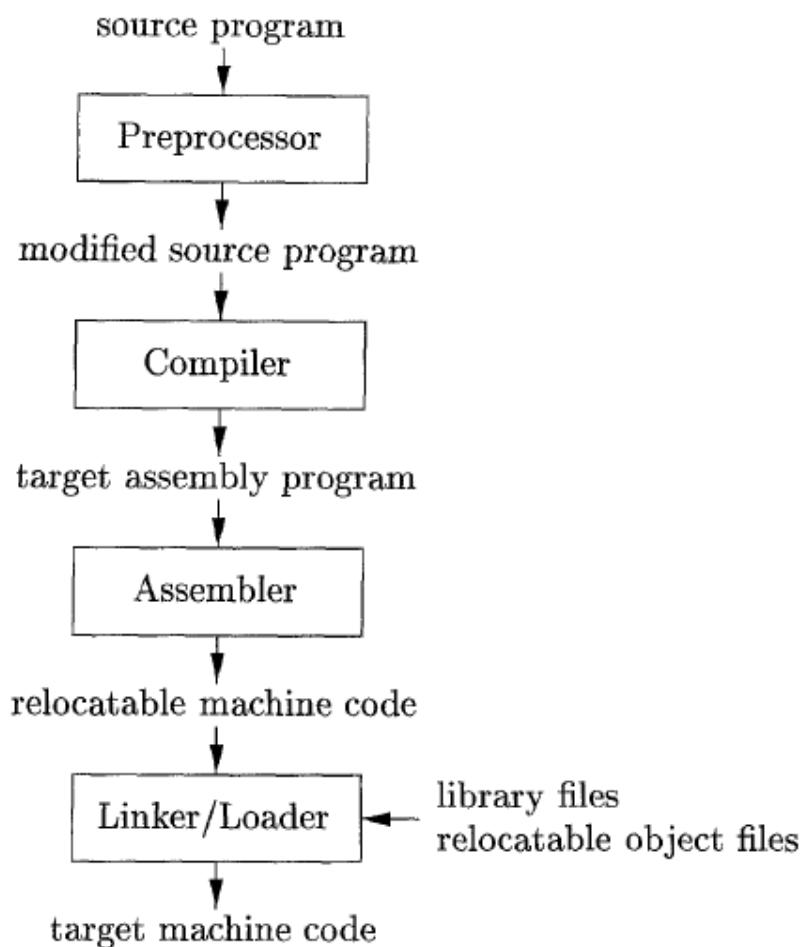
# Options

- Design a compiler that will understand high level (not necessarily English) language to assembly language
  - Relatively simpler, but need a mapping of the high level language to assembly language

# Options

- Design an assembler that converts assembly language to machine language
  - Target language need to be specified. Output of the various compilers to be known prior time

# Language Processing System



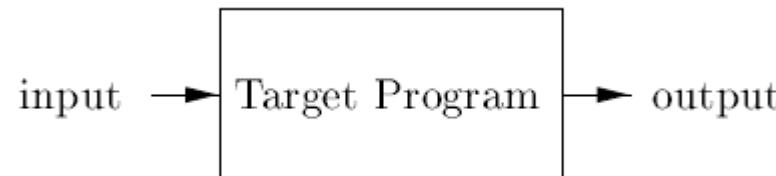
#define MAX 100

# Compiler

- The first real compiler
  - FORTRAN compilers of the late 1950s
  - 18 person-years to build

# What are Compilers?

- A compiler acts as a translator, transforming human-oriented programming languages into computer-oriented machine languages.
- No concern about machine-dependent details for programmer



# Compiler

- Processes source program
- Prompts errors in source program
- Recovers / Corrects the errors
- Produce assembly language program
- Compiler + assembler – Converts this to relocatable machine code

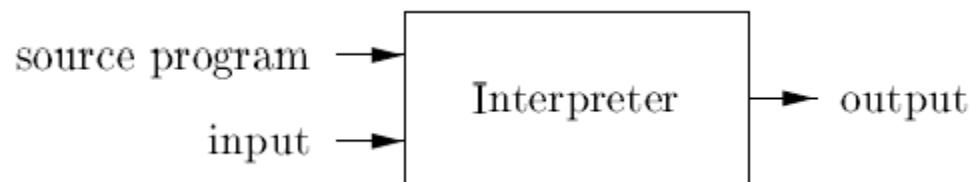
```
f1( )  
{   _____> p;  
    p = f2();  
}  
f2( )  
{  
    f3( );  
    a = a;
```

# Compiler - Overview

- Translates a source program written in a High-Level Language (HLL) such as Pascal, C++ into computer's machine language (Low-Level Language (LLL))
- The time of conversion from source program into object program is called **compile time**
- The object program is executed at **run time**

# Interpreter

- Language processor that executes the operation as specified in the source program
- Inputs are supplied by the user
- Processes an internal form of the source program and data at the same time (at run time); no object program is generated.

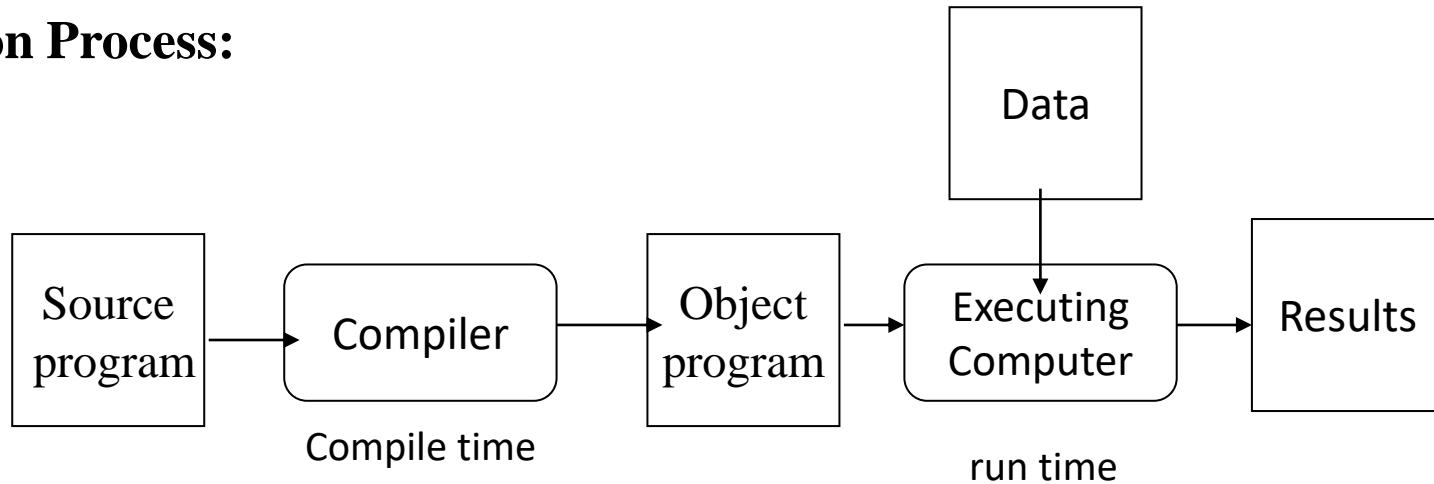


# Compiler vs Interpreter

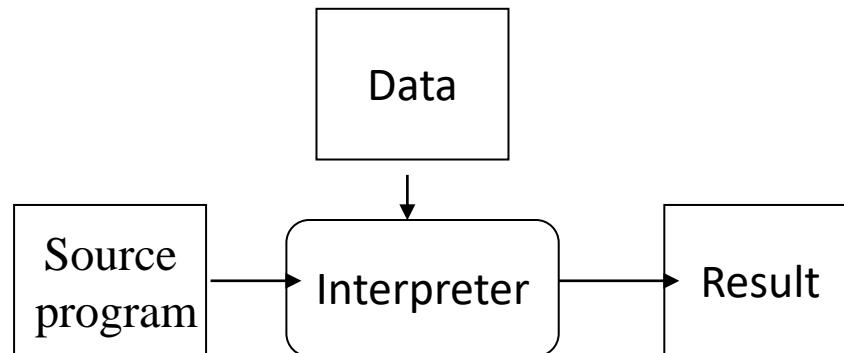
- The machine-language target program produced by a compiler is much faster than an interpreter at mapping inputs to outputs
- An interpreter, is better with error diagnostics as it executes the source program statement by statement

# Overview of Compilers

## Compilation Process:



## Interpretive Process:



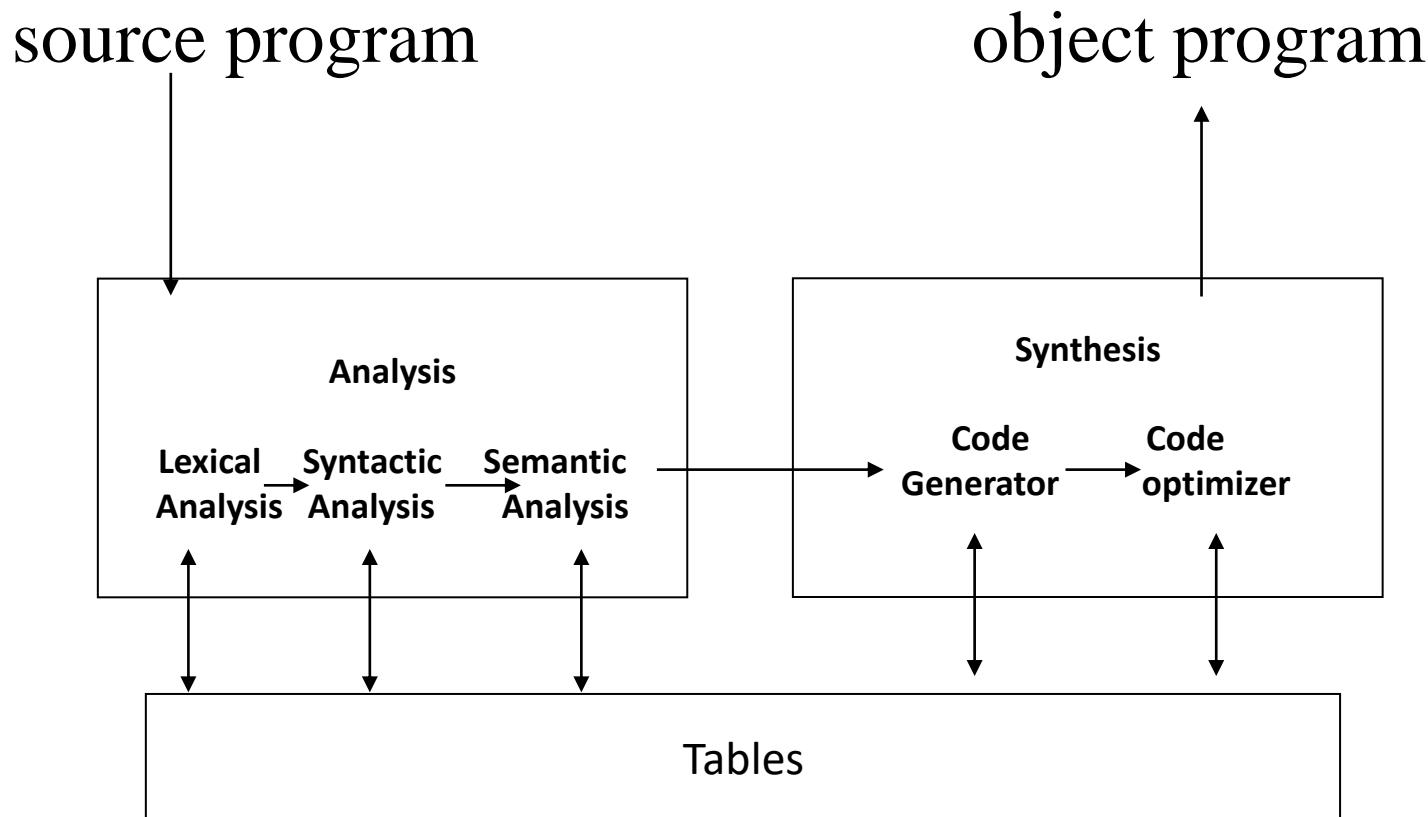
# Compiler

- **Analysis** of source program: The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program.
- **Synthesis** of its corresponding program: constructs the desired target program from the intermediate representation and the information in the symbol table.
- The analysis part is often called the **front end** of the compiler; the synthesis part is the **back end**.

# Compiler

- Front End – Language Dependent – Depends on the source language and Target Independent
- Back End – Target Dependent – Depends on the target language but Source independent

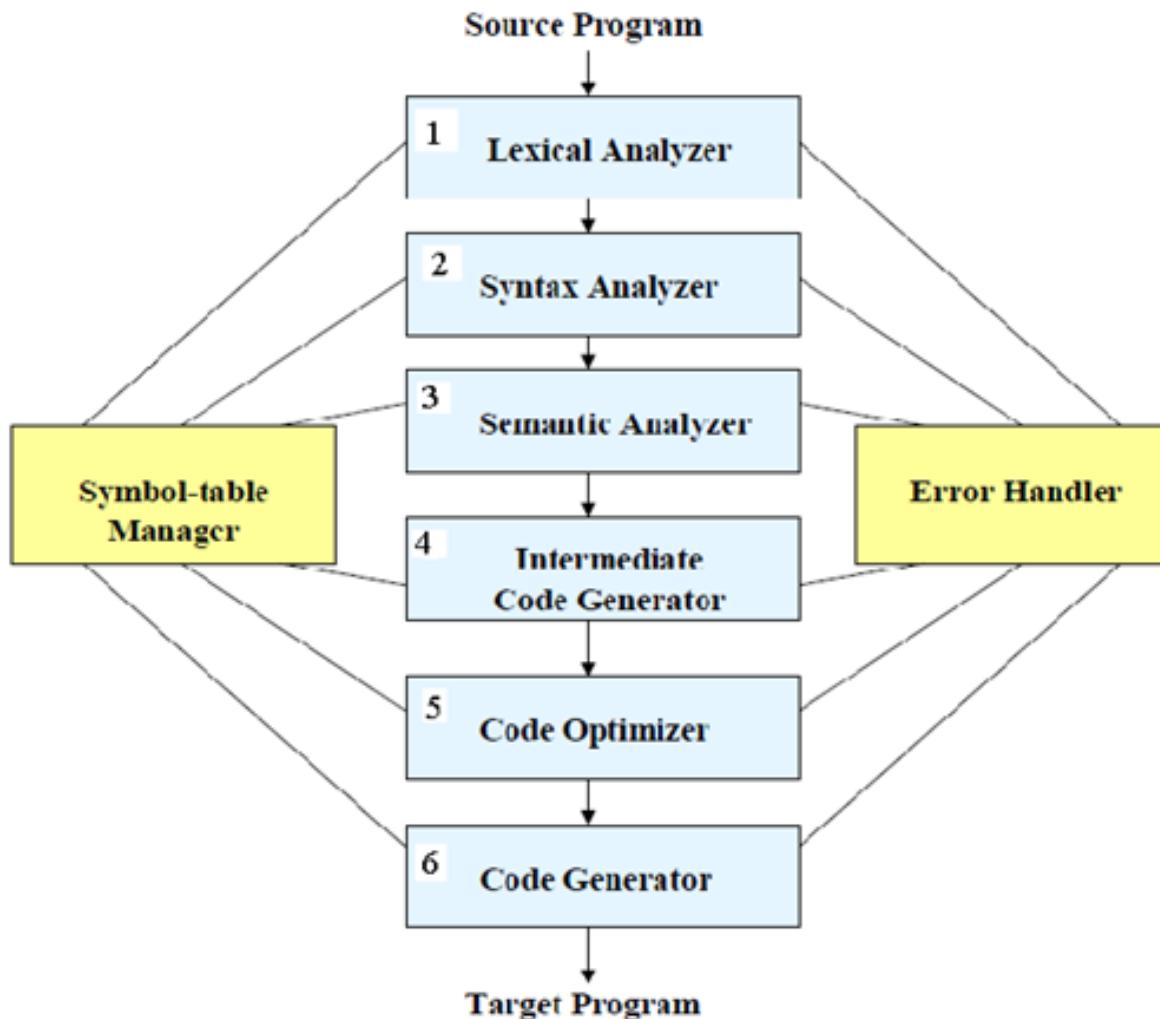
# Flow of Compiler



# Compiler Passes

- How many passes should the compiler go through?
- One for analysis and one for synthesis?
- One for each division of the analysis and synthesis?
- The work done by a compiler is grouped into phases

# Phases of the compiler



## Lexical Analysis (scanner): The first phase of a compiler

- Lexical analyzer reads the stream of characters from the source program and combines the characters into meaningful sequences called ***lexeme***
- For every lexeme, the lexer produces a token of the form which is passed to the next phase of the compiler

(token-name, attribute-value)

C identifiers.

area

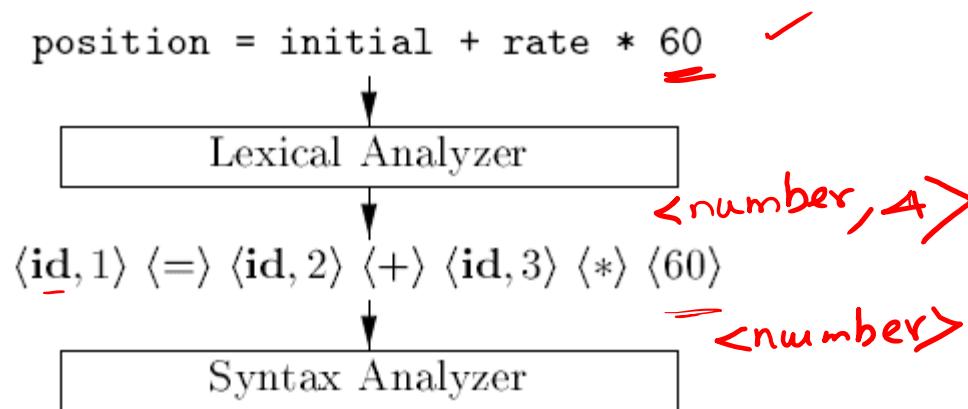
area-1

/area

# Lexical Analysis (scanner): The first phase of a compiler

- Token-name: an abstract symbol is used during syntax analysis, an attribute-value: points to an entry in the symbol table for this token.
- Blanks will be discarded by the lexical analyser

$\langle, \langle=, \rangle, \rangle=$   
 $\text{loop}$   
 $\langle \text{loop} \rangle$   
 $\langle \text{loop}, 5 \rangle$   
 $\langle \text{loop}, \text{GET} \rangle$



SYMBOL TABLE	
1	position
2	initial
3	rate
4	number 60

$\frac{-a=b+c}{\rightarrow a = b + c}$

# Example: position =initial + rate \* 60

1. "position" is a lexeme mapped into a token (id, 1), where id is an abstract symbol standing for identifier and 1 points to the symbol table entry for position. The symbol-table entry for an identifier holds information about the identifier, such as its name and type.
2. = is a lexeme that is mapped into the token (=). Since this token needs no attribute-value, we have omitted the second component. For notational convenience, the lexeme itself is used as the name of the abstract symbol.
3. "initial" is a lexeme that is mapped into the token (id, 2), where 2 points to the symbol-table entry for initial

4. + is a lexeme that is mapped into the token (+).
5. “rate” is a lexeme mapped into the token (id, 3), where 3 points to the symbol-table entry for rate.
6. \* is a lexeme that is mapped into the token (\*) .
7. 60 is a lexeme that is mapped into the token (60)

# Lexical Analysis

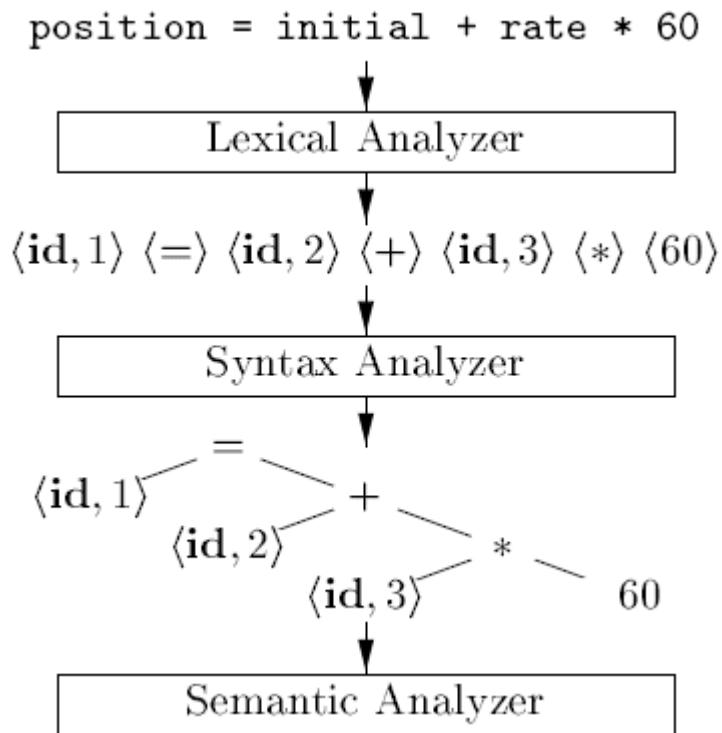
- Interface of the compiler to the outside world
- Scans input program, identifies valid words of the language in it
- Removes extra white spaces, comments etc
- Expand user defined macros
- Reports presence of foreign words
- May perform case conversions
- Generates tokens
- Generally implemented as finite automata

## Syntax Analysis (parser) : The second phase of the compiler

- The parser uses the tokens produced by the lexer to create a tree-like intermediate representation that verifies the grammatical structure of the sequence of tokens
- Works hand-in-hand with lexical analyzer
- Identifies sequence of grammar rules to derive the input program from the start symbol
- A parse tree is constructed
- Error messages are flashed for syntactically incorrect programs

# Syntax Analysis (parser) : The second phase of the compiler

- A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation



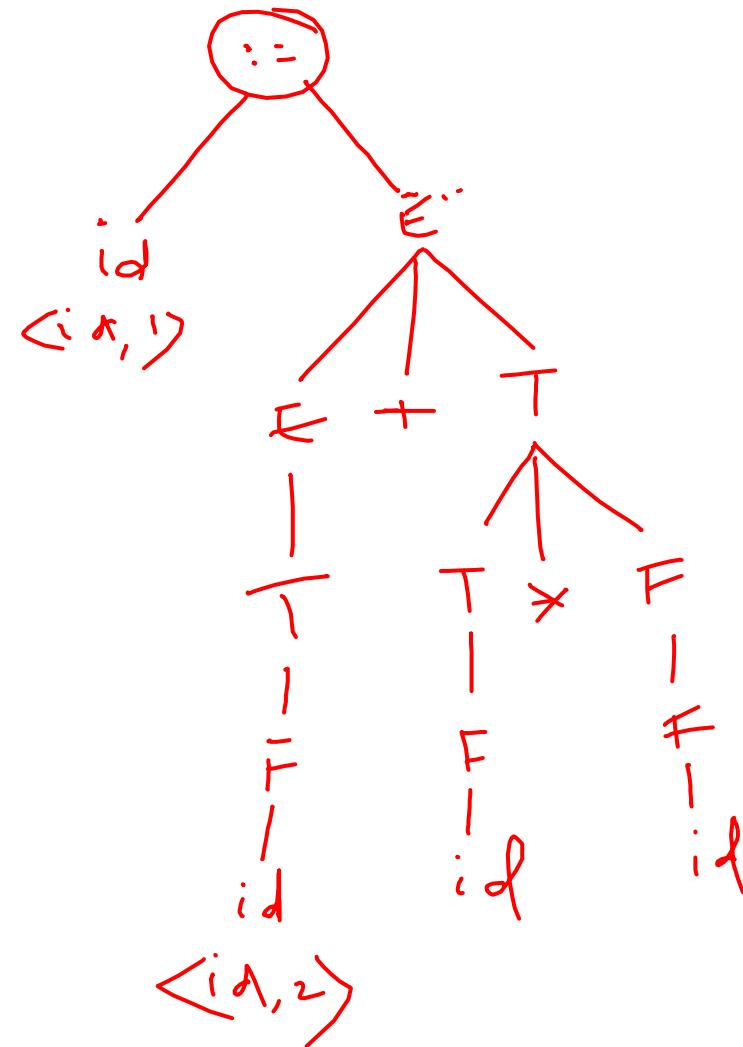
# Syntax phase

- Grammar for assignment expression

$$S \rightarrow id = E \quad -$$
  
$$\underline{E \rightarrow E + T \mid T} \quad -$$

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

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



## **Semantic Analysis: Third phase of the compiler**

- The semantic analyzer uses the output of the parser – syntax tree and the information in the symbol table to check for semantic consistency in the source program
- Gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

# Semantic Analysis: Third phase of the compiler

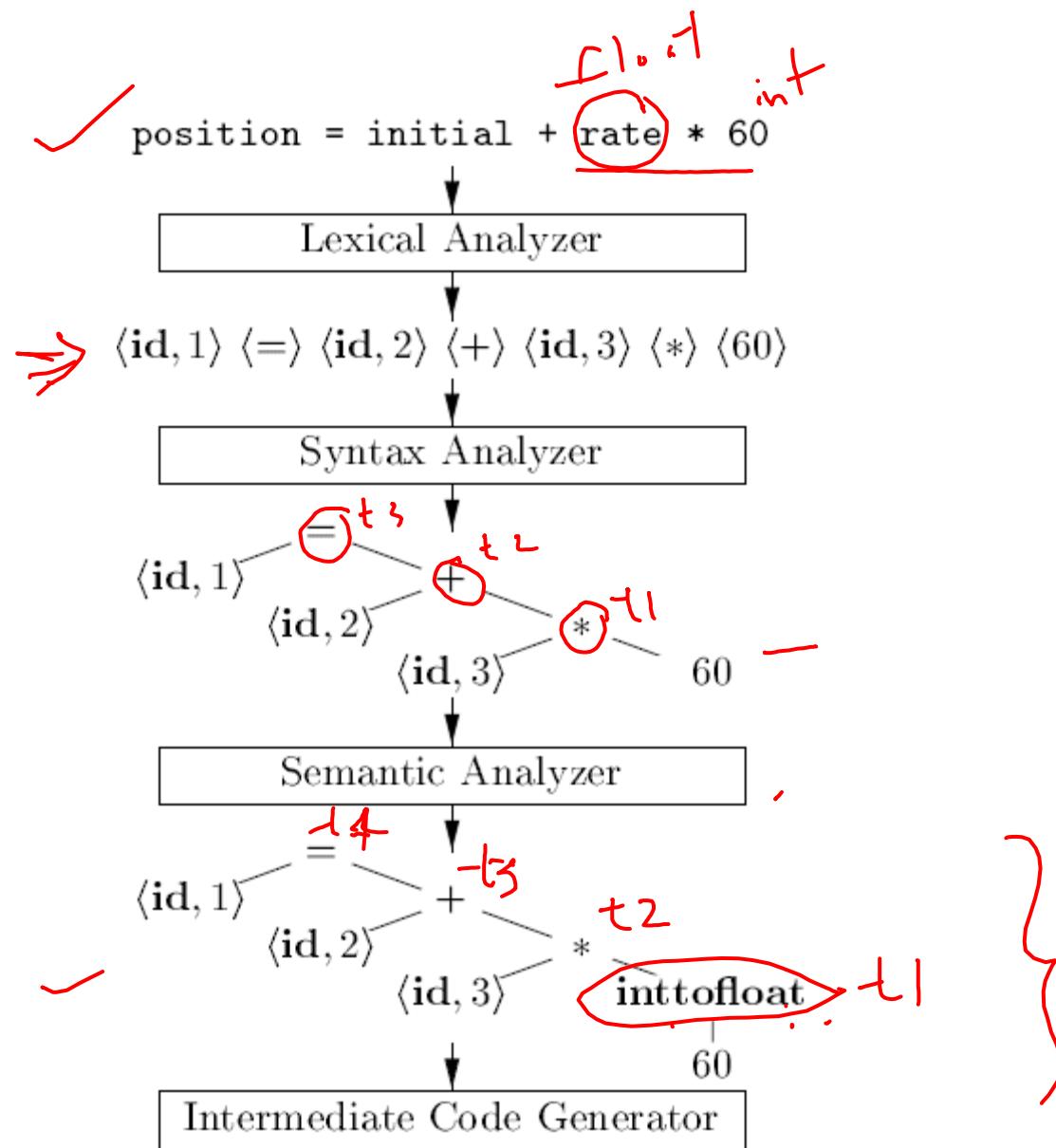
- Type checking, where the compiler checks that each operator has matching operands.
  - Array index need to be an integer; the compiler must identify an error if a floating-point number is used to as an array index
- Scope rules of the language are applied to determine types – static scope or dynamic scope

for (  
  float  
{ int a[10]; i=0;  
      a[i] = 5; }

# Semantic Analysis: Third phase of the compiler

- Coercions – a way of type conversion
- For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers. If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number.

f1. + : + float -  
a = b + c;  
- z



# Intermediate Code Generation: Fourth phase of the compiler

- Optional towards target code generation
- Compilers generate an explicit low-level or machine-like intermediate representation (a program for an abstract machine). This intermediate representation:
  - should be easy to produce
  - should be easy to translate into the target machine
  - Powerful enough to express the programming language constructs
- Helps to retarget the code from one processor to another

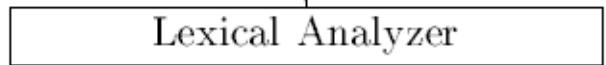
# Intermediate code Generation : Three address code

- A convention for Intermediate code generation is three address code
- Three operands at the most and 2 operators
- Example:
  - $x = y \text{ op } z$
  - $x = \text{op } y$

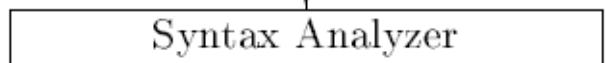
$$x = z - y$$

$$\begin{array}{c} \cancel{x = z + y} \\ \searrow \\ y = y + 1 \end{array}$$

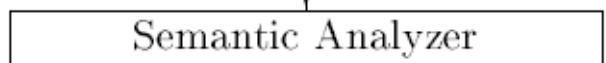
```
position = initial + rate * 60
```



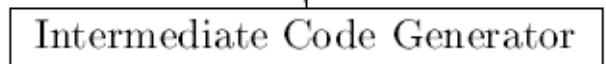
```
⟨id,1⟩ <=⟩ ⟨id,2⟩ <+⟩ ⟨id,3⟩ <*⟩ ⟨60⟩
```



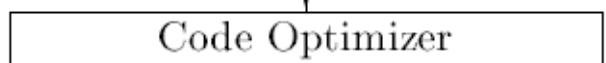
```
⟨id,1⟩ =  
       |  
       <id,2> +  
       |  
       <id,3> *  
       |  
       60
```



```
⟨id,1⟩ =  
       |  
       <id,2> +  
       |  
       <id,3> *  
       |  
       inttofloat  
       |  
       60
```

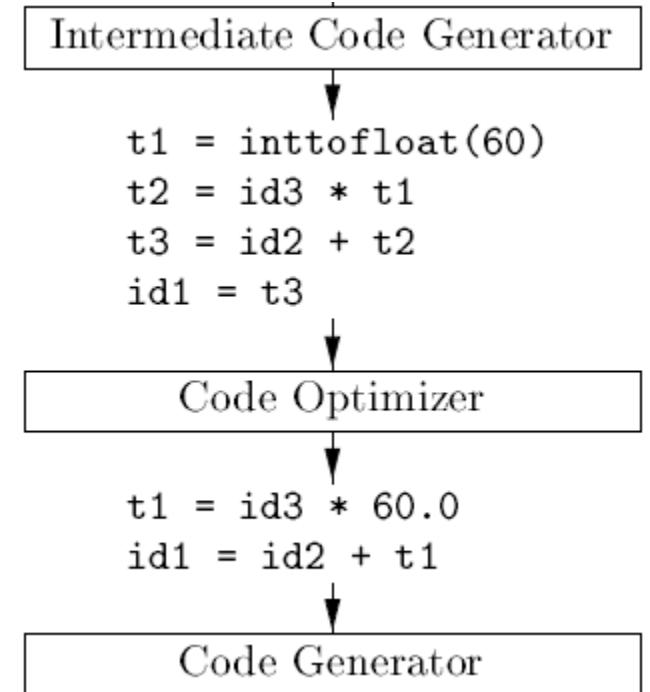


```
t1 = inttofloat(60)  
t2 = id3 * t1  
t3 = id2 + t2  
id1 = t3
```



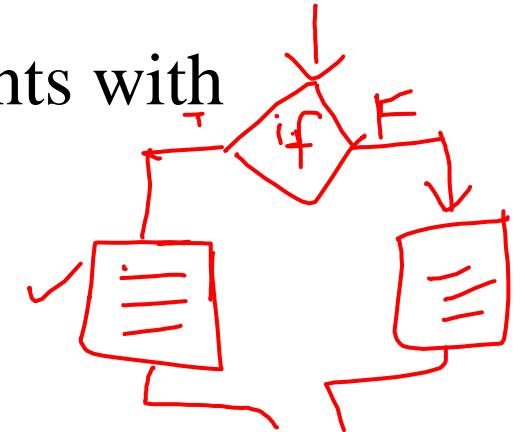
# Code Optimization: - Fifth phase of the compiler

- attempts to improve the intermediate code for better target code
  - faster, shorter code, or target code that consumes less power.
  - simple optimizations that significantly improve the running time of the target program without slowing down compilation



# Code Optimization: - Fifth phase of the compiler

- Automated steps of compiler generate lots of redundant code that can possibly eliminated
  - Code is divided into *basic blocks* – a sequence of statements with single entry and exit
    - *Local optimizations* restrict within a single basic block
    - *Global optimizations* spans across basic blocks
  - Optimize loops, algebraic simplifications, elimination of load-and store are common optimizations



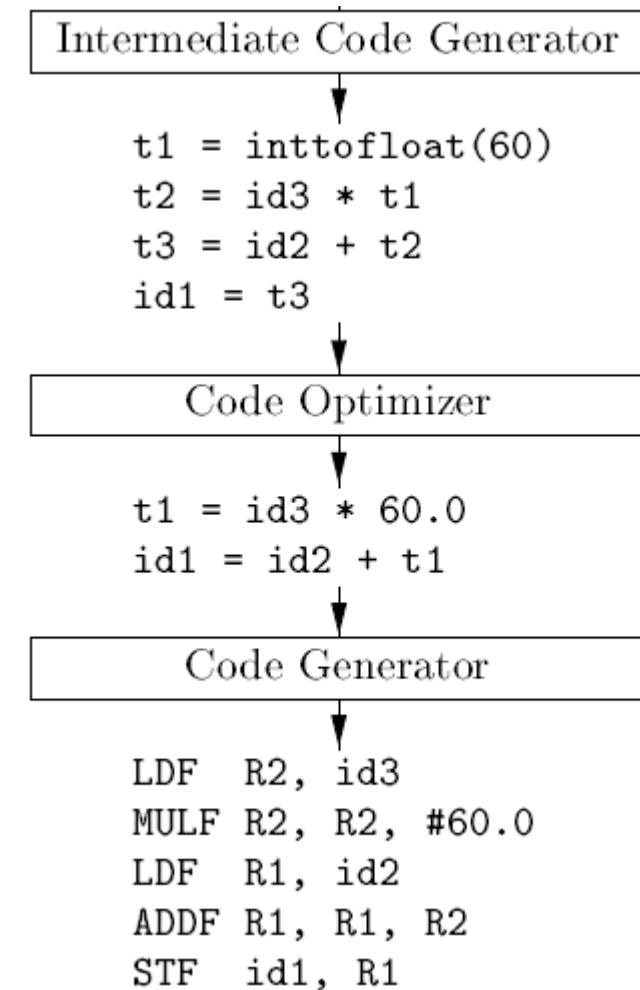
ation of load-and-

$$1 \text{ min} \Rightarrow 1 \text{ sec}$$

$$a = b + \cancel{\frac{d}{10}} + \cancel{\frac{d}{10}} + \cancel{\frac{d}{10}}$$

# Code Generation: Sixth phase of the compiler

- If the target language is machine code, then registers or memory locations are selected for each of the variables used by the program.
- Then, the intermediate instructions are translated into sequences of machine instructions to complete an operation



# Code Generation: Sixth phase of the compiler

- Important consideration of code generation is the assignment of registers to hold variables.
- Choice of instructions involving registers, memory or a mix of the two

# **Symbol-Table Management: - Interaction with all the compiler's phases**

- The symbol table is a data structure containing a record for each variable name (all symbols defined in the source program), with fields for the attributes of the name.
- The data structure is designed to help the compiler to identify and fetch the record for each name quickly
- To store or retrieve data from that record quickly
- Not part of the final code, but used as reference by all phases
- Generally created by lexical and syntax analyzer

# Symbol-Table Management: - Interaction with all the compiler's phases

- attributes may provide information about the storage allocated for a name, its type, its scope, size, relative offset of variables
- Function or Procedure names, the number and types of its arguments, the method of passing each argument and the return type

# Error Handling and Recovery

- An important criteria for selecting the quality of the compiler
- For semantic errors, compiler can proceed
- For syntax errors, parser enters into erroneous state
- Needs to undo some processing already carried out by parser
- A few tokens may need to be discarded to reach a descent state
- Recovery is essential to provide a bunch of errors to the users

## Compiler Phases vs Passes

- Several phases can be implemented as a single pass consist of reading an input file and writing an output file.

# Compiler Phases vs Passes

- A typical multi-pass compiler looks like:
  - First pass: preprocessing, macro expansion
  - Second pass: syntax-directed translation, IR code generation
  - Third pass: optimization
  - Last pass: target machine code generation

# Cousins of Compilers

- Preprocessors
- Assemblers
  - Compiler may produce assembly code instead of generating relocatable machine code directly.

# Cousins of the Compiler

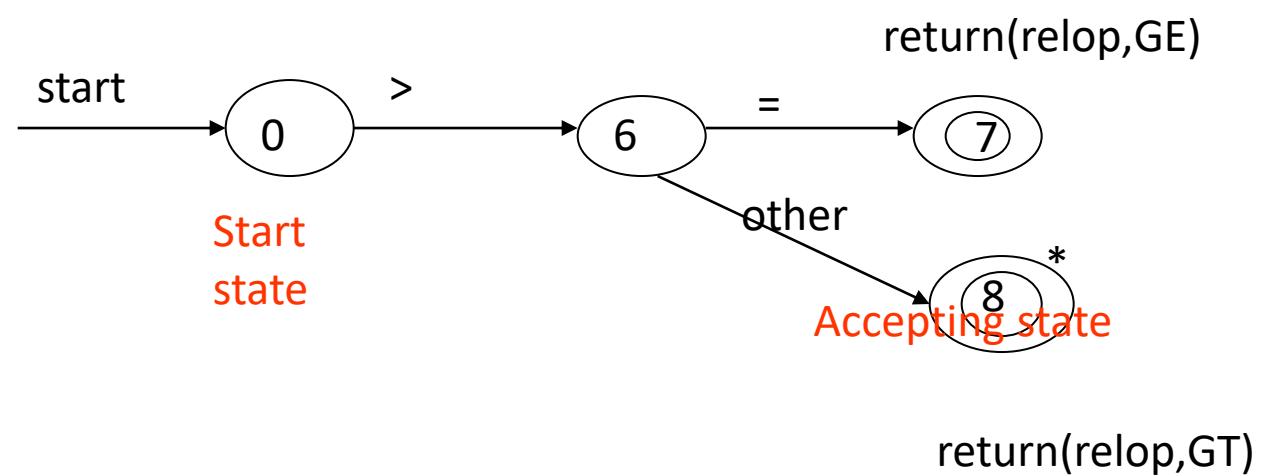
- Loaders and Linkers
  - Loader copies code and data into memory, allocates storage, setting protection bits, mapping virtual addresses, .. Etc
  - Linker handles relocation and resolves symbol references.
- Debugger

# Lexical-Analyser: Automata

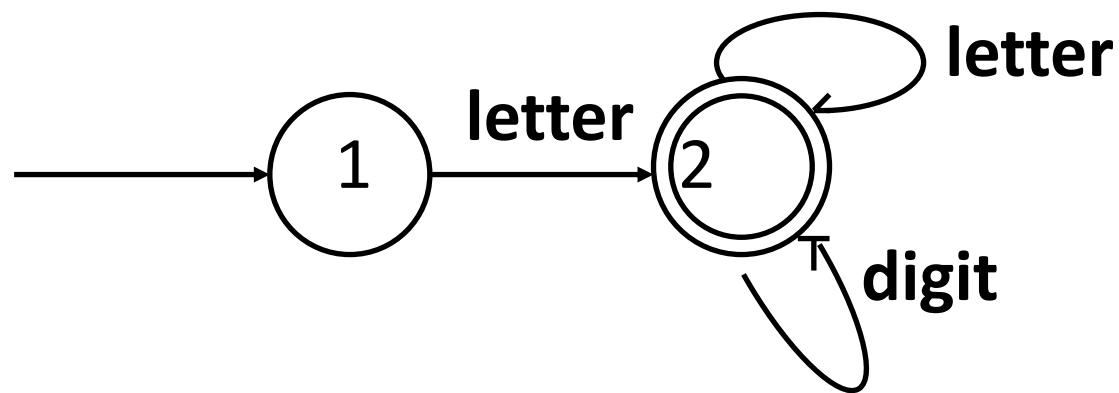
# Automata – Transition Diagrams

- Transition Diagram(Stylized flowchart)
  - Depict the actions that take place when a lexical analyzer is called by the parser to get the next token

# Example NFA



# Example for Identifier



- Which represent the rule:  
**identifier=letter(letter|digit)\***

# Finite Automata

- By default a Deterministic one.
- Five tuple representation  
 $(Q, \Sigma, \delta, q_0, F)$ ,  $q_0$  belongs to  $Q$  and  $F$  is a subset of  $Q$   
 $\delta$  is a mapping from  $Q \times \Sigma$  to  $Q$
- Every string has exactly one path and hence faster string matching

# DFA

- In a DFA, no state has an  $\epsilon$ -transition
- In a DFA, for each state  $s$  and input symbol  $a$ , there is at most one edge labeled  $a$  leaving  $s$
- To describe a FA, we use the transition graph or transition table

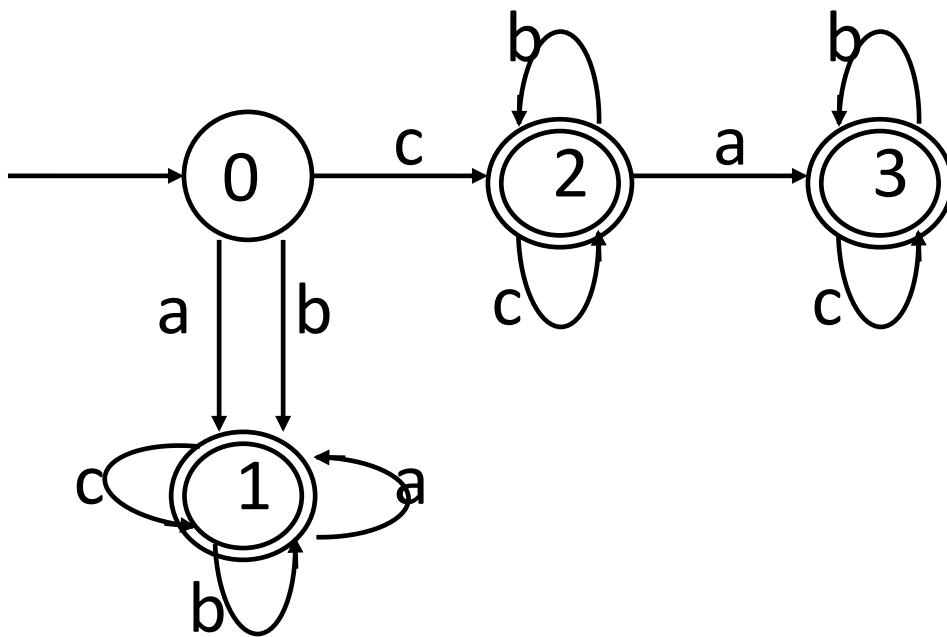
# DFA

- A DFA accepts an input string  $x$  if and only if there is some path in the transition graph from start state to some accepting state

# Example

- Recognition of Tokens
- Construct a DFA  $M$ , which can accept the strings which begin with  $a$  or  $b$ , or begin with  $c$  and contain at most one  $a$ .

## Example



c bbcc  
cccba  
cccaab x

# Non-deterministic Finite automata

- Same as deterministic, gives some flexibility.
- Five tuple representation  
 $(Q, \Sigma, \delta, q_0, F)$ ,  $q_0$  belongs to  $Q$  and  $F$  is a subset of  $Q$   
 $\delta$  is a mapping from  $Q \times \Sigma$  to  $2^Q$
- More time for string matching as multiple paths exist.

# Non-Deterministic Finite automata with $\epsilon$

- Same as NFA. Still more flexible in allowing to change state without consuming any input symbol.
- $\delta$  is a mapping from  $Q \times \Sigma \cup \{\epsilon\}$  to  $2^Q$
- Slower than NFA for string matching

# NFA Some Observations

- In a NFA, the same character can label two or more transitions out of one state;
- In a NFA,  $\epsilon$  is a legal input symbol.
- A DFA is a special case of a NFA

# NFA Some Observations

- A NFA accepts an input string ‘x’ if and only if there is some path in the transition graph from start state to some accepting state. A path can be represented by a sequence of state transitions called moves.
- The language defined by a NFA is the set of input strings it accepts

# RE to DFA

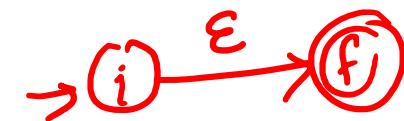
- Regular Expression could be converted to E-NFA using Thompson Construction Algorithm
- E-NFA could be converted to DFA using Subset construction algorithm

# Basic Regular Expression and its NFA

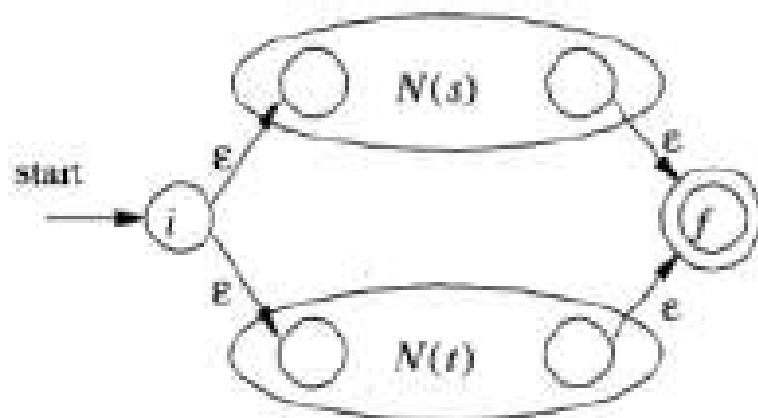


$r = a$

$\epsilon$



# Regular expression – Union operator and its corresponding NFA



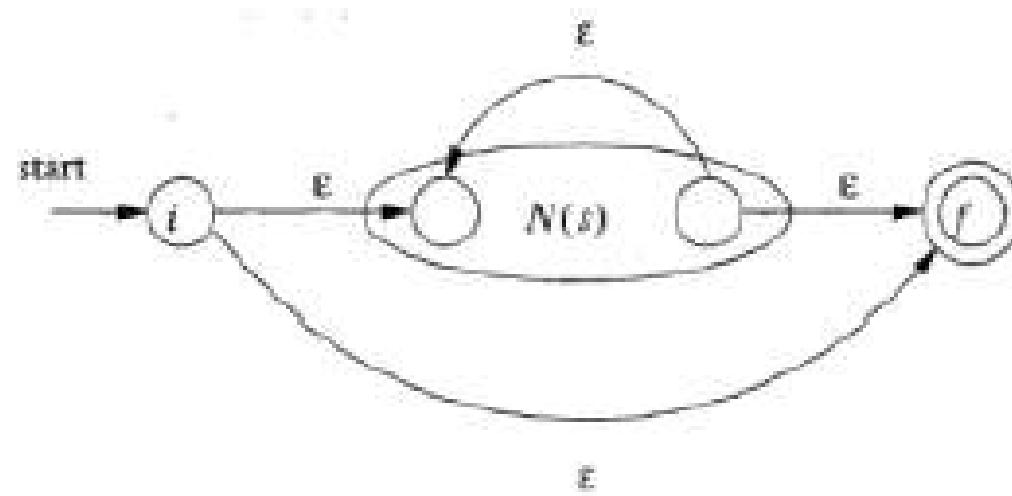
$$r = s | t$$

# Regular expression with concatenation operator and its corresponding NFA



$$r = s t$$

# Regular expression involving kleene closure operator and its NFA

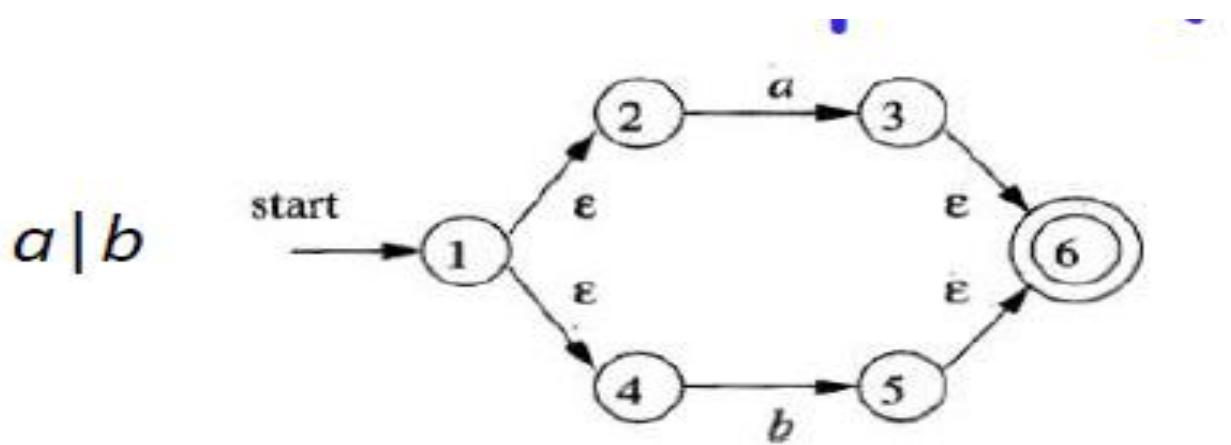


$$r = s^*$$

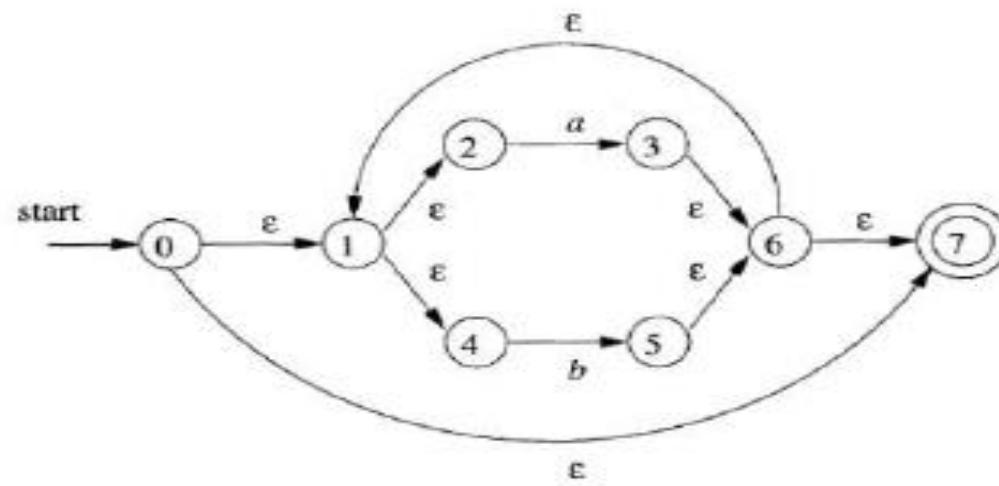
# Algorithm

- Construct the basic NFA for each of the input symbols
- Prioritize the operators  $()$ ,  $*$ ,  $\cdot$ ,  $|$
- Use the discussed variations and form an NFA.

Example :  $(a|b)^*abb$

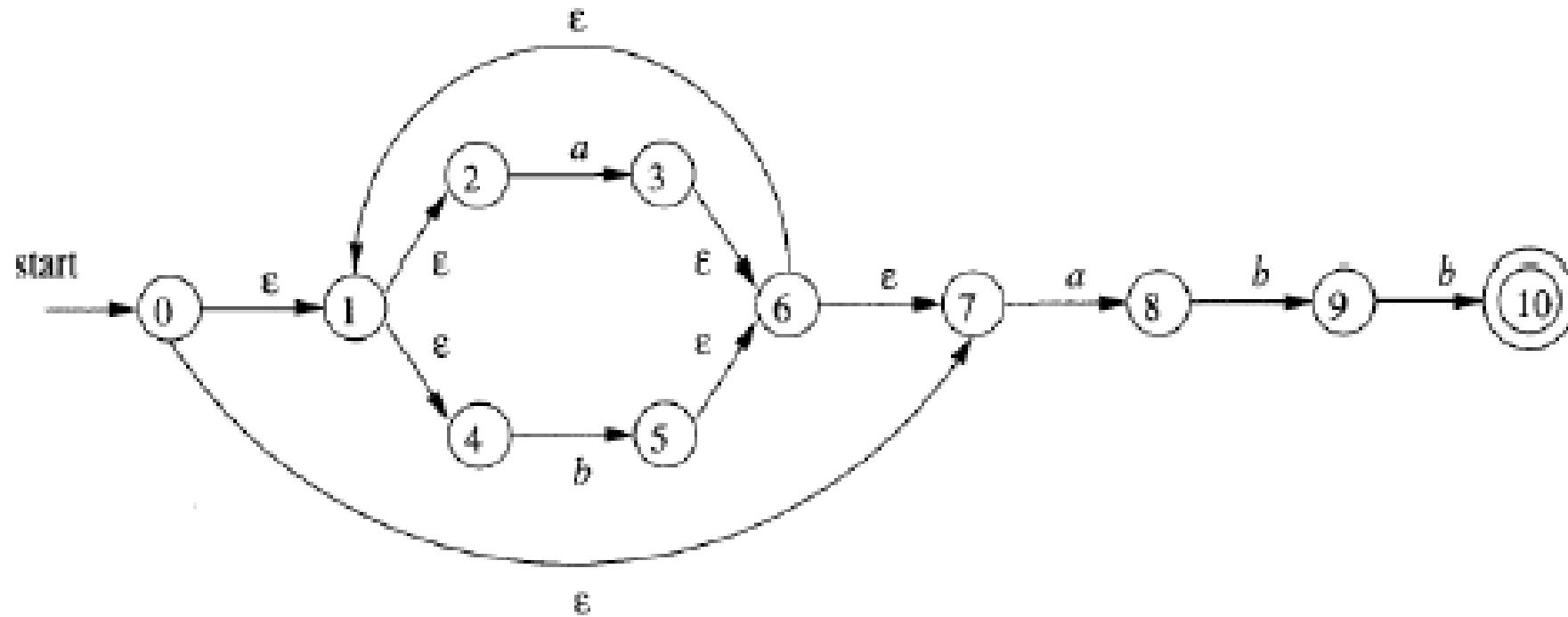


$(a|b)^*$



# Example

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



# Conversion from NFA to DFA

- Reasons to conversion
  - Avoiding ambiguity
- The algorithm idea

Subset construction: The state set of a state in a NFA is thought of as a following STATE of the state in the converted DFA

# Subset Construction algorithm

- Input. An NFA  $N=(S,\Sigma,move,S_0,Z)$
- Output. A DFA  $D= (Q,\Sigma,\delta,I_0,F)$ , accepting the same language
- Requires Pre-processing - Determination of E-Closure

## Pre-process-- $\varepsilon$ -closure( $T$ )

- Obtain  $\varepsilon$ -closure( $T$ )  $T \subseteq S$
- $\varepsilon$ -closure( $T$ ) definition
  - A set of NFA states reachable from NFA state  $s$  in  $T$  on  $\varepsilon$ -transitions alone

# Conversion from NFA to DFA – The pre-process--- $\epsilon$ -closure( $T$ )

- $\epsilon$ -closure( $T$ ) algorithm
  - push all states in  $T$  onto stack;
  - initialize  $\epsilon$ -closure( $T$ ) to  $T$ ;
  - while stack is not empty do {
    - pop the top element of the stack into  $t$ ;
    - for each state  $u$  with an edge from  $t$  to  $u$  labeled  $\epsilon$  do {
      - if  $u$  is not in  $\epsilon$ -closure( $T$ ) {
        - add  $u$  to  $\epsilon$ -closure( $T$ )
        - push  $u$  into stack}}

# Subset Construction Algorithm

- $I_0 = \varepsilon\text{-closure}(S_0), I_0 \in Q$
- For each  $I_i, I_i \in Q$ ,  
    let  $I_t = \varepsilon\text{-closure}(\text{move}(I_i, a))$   
    if  $I_t \notin Q$ , then put  $I_t$  into  $Q$
- Repeat above step until there are no new states to put into  $Q$
- Let  $F = \{I \mid I \in Q, \text{ such that } I \cap Z > \emptyset\}$

# Example

$$A \text{ or } \mathcal{G}_0 \rightarrow \mathcal{E}\text{-closure}(0) = \{0, \underline{1}, \underline{2}, \underline{4}, T\}$$

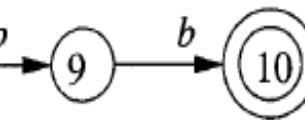
$$\mathcal{E}\text{-closure}\{3, 8\}$$

$$= \{3, 6, T, 1, 2, 4, 8\}$$

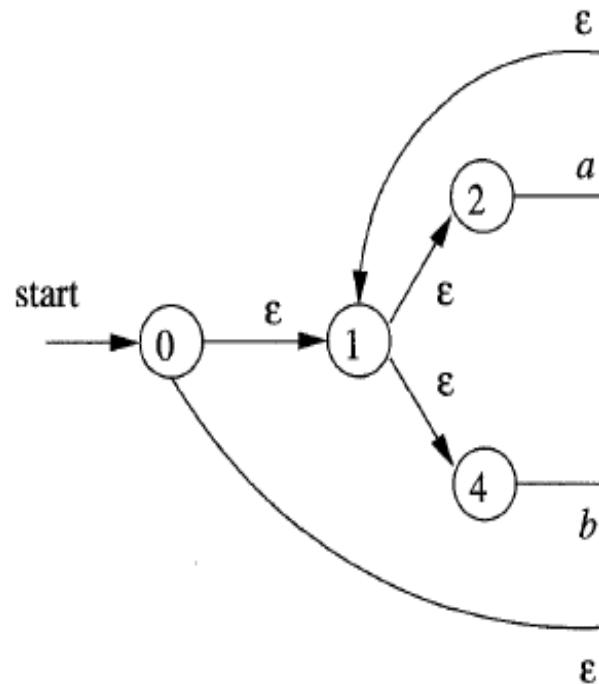
$$B = \{1, 2, 3, 4, 6, T, 8\}$$

$$\delta(B, b) = \mathcal{E}\text{-closure}(5)$$

$$= \{5, 6, T, 1, 2, 4\}$$



$$C = \{1, 2, 4, 5, 6, T\}$$

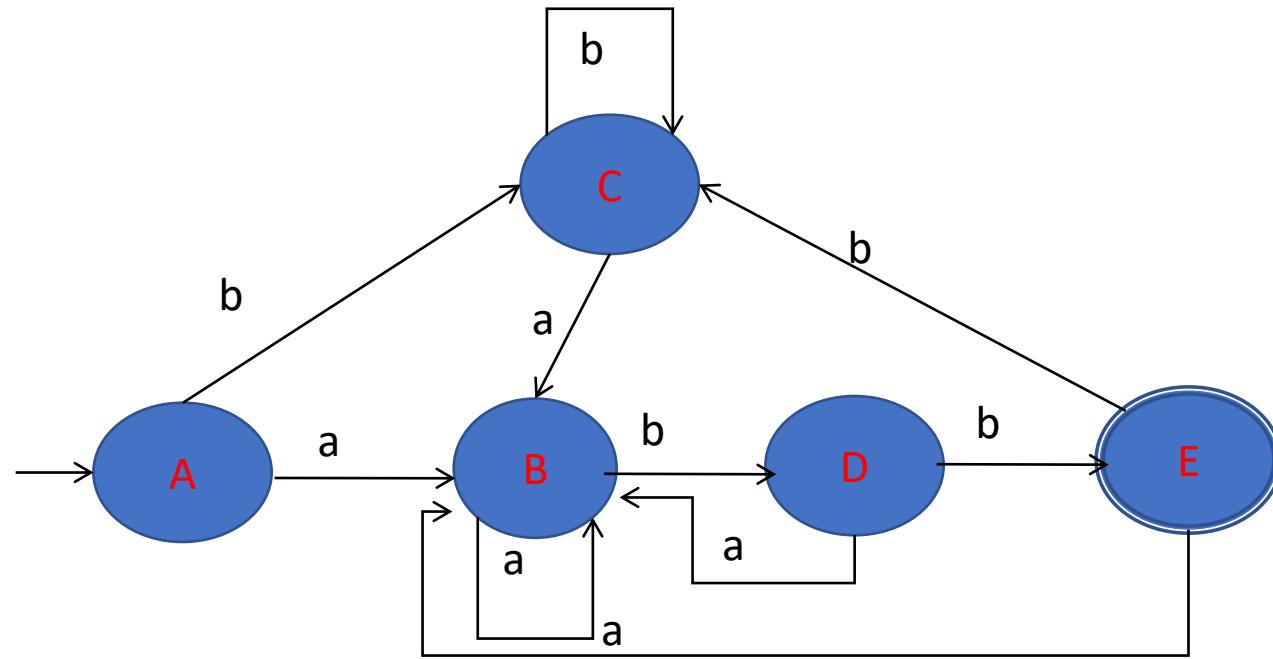


# Result

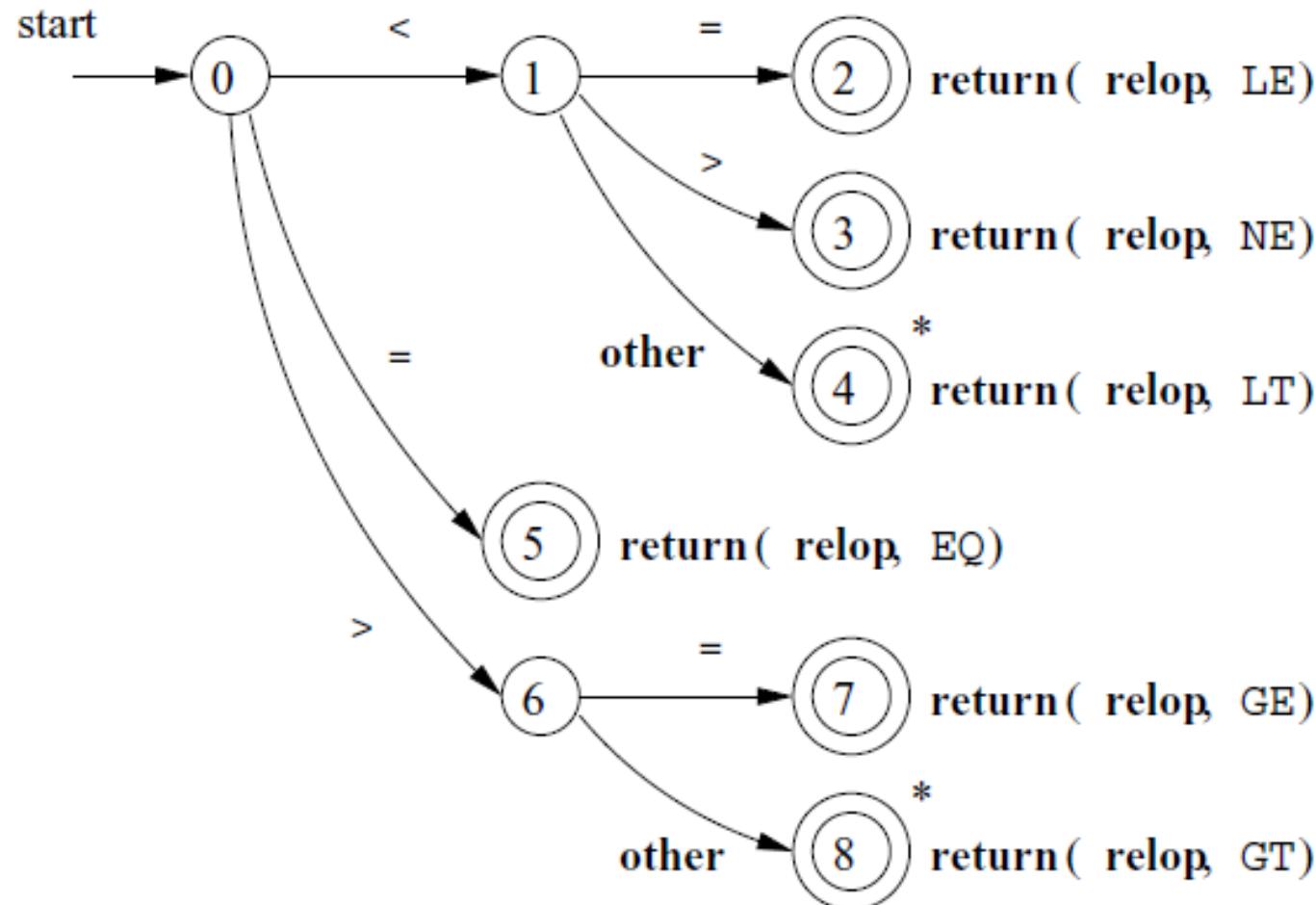
$$\begin{aligned}\delta(A, a) \cdot \\ = \varepsilon\text{-closure}(\text{move}(A, a)) \\ = B\end{aligned}$$

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

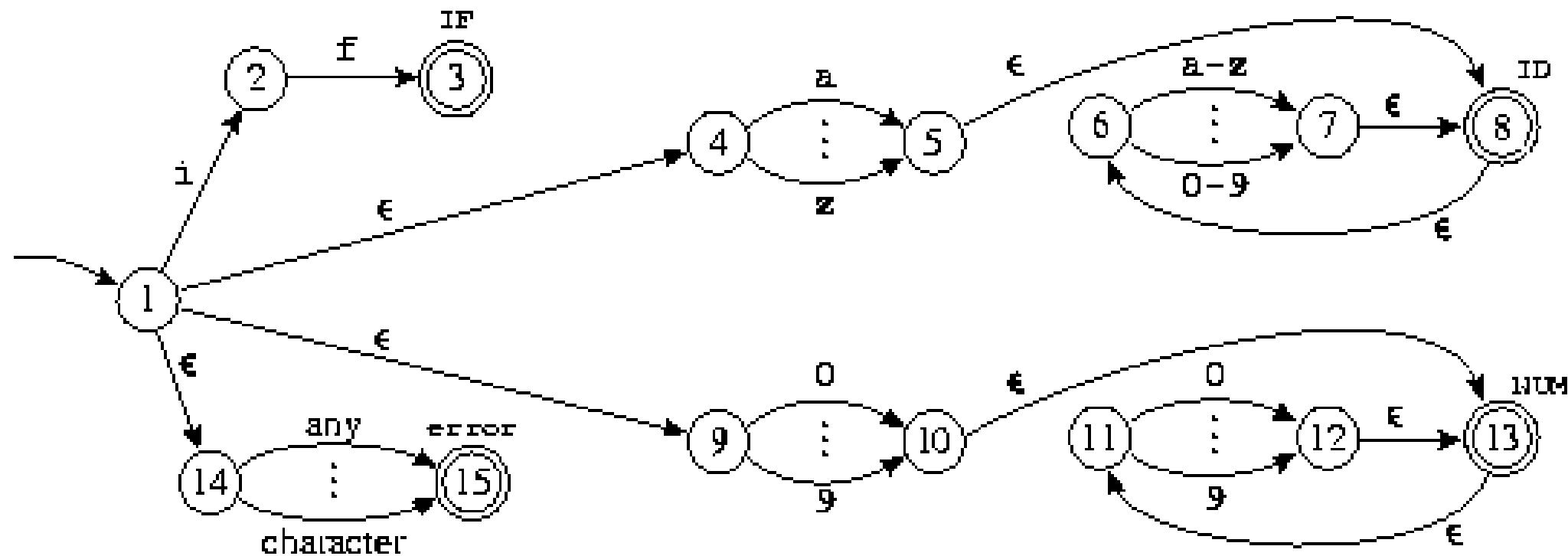
$$\delta(A, b)$$



# Example



# Example



# Subset Construction Algorithm

- RE to E-NFA and then to DFA is time consuming and results in redundant states in the DFA
- Need to minimize the DFA for faster string matching

# Summary till now

- DFA,NFA and NFA with  $\epsilon$  as ways of defining patterns.
- DFA is faster, but construction is difficult
- NFA construction is easier but slower during string matching
- Conversion of RE to E-NFA
- Convert NFA to DFA

# NFA and DFA

- Constructing NFA is easier. But string matching with DFA is faster.
- RE to DFA – done by converting to E-NFA and then to DFA
- This results in an increased number of states in the DFA – need for minimization

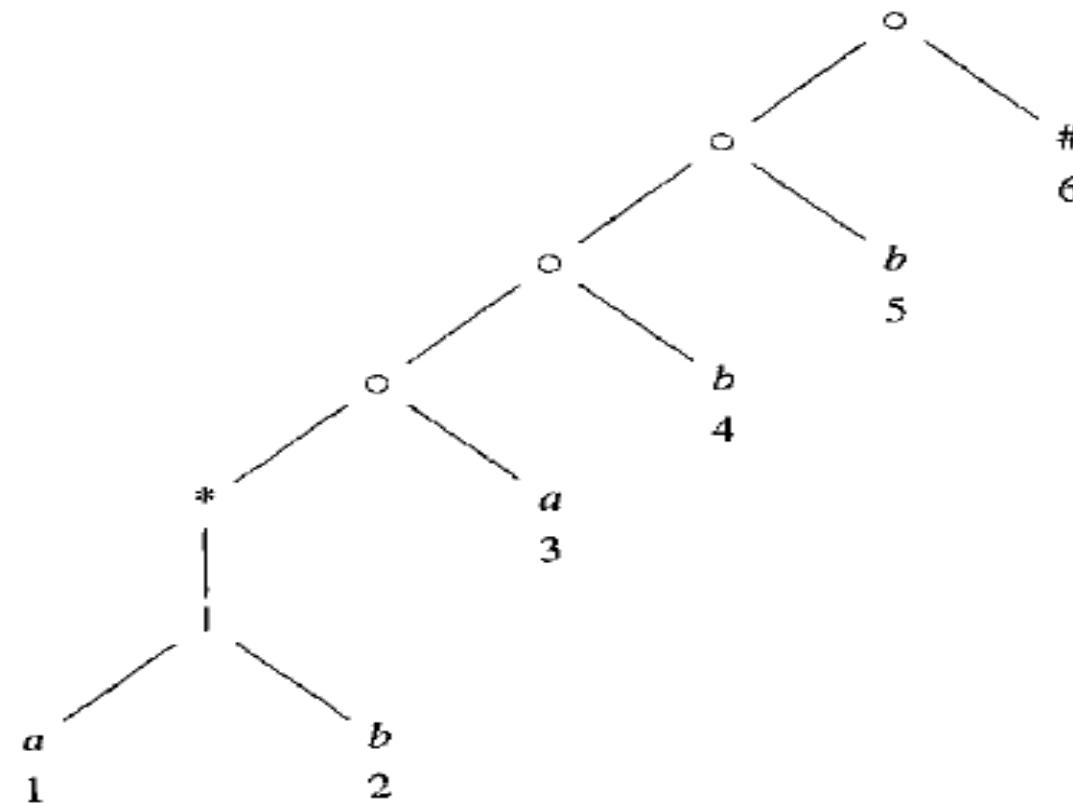
# Minimized DFA

- Construct the DFA directly from RE by using a new algorithm
- Table filling minimization algorithm
  - Construct DFA and then use a procedure to eliminate redundant state

# From Regular Expression to DFA Directly (Algorithm)

- Augment the regular expression  $r$  with a special end symbol  $\#$  to make accepting states important: the new expression is  $r \#$
- Construct a syntax tree for  $r\#$
- Traverse the tree to construct functions *nullable*, *firstpos*, *lastpos*, and *followpos*

# Example Syntax tree for $(a \mid b)^* abb$



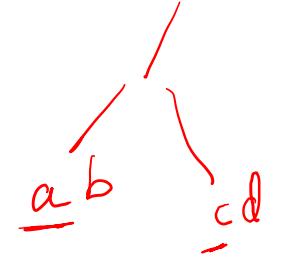
# From Regular Expression to DFA Directly: Annotating the Tree

- $\text{nullable}(n)$ : the subtree at node  $n$  generates languages including the empty string
- $\text{firstpos}(n)$ : set of positions that can match the first symbol of a string generated by the subtree at node  $n$

# Algorithm

- $\text{lastpos}(n)$ : the set of positions that can match the last symbol of a string generated by the subtree at node  $n$
- $\text{followpos}(i)$ : the set of positions that can follow position  $i$  in the tree

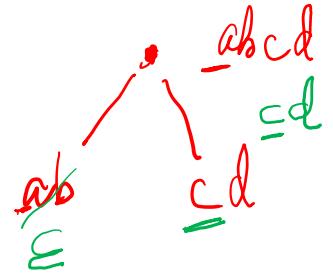
# Annotating tree

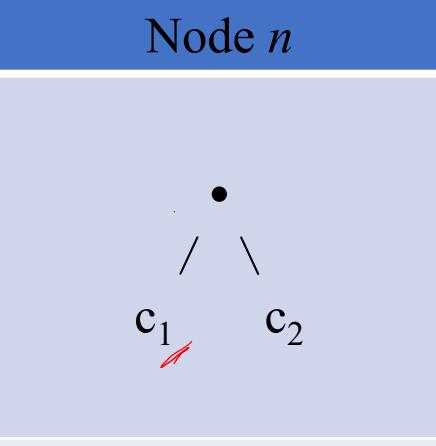


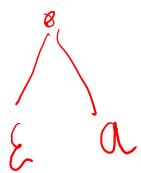
Node $n$	$\text{nullable}(n)$	$\text{firstpos}(n)$	$\text{lastpos}(n)$
Leaf $\epsilon$	true	$\emptyset$	$\emptyset$
Leaf $i$	false	$\{i\}$	$\{i\}$
$\begin{array}{c}   \\ c_1 \quad c_2 \end{array}$	$\text{nullable}(c_1)$ or $\text{nullable}(c_2)$	$\text{firstpos}(c_1) \cup \text{firstpos}(c_2)$	$\text{lastpos}(c_1) \cup \text{lastpos}(c_2)$

$E$     $R$

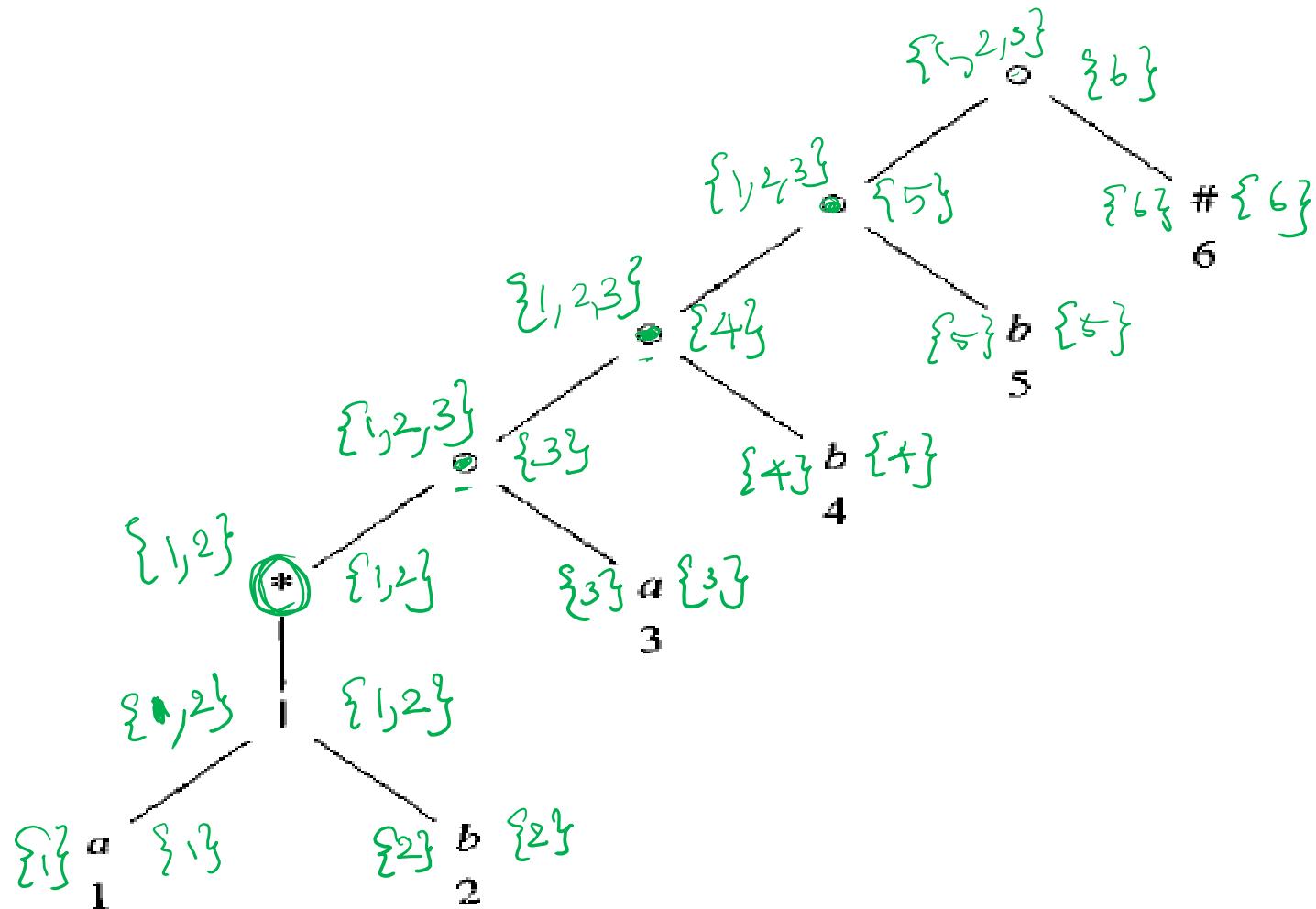
# Annotating tree



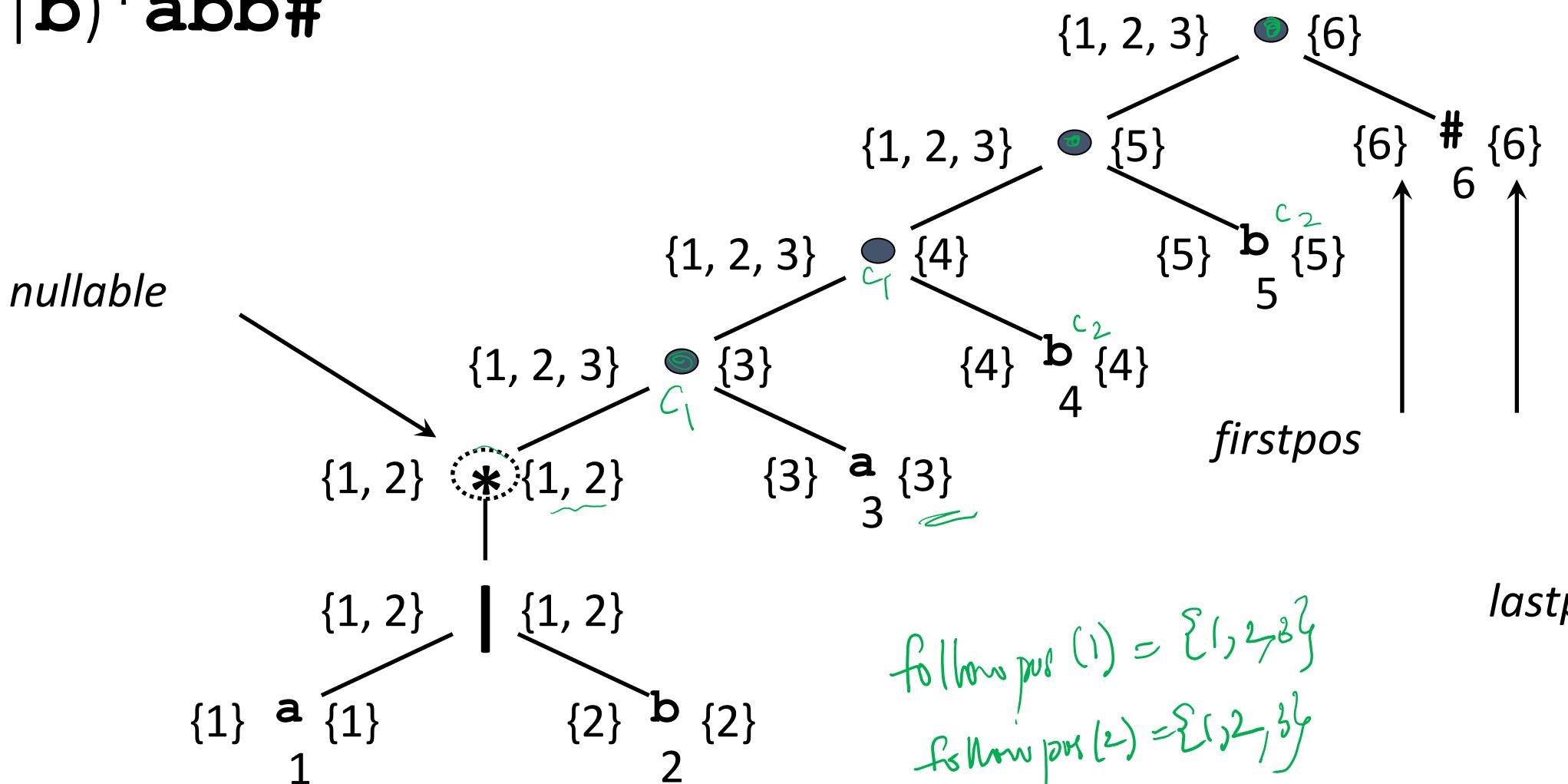
Node $n$	$\text{nullable}(n)$	$\text{firstpos}(n)$	$\text{lastpos}(n)$
 $c_1$ $c_2$	$\text{nullable}(c_1)$ and $\text{nullable}(c_2)$	<b>if</b> $\text{nullable}(c_1)$ <b>then</b> $\text{firstpos}(c_1) \cup$ $\text{firstpos}(c_2)$ <b>else</b> $\text{firstpos}(c_1)$	<b>if</b> $\text{nullable}(c_2)$ <b>then</b> $\text{lastpos}(c_1) \cup$ $\text{lastpos}(c_2)$ <b>else</b> $\text{lastpos}(c_2)$
$*$  $ $ $c_1$	true	$\text{firstpos}(c_1)$	$\text{lastpos}(c_1)$



Syntax tree for  $(a|b)^* abb$



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



# *followpos*

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

# Follow pos

Node	Followpos(n)
1	{1, 2, 3}
2	{1,2,3}
3	{4}
4	{5}
5	{6}
6	$\Phi$

# Algorithm

$s_0 := \text{firstpos}(\text{root})$  where  $\text{root}$  is the root of the syntax tree

$D\text{states} := \{s_0\}$  and is unmarked

**while** there is an unmarked state  $T$  in  $D\text{states}$  **do**

    mark  $T$

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

            let  $U$  be the set of positions that are in  $\text{followpos}(p)$

                for some position  $p$  in  $T$ ,

                    such that the symbol at position  $p$  is  $a$

**if**  $U$  is not empty and not in  $D\text{states}$  **then**

                    add  $U$  as an unmarked state to  $D\text{states}$

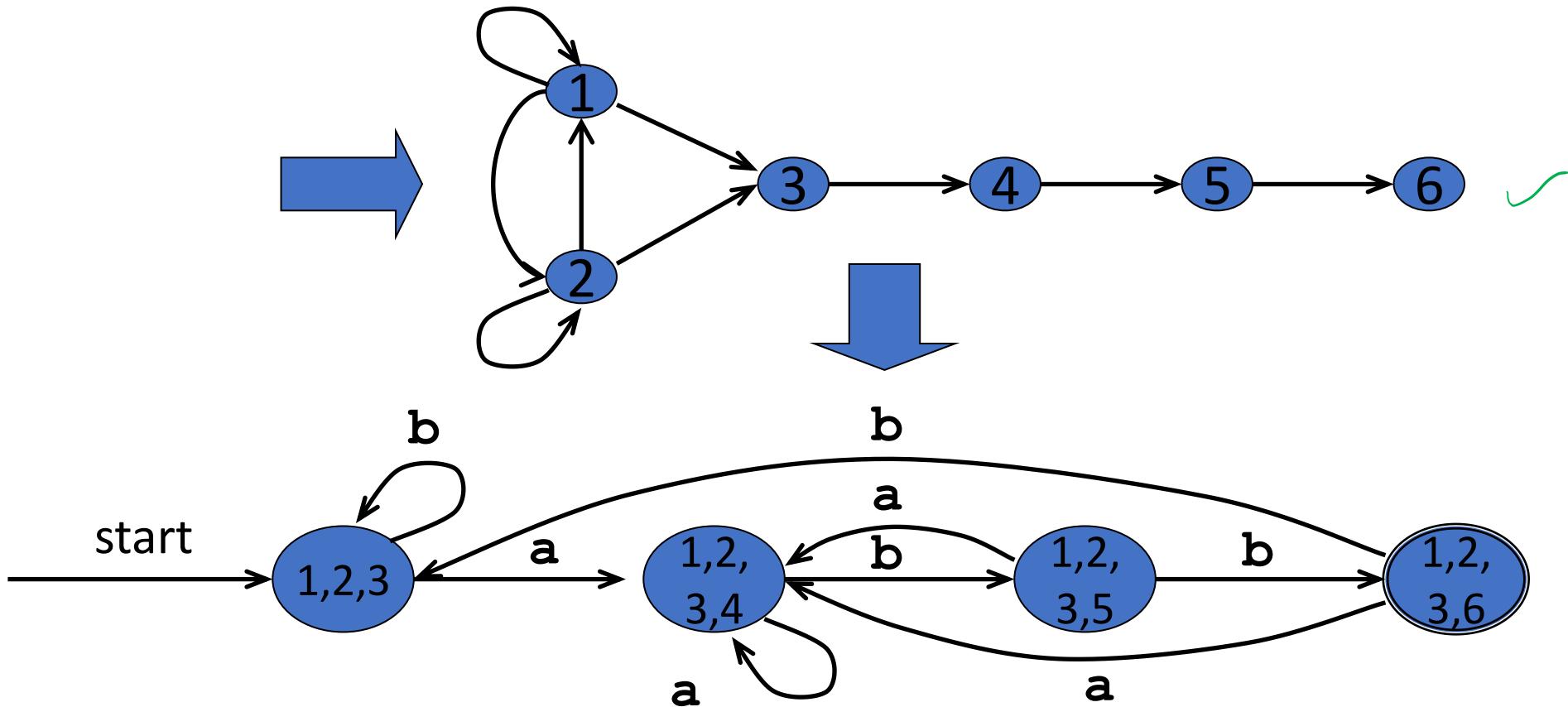
**end if**

$D\text{tran}[T,a] := U$

**end do**

**end do**

# From Regular Expression to DFA Directly: Example



# Minimized DFA - Table filling minimization algorithm

- Table filling minimization algorithm
  - Construct DFA and then use a procedure to eliminate redundant state
- Construct the DFA directly from RE by using a new algorithm

# Basic Idea

- Find all groups of states that can be distinguished by some input string.
- At beginning of the process, we assume two distinguished groups of states:
  - the group of non-accepting states
  - the group of accepting states..
- Then we use the method of partition of equivalent class on input string to partition the existed groups into smaller groups

# Minimization Algorithm

- Input: A DFA  $M = \{S, \Sigma, \text{move}, s_0, F\}$
- Output: A DFA  $M'$  accepting the same language as  $M$  and having as few states as possible.

# Minimization Algorithm

1. Construct an initial partition  $\Pi$  of the set of states with two groups: the accepting states  $F$  and the non-accepting states  $S-F$ .  $\Pi_0 = \{I_0^1, I_0^2\}$
2. For each group  $I$  of  $\Pi_i$ , partition  $I$  into subgroups such that two states  $s$  and  $t$  of  $I$  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  $\Pi_i$ ; replace  $I$  in  $\Pi_{i+1}$  by the set of subgroups formed.
3. If  $\Pi_{i+1} = \Pi_i$ , let  $\Pi_{final} = \Pi_{i+1}$  and continue with step (4). Otherwise, repeat step (2) with  $\Pi_{i+1}$

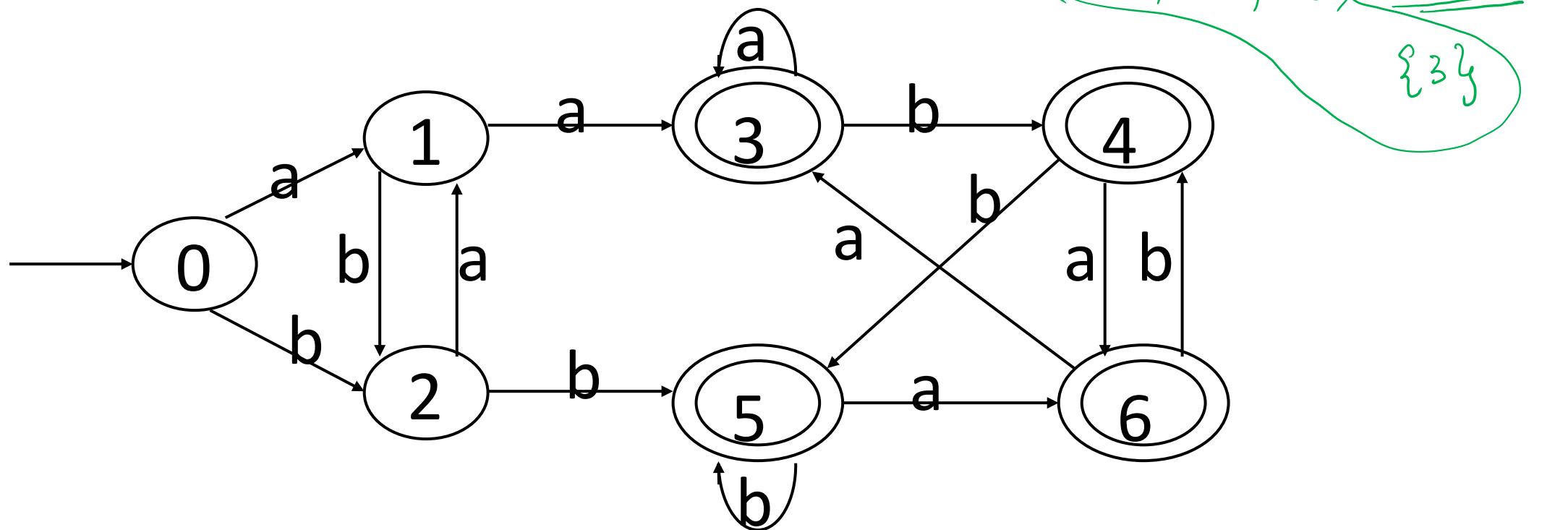
# Minimization Algorithm

- Choose one state in each group of the partition  $\Pi_{final}$  as the representative for that group which will be the states of the reduced DFA  $M'$ .
- Let  $s$  and  $t$  be representative states for  $s$ 's and  $t$ 's group respectively, and suppose on input  $a$  there is a transition of  $M$  from  $s$  to  $t$ . Then  $M'$  has a transition from  $s$  to  $t$  on  $a$ .

# Minimization Algorithm

- If  $M'$  has a dead state(a state that is not accepting and that has transitions to itself on all input symbols),then remove it. Also remove any states not reachable from the start state.

# Example



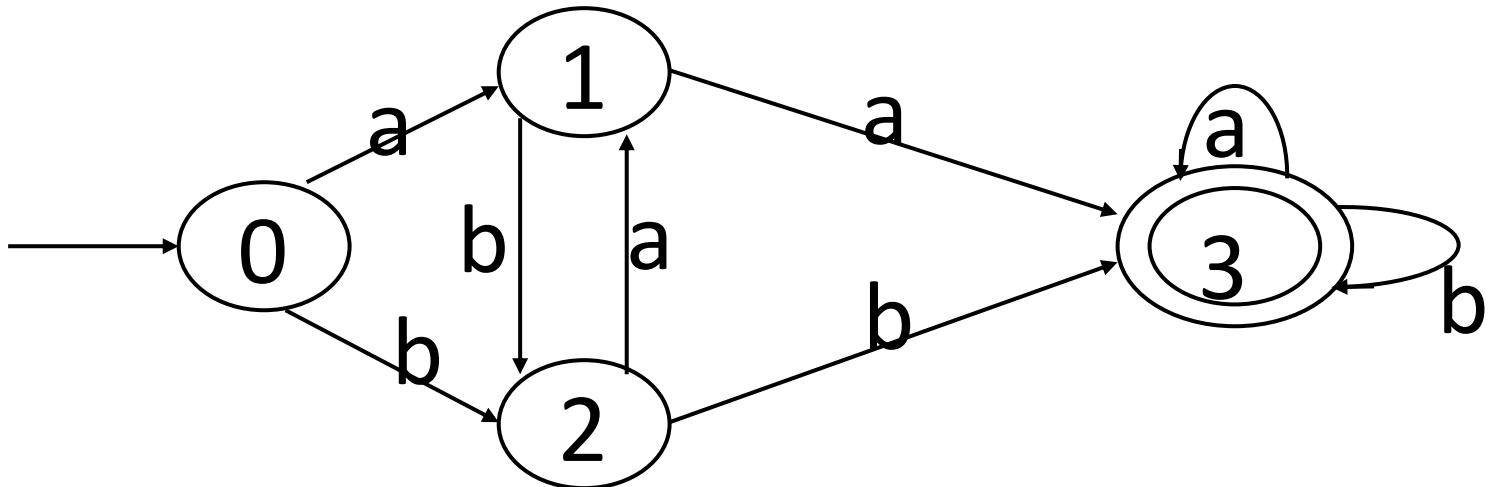
# Example

- Initialization:  $\Pi_0 = \{\{0,1,2\}, \{3,4,5,6\}\}$
- For Non-accepting states in  $\Pi_0$  :
  - a:  $\text{move}(\{0,2\}, a) = \{1\}$ ;  $\text{move}(\{1\}, a) = \{3\}$ . 1,3 do not in the same subgroup of  $\Pi_0$ .
  - So,  $\Pi_1` = \{\{1\}, \{0,2\}, \{3,4,5,6\}\}$
  - b:  $\text{move}(\{0\}, b) = \{2\}$ ;  $\text{move}(\{2\}, b) = \{5\}$ . 2,5 do not in the same subgroup of  $\Pi_1`$ .
  - So,  $\Pi_1`` = \{\{1\}, \{0\}, \{2\}, \{3,4,5,6\}\}$

# Example

- For accepting states in  $\Pi_0$  :
  - a:  $\text{move}(\{3,4,5,6\}, a) = \{3,6\}$ , which is the subset of  $\{3,4,5,6\}$  in  $\Pi_1$ “
  - b:  $\text{move}(\{3,4,5,6\}, b) = \{4,5\}$ , which is the subset of  $\{3,4,5,6\}$  in  $\Pi_1$ “
  - So,  $\Pi_1 = \{\{1\}, \{0\}, \{2\}, \{3,4,5,6\}\}$ .
- Apply the same step again to  $\Pi_1$  ,and get  $\Pi_2$ .
  - $\Pi_2 = \{\{1\}, \{0\}, \{2\}, \{3,4,5,6\}\} = \Pi_1$ ,
  - So,  $\Pi_{\text{final}} = \Pi_1$
- Let state 3 represent the state group  $\{3,4,5,6\}$

# Minimized DFA



# Lexical Analysis

# Lexical Phase

- Scanning
  - Deletion of comments, and compaction of consecutive white space characters into one
- Lexical Analysis
  - Complex portion, to produce tokens from the output of the scanner

# Lexical Analysis

- Input
  - program text (file)
- Output
  - sequence of tokens
- Read input file
- Identify language keywords and standard identifiers

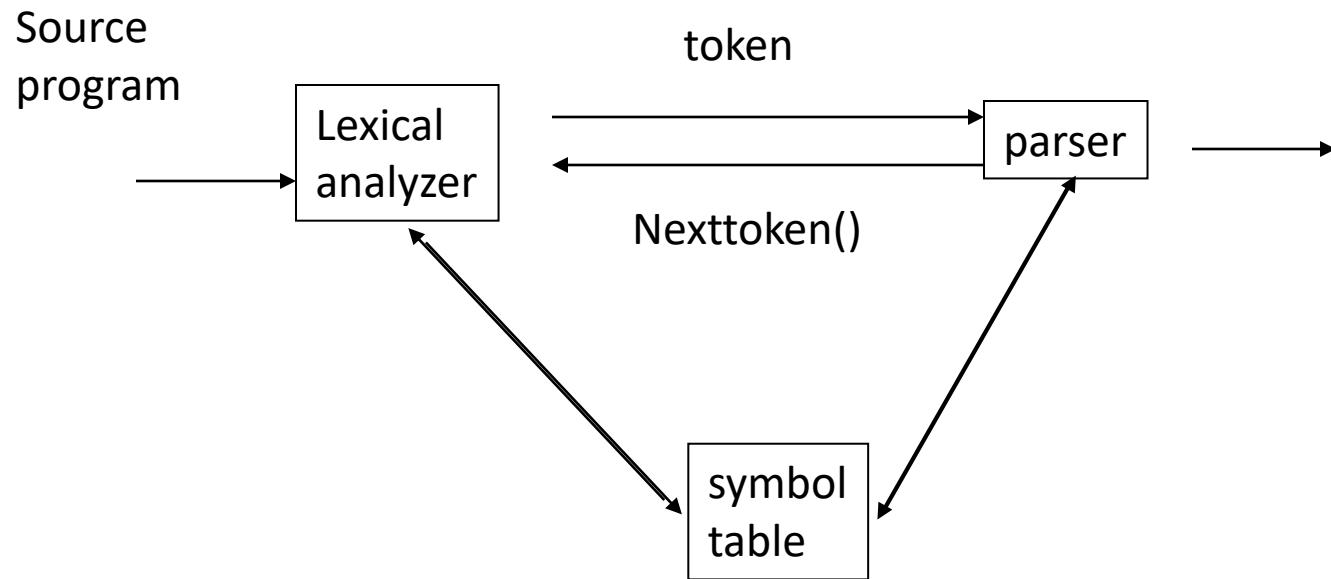
# Lexical Analysis

- Handle include files and macros
- Count line numbers
- Remove whitespaces
- Report illegal symbols
- Create symbol table

# Lexical Analyzer

- Lexical analyzer does not have to be an individual phase.
- But having a separate phase simplifies the design and improves the efficiency and portability.

# Interaction of Lexical analyzer with parser



# Why Lexical Analysis

- Simplifies the syntax analysis
  - And language definition
- Modularity / Portability
- Reusability
- Efficiency

# Definitions

- Lexeme is a particular instant of a token.
- Token: a group of characters having a collective meaning.
  - token: identifier, lexeme: area, rate etc.
- Pattern: the rule describing how a token can be formed.
  - identifier:  $([a-z] | [A-Z]) ([a-z] | [A-Z] | [0-9])^*$

# Issues in lexical Analyzer

- How to identify tokens?
  - Patterns as RE, NFA, DFA
- How to recognize the tokens giving a token specification (how to implement the nexttoken() routine)?
  - Integrate the first two phases of the compiler

# The Lexical Analysis Problem

- Given
  - A set of token descriptions
    - Token name
    - Regular expression defining the pattern for a lexeme
  - An input string
- Partition the strings into tokens  
(class, value)

# Lexical Analysis problem

- Ambiguity resolution
  - The longest matching token
  - Between two equal length tokens select the first

$\underline{a} \textcolor{green}{<} \textcolor{red}{b}$        $\Leftarrow$   
 $\underline{a} \textcolor{green}{|} \textcolor{red}{<} = b$   
 $\uparrow \uparrow \uparrow \uparrow \uparrow$

$\textcolor{green}{<}$        $\textcolor{red}{\text{<} \text{=}}$

$= =$        $>$        $> =$

# Example of Token

TOKEN	Description	Sample lexeme
if	Character i, f	If
else	Characters e, l, s, e	else
Comparison	< or > or < = or >= or == or !=	<=
id	Letter followed by letters and digits	Pi, score, a123
Number	Any numeric constant	3.14, 9.08
Literal	Anything within “ ”	“Seg fault”

# Classes covering most of the tokens

- One token for each keyword. The pattern for a keyword is the same as the keyword itself.
- Tokens for the operators, either individually or in classes such as the token comparison
- One token representing all identifiers.
- One or more tokens representing constants, such as numbers and literal strings.
- Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

# Attributes for Tokens

- A pointer to the symbol-table entry in which the information about the token is kept

E.g E=M\*C\*\*2

<**id**, pointer to symbol-table entry for E>

<**assign\_op**>

<**id**, pointer to symbol-table entry for M>

<**mult\_op**>

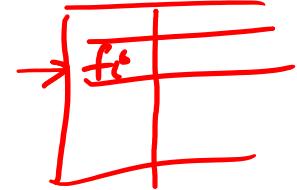
<**id**, pointer to symbol-table entry for C>

<**exp\_op**>

<**num**, integer value 2>

# Lexical Errors

⇒ A<sub>i</sub>



- It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error

else<sub>2</sub>

- Ex: fi (a == f(x)) . . .      if (condition)

- simplest recovery strategy is “panic mode” recovery

else<sub>1</sub> ✓

- Other possible error-recovery actions are:

else

else

else<sub>1</sub>

- Delete one character from the remaining input.
- Insert a missing character into the remaining input.
- Replace a character by another character.
- Transpose two adjacent characters

fi  
if

abc  
abd  
c

# Strings and Languages

- An alphabet is any finite set of symbols
  - Typical examples of symbols are letters, digits, and punctuation
- A string over an alphabet is a finite sequence of symbols drawn from that alphabet
- The length of a string  $s$ ,  $|s|$ , is the number of occurrences of symbols in  $s$
- The empty string, denoted  $\epsilon$ , is the string of length zero
- A language is any countable set of strings over some fixed alphabet

# Regular Expressions

Basic patterns	Matching
x	The character x
.	Any character except newline
[xyz]	Any of the characters x, y, z
R?	An optional R

$\epsilon|R$

# Regular expression

$R^*$	Zero or more occurrences of R
$R^+$	One or more occurrences of R
$R_1R_2$	$R_1$ followed by $R_2$
$R_1 R_2$	Either $R_1$ or $R_2$
(R)	R itself

$$R^* = \bigcup_{i=0}^{\infty} R^i$$

$$\{ R^0, R^1, R^2, R^3, \dots, R^\infty \}$$

$$R = ab$$

$$R = a/b$$

$$R^* = \{ \epsilon, ab, abab, ababab, \dots \}$$

$$R^* = \{ \epsilon, a, b, \underbrace{aa, ab, ba, bb}, \dots \}$$

$$R^+ = \bigcup_{i=1}^{\infty} R^i$$

$$\{ R^1, R^2, R^3, \dots, R^\infty \}$$

# Properties of Regular Expression

- $L(r) \cup L(s)$  is also a RE
- $L(r) \cap L(s)$  is also RE
- $R^*$  is also RE if R is one
- If  $\Sigma = \{a, b\}$ , then
- $L_1 = a^* = \{\epsilon, a, aa, aaa, \dots\}$
- $L_2 = a \mid b = \{a, b\}$

# Regular Expression

- Pascal language identifiers

$L(r) = \text{letter} (\text{letter} \mid \text{digit})^*$

Language for defining C language identifiers

- \* has the highest precedence, followed by concatenation followed by |
- $\epsilon$  is a regular expression which is a string of length 0

# Regular Definitions

- Names given to certain regular expressions and use these names later
- Regular definition is a sequence of the form  
 $d_1 \rightarrow r_1, d_2 \rightarrow r_2, d_3 \rightarrow r_3\dots$
- Each  $d_i$  is a symbol not in the input alphabet
- Each  $r_i$  is a regular expression

$$\underline{d_1} \rightarrow r_1$$

$$d_2 \rightarrow r_2$$

$$d_3 \rightarrow r_3$$

$$\Sigma$$

$$\Sigma \cup d_1$$

$$\Sigma \cup d_1 \cup d_2$$

# Regular Definitions

- letter  $\rightarrow A \mid B \mid \dots \mid z \mid -$
- digit  $\rightarrow 0 \mid 1 \mid 2 \mid 3 \dots \mid 9$
- id  $\rightarrow \text{letter} (\text{letter} \mid \text{digit})^*$

id  $\rightarrow ([a-z] \mid [A-Z] \mid -) ([a-z][A-Z] \mid - \mid [0-9])^*$

# Example

1234  
1

10.23 ✓  
10E2

- digit → 0 | 1 | ... | 9
- digits → digit digit\*
- optionalFraction → .digits | ε
- optionalExponent → (E(+ | - | ε)digits) | ε
- number → digits optionalFraction optionalExponent

$$z = y + \underline{\underline{3.29}}$$

# Token Recognition

if (expr) stmt      if (expr) stmt else stmt

if ( )  
{ } ==

• Stmt  $\rightarrow$  if expr then Stmt | if expr then Stmt else Stmt |  $\epsilon$  ✓

if ( )  
{ } ==

• expr  $\rightarrow$  term relop term | term

{ } else

• term  $\rightarrow$  id | number

if ( a)

• id  $\rightarrow$  letter (letter|digits)\*

{ } ==

• relop  $\rightarrow$  < | > | <= | >= | == | !=

if (a > b)

• number  $\rightarrow$  digits

{ } ==

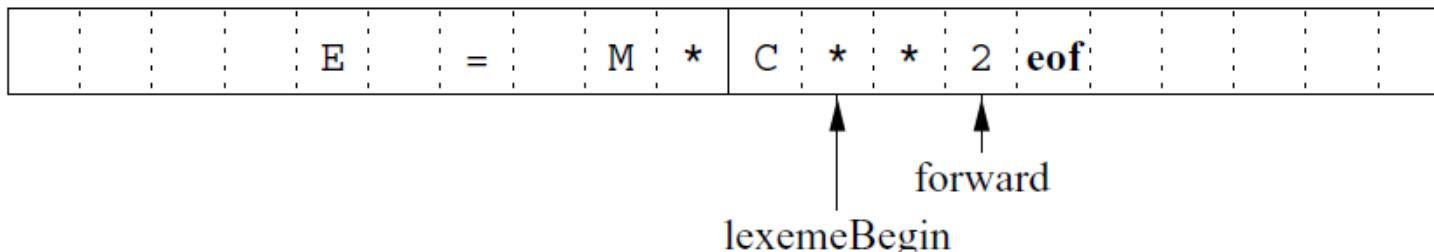
s  $\rightarrow$  s ; s | s

# Input Buffering

- Have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme
  - Ex: can't determine the end of an identifier until we see a character that is not a letter or digit
  - Ex: In C, single-character operators like -, =, or < could also be the beginning of a two-character operator like ->, ==, or <=

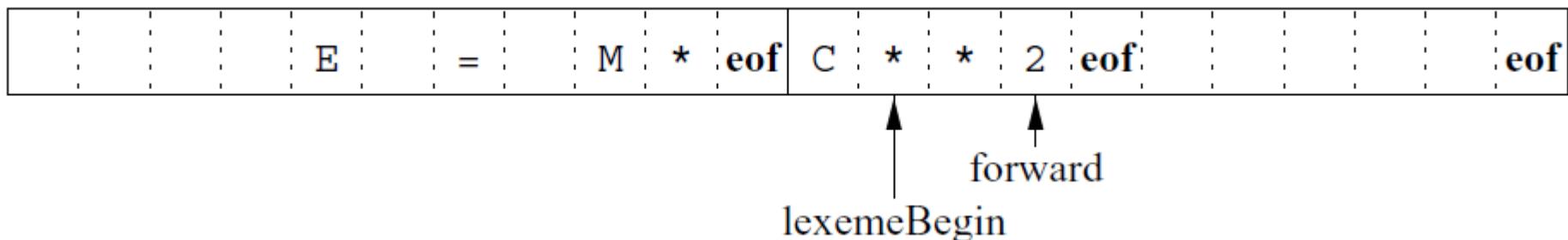
# Buffer Pairs

- two buffers that are alternately reloaded
- Each buffer is of the same size N, and N is usually the size of a disk block
- Using one system read command we can read N characters into a buffer, rather than using one system call per character
- If fewer than N characters remain in the input file, then a special character, represented by eof, marks the end of the source file
- Two pointers to the input are maintained:
  - *lexemeBegin*, marks the beginning of the current lexeme, whose extent we are attempting to determine
  - *forward* scans ahead until a pattern match is found



# Sentinels

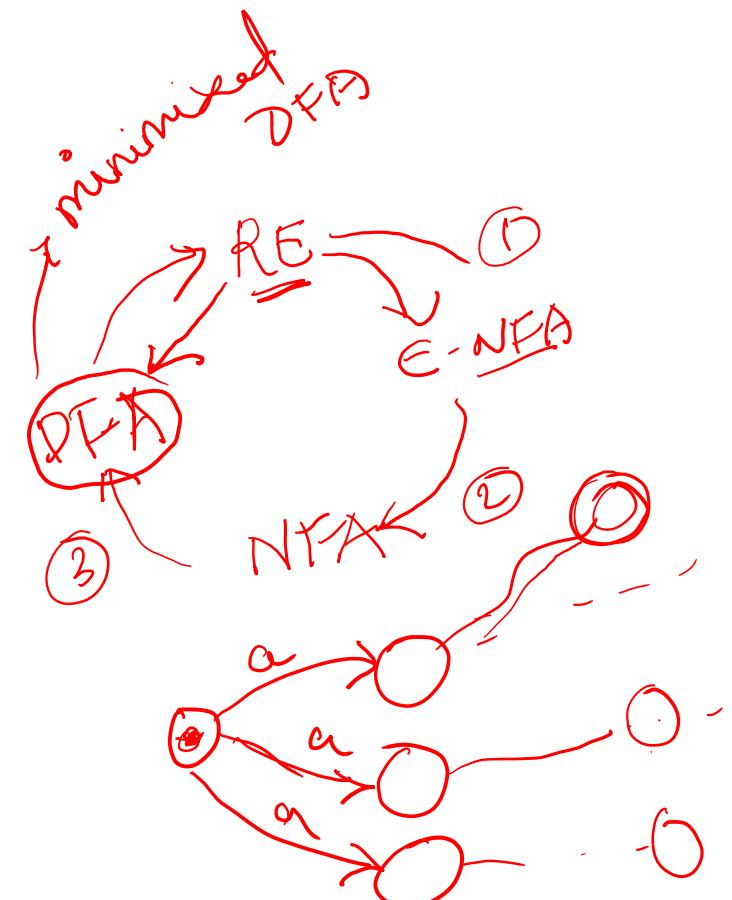
- Before advancing ***forward***, test whether end of one of the buffers is reached, if so, reload the other buffer from the input, and move ***forward*** to the beginning of the newly loaded buffer
- For each character read, we make two tests: one for the end of the buffer, and one to determine what character is read
- 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
- Any eof that appears other than at the end of a buffer means end of input



```
switch ( *forward++ ) {
    case EOF:
        if (forward is at end of first buffer) {
            reload second buffer;
            forward = beginning of second buffer;
        }
        else if (forward is at end of second buffer) {
            reload first buffer;
            forward = beginning of first buffer;
        }
        else /* EOF within a buffer marks the end of input */
            terminate lexical analysis;
        break;
    Cases for the other characters
}
```

# Why Automata?

- It may be hard to specify regular expressions for certain constructs
  - Examples
    - Strings
    - Comments
- Writing automata may be easier
- Can combine both



# Why Automata?

- Specify partial automata with regular expressions on the edges
  - No need to specify all states
  - Different actions at different states

# Constructing Automaton from Specification

- Create a non-deterministic automaton (NDFA) from every regular expression
- Merge all the automata using epsilon moves (like the  $\mid$  construction)
- Construct a deterministic finite automaton (DFA)
  - State priority
- Minimize the automaton starting with separate accepting states

# Finite Automata

- By default a Deterministic one.
- Five tuple representation  
 $(Q, \Sigma, \delta, q_0, F)$ ,  $q_0$  belongs to  $Q$  and  $F$  is a subset of  $Q$   
 $\delta$  is a mapping from  $Q \times \Sigma$  to  $Q$
- Every string has exactly one path and hence faster string matching

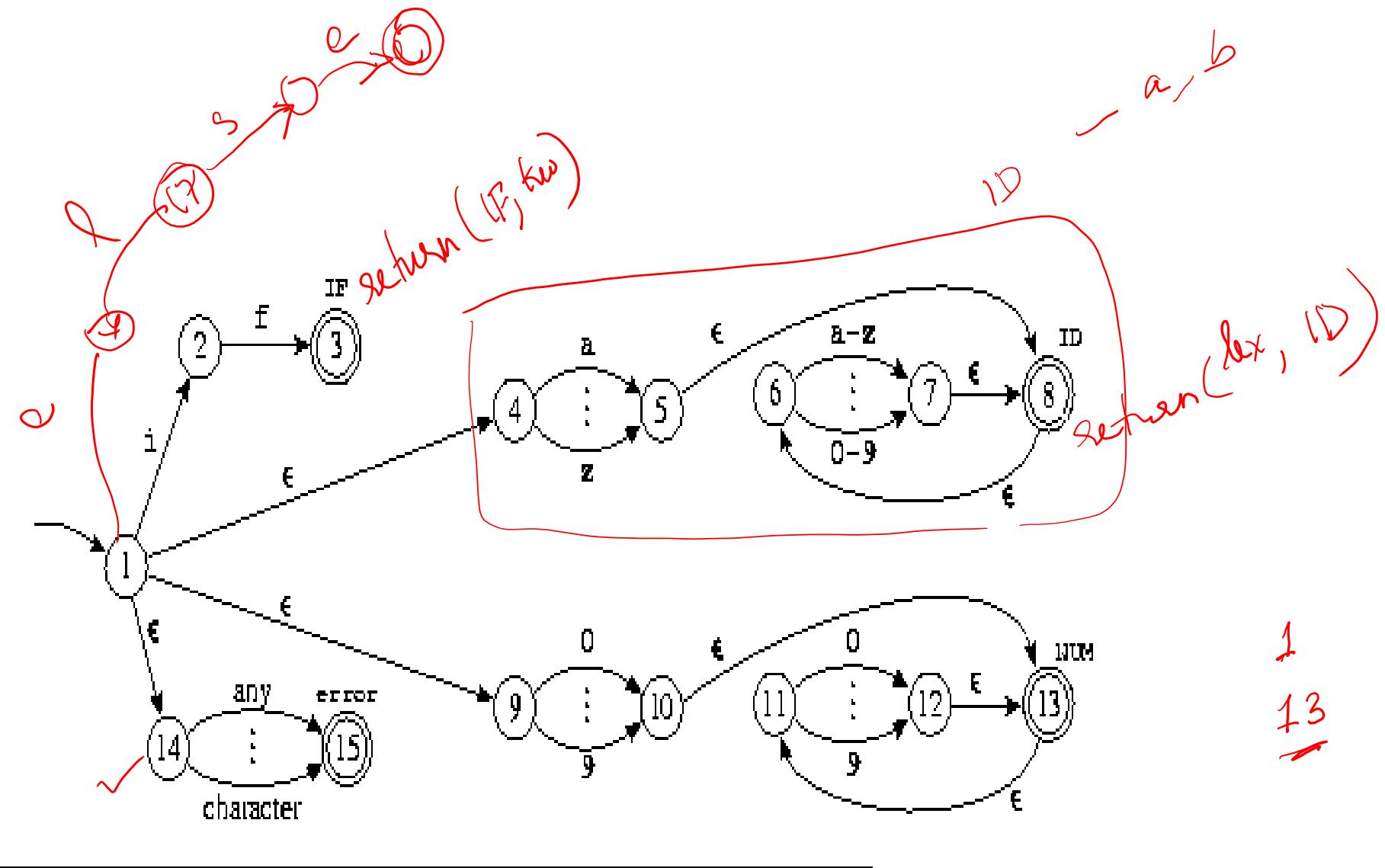
# Non-deterministic Finite automata

- Same as deterministic, gives some flexibility.
- Five tuple representation  
 $(Q, \Sigma, \delta, q_0, F)$ ,  $q_0$  belongs to  $Q$  and  $F$  is a subset of  $Q$   
 $\delta$  is a mapping from  $Q \times \Sigma$  to  $2^Q$
- More time for string matching as multiple paths exist.

# Non-Deterministic Finite automata with $\epsilon$

- Same as NFA. Still more flexible in allowing to change state without consuming any input symbol.
- $\delta$  is a mapping from  $Q \times \Sigma \cup \underline{\{ \epsilon \}}$  to  $2^Q$
- Slower than NFA for string matching

# Example



# Summary

- The work involved in lexical phase
- Constructing Regular expression
- Introduction to DFA, NFA and NFA with  $\epsilon$

# CALR Parsing

# Conflict in SLR parsers

- Shift / reduce conflict arises
- Follow information alone is not sufficient to decide when to reduce.
- Hence, powerful parser is required

# Conflicts in SLR parsers

- In SLR, if there is a production of the form  $A \rightarrow \alpha\cdot$ , then a reduce action takes place based on  $\text{follow}(A)$
- There would be situations, where when state  $i$  appears on the TOS, the viable prefix  $\beta\alpha$  on the stack is such that  $\beta A$  cannot be followed by terminal 'a' in a right sentential form
- Hence, the reduction  $A \rightarrow \alpha$  would be invalid on input 'a'

# CALR parsers motivation

- If it is possible to do more in the states that allow us to rule out some of the invalid reduction, introduce more states
- Introduce exactly which input symbols to follow a particular non-terminal

# CALR parsers

- Construct LR(1) items
- Use these items to construct the CALR parsing table involving action and goto
- Use this table, along with input string and stack to parse the string

# CALR motivation

- Extra symbol is incorporated in the items to include a terminal symbol as a second component
- $A \rightarrow [\alpha .\beta, a]$  where  $A \rightarrow \alpha\beta$  is a production and ‘a’ is a terminal or the right end marker \$ - LR(1) item

# LR(1) item

- 1 – refers to the length of the second component – lookahead of the item
- Lookahead has no effect in  $A \rightarrow [\alpha .\beta , a]$  where  $\beta$  is not  $\epsilon$ , but  $A \rightarrow [\alpha . , a]$  calls for a reduction  $A \rightarrow \alpha$  if the next input symbol is ‘a’, ‘a’ will be subset of  $\text{follow}(A)$

# LR(1) item

- $A \rightarrow [\alpha .\beta, a]$  is a valid item for a viable prefix  $\gamma$  if there is a derivation  $S \Rightarrow \delta A w \Rightarrow \delta \alpha \beta w$  where  $\gamma = \delta \alpha$  and either 'a' is the first symbol of 'w' or 'w' is  $\epsilon$  and 'a' is  $\$$

# LR(1) item algorithm

- Closure ( $I$ )
    - {repeat for each item  $[A \rightarrow \alpha \cdot B\beta, a]$  in  $I$ ,  
for each production  $B \rightarrow \gamma$  in  $G'$  and  
each terminal  $b$  in  $\text{First}(\beta a)$  such that  $[B \rightarrow \cdot\gamma, b]$  is not in  $I$  do  
add  $[B \rightarrow \cdot\gamma, b]$  to set  $I$   
until no more items can be added to  $I$
- end }

# Goto(I, X)

Begin

  Initialize J to be the empty set

  For each item  $[A \rightarrow \alpha.X\beta, a]$  in I such that

    add item  $[A \rightarrow \alpha X. \beta, a]$  to set J;

  Return closure(J)

end

# Items( $G'$ )

```
Begin C:= closure ( { $S' \rightarrow .S, \$$ } );
repeat
    for each set of items I in C
        for each grammar symbol X
            if goto(I,X) is not empty and not in C
                add goto(I, X) to C;
    until no more set of items can be added to C
end
```

# Example

- $S \rightarrow CC$
- $C \rightarrow cC$
- $C \rightarrow d$
- Augmented
- $S' \rightarrow S$
- $S \rightarrow CC$
- $C \rightarrow cC$
- $C \rightarrow d$

# LR(1) items

- $I_0$  :

$S' \rightarrow .S, \$$

$S \rightarrow .CC, \$$

$C \rightarrow .cC, c/d$  (first(C\$))

$C \rightarrow .d, c/d$

- $I_1 : \text{goto}(I_0, S)$

$S' \rightarrow S., \$$

- $I_2 : \text{goto}(I_0, C)$

$S \rightarrow C.C, \$$

$C \rightarrow .cC, \$$

$C \rightarrow .d, \$$

- $I_3 : \text{goto}(I_0, c), \text{goto}(I_3, c),$   
 $C \rightarrow c.C, c/d$   
 $C \rightarrow .cC, c/d$   
 $C \rightarrow .d, c/d$
- $I_4 : \text{goto}(I_0, d) \text{ goto}(I_3, d)$   
 $C \rightarrow d., c/d$
- $I_5 : \text{goto}(I_2, C)$   
 $S \rightarrow CC., \$$
- $I_6 : \text{goto}(I_2, c) \text{ goto}(I_6, c)$   
 $C \rightarrow c.C, \$$   
 $C \rightarrow .cC, \$$   
 $C \rightarrow .d, \$$

- $I_7 : goto(I_2, d) \quad goto(I_6, d)$

$C \rightarrow d., \$$

- $I_8 : goto(I_3, C)$

$C \rightarrow cC., c/d$

- $I_9 : goto(I_6, C)$

$C \rightarrow cC., \$$

# Parsing Table

- Construct  $C = \{I_0, I_1, I_2 \dots I_n\}$  the collection of LR(1) items for  $G'$
- State  $I$  of the parser is from  $I_i$ 
  - if  $[A \rightarrow \alpha.a\beta, b]$  is in  $I_i$  and  $\text{goto}(I_i, a) = I_j$  set action  $[i, a] = \text{shift } j$ , where  $a$  is a terminal
  - if  $[A \rightarrow \alpha . , a]$  is in  $I_i$  and  $A \neq S'$ , then set action  $[i, a] = \text{reduce by } A \rightarrow \alpha$   
// a conflict here implies the grammar is not CALR grammar
- If  $\text{goto}(I_i, A) = I_j$  then  $\text{goto}(i, A) = j$
- $[S' \rightarrow .S, \$]$  implies an accept action
- All other entries are error

# Parsing table - CALR

Stat	Action			goto	
e	c	d	\$	s	c
0	s3	s4		1	2
1			accept		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9

State	Action			goto	
	c	d	\$	s	c
7			r3		
8	r2	r2			
9			r2		

# Parsing algorithm

- Set input to point to the first symbol of w\$
- Repeat
  - Let s be the state on the top of the stack
  - Let a be the symbol pointed to by ip
  - If action [s, a] = shift  $s'$  then
    - Push a then  $s'$  on top of the stack
    - Move input to the next input symbol
  - Else if action [s, a] = reduce  $A \rightarrow \beta$  then
    - Pop 2 \* |  $\beta$  | symbols off the stack
    - Let  $s'$  be the state now on the top of the stack
    - Push A then goto  $[s', A]$  on top of the stack
    - Output the production  $A \rightarrow \beta$
  - Else if action[s, a] = accept then return;
  - Else error()

# Parsing with CALR parser

Stack	Input	Action
0	ccdd\$	[0, c] – shift 3
0 c 3	c d d \$	[3, c] – shift 3
0 c 3 c 3	d d \$	[3, d] – shift 4
0 c 3 c 3 d 4	d \$	[4, d] – reduce 3, pop 2 symbols from stack, push C, goto(3, C) = 8
0 c 3 c 3 C 8	d \$	[8, d] – reduce 2, pop 4 symbols from the stack, push C, goto(3, C) = 8
0 c 3 C 8	d \$	[8, d] – reduce 2, pop 4 symbols from the stack, push C, goto(0, C) = 2

Stack	Input	Action
0 C 2	d \$	[2, d] – shift 7
0 C 2 d 7	\$	[7, \$] – reduce 3, pop 2 symbols from the stack, goto(2, C) = 5
0 C 2 C 5	\$	[5, \$] – reduce 1, pop 4 symbols off the stack, goto(0, S) = 1
0 S 1	\$	[1, \$] – accept – successful parsing

# Example

- $S' \rightarrow S$
- $S \rightarrow L = R$
- $S \rightarrow R$
- $L \rightarrow * R$
- $L \rightarrow \text{id}$
- $R \rightarrow L$

# Another Example

- $I_0$   
 $[S' \rightarrow \bullet S, \$] \text{ goto}(I_0, S) = I_1$   
 $[S \rightarrow \bullet L=R, \$] \text{ goto}(I_0, L) = I_2$   
 $[S \rightarrow \bullet R, \$] \text{ goto}(I_0, R) = I_3$   
 $[L \rightarrow \bullet *R, =/\$] \text{ goto}(I_0, *) = I_4$   
 $[L \rightarrow \bullet \text{id}, =/\$] \text{ goto}(I_0, \text{id}) = I_5$   
 $[R \rightarrow \bullet L, \$] \text{ goto}(I_0, L) = I_2$
- $I_1 : \text{ goto}(I_0, S)$   
 $[S' \rightarrow S\bullet, \$]$
- $I_2 : \text{ goto}(I_0, L)$   
 $[S \rightarrow L\bullet=R, \$] \text{ goto}(I_2, =) = I_6$   
 $[R \rightarrow L\bullet, \$]$
- $I_3 : \text{ goto}(I_0, R)$   
 $[S \rightarrow R\bullet, \$]$
- $I_4 : \text{ goto}(I_0, *) \text{ goto}(I_4, *)$   
 $[L \rightarrow * \bullet R, =/\$] \text{ goto}(I_4, R) = I_7$   
 $[R \rightarrow \bullet L, =/\$] \text{ goto}(I_4, L) = I_8$   
 $[L \rightarrow \bullet *R, =/\$] \text{ goto}(I_4, *) = I_4$   
 $[L \rightarrow \bullet \text{id}, =/\$] \text{ goto}(I_4, \text{id}) = I_5$
- $I_5 : \text{ goto}(I_0, \text{id}) \text{ goto}(I_4, \text{id})$   
 $[L \rightarrow \text{id}\bullet, =/\$]$

- $I_6 : \text{goto}(I_2, =)$

$[S \rightarrow L=\bullet R, \$] \text{ goto}(I_6, R) = I_9$   
 $[R \rightarrow \bullet L, \$] \text{ goto}(I_6, L) = I_{10}$   
 $[L \rightarrow \bullet^* R, \$] \text{ goto}(I_6, *) = I_{11}$   
 $[L \rightarrow \bullet \text{id}, \$] \text{ goto}(I_6, \text{id}) = I_{12}$

- $I_7 : \text{goto}(I_4, R)$

$[L \rightarrow *R\bullet, =/\$]$

- $I_8 : \text{goto}(I_4, L)$

$[R \rightarrow L\bullet, =/\$]$

- $I_9 : \text{goto}(I_6, R)$

$[S \rightarrow L=R\bullet, \$]$

- $I_{10} : \text{goto}(I_6, L) \text{ goto}(I_{11}, L)$

$[R \rightarrow L\bullet, \$]$

- $I_{11} : \text{goto}(I_6, *) \text{ goto}(I_{11}, *)$

$[L \rightarrow * \bullet R, \$] \text{ goto}(I_{11}, R) = I_{13}$

$[R \rightarrow \bullet L, \$] \text{ goto}(I_{11}, L) = I_{10}$

$[L \rightarrow \bullet^* R, \$] \text{ goto}(I_{11}, *) = I_{11}$

$[L \rightarrow \bullet \text{id}, \$] \text{ goto}(I_{11}, \text{id}) = I_{12}$

- $I_{12} : \text{goto}(I_6, \text{id}) \text{ goto}(I_{11}, \text{id})$   
 $[L \rightarrow \mathbf{id}\bullet, \$]$

- $I_{13} : \text{goto}(I_{11}, R)$   
 $[L \rightarrow *R\bullet, \$]$

# Parsing Table

State	Action					goto		
	id	*	=	\$	S	L	R	
0	s5	s4			1	2	3	
1				accept				
2			s6	r5				
3				r2				
4	s5	s4				8	7	
5			r4	r4				
6	s12	s11				10	9	

State	Action				Goto		
	id	*	=	\$	S	L	R
7			r3	r3			
8			r5	r5			
9				r1			
10				r5			
11	s12	s11				10	13
12				r4			
13				r3			

# Summary

- CALR – most powerful parser
- Have so many items and states
- No conflicts

# Bottom-up Parser

# Bottom-up Parser

- LR methods (Left-to-right, Rightmost derivation)
  - SLR
  - Canonical LR (CALR)
  - Look Ahead LR (LALR)
- Other special cases:
  - Shift-reduce parsing
  - Operator-precedence parsing

# Bottom-up parser

- Bottom-up parsers build a derivation by working from the input back toward the start symbol
  - Builds parse tree from leaves to root
  - Builds reverse rightmost derivation

# Handle

- Since Bottom-up parsers match the RHS of production with LHS, a concept called ‘handle’ is defined
- A *handle* is a substring of grammar symbols in a *right-sentential form* that matches a right-hand side of a production
- A handle’s reduction to the non-terminal on the LHS represents one step along the reverse of a rightmost derivation
- This sub-string is a handle

# Handle - Example

- Expression Grammar Handles
  - id
  - $E * E$
  - $(E)$

# Shift Reduce Parser

- Simplest of the Bottom-up Parsers
- *Shift* input symbols until a handle is found.
- *Reduce* the substring to the non-terminal on the LHS of the corresponding production.

# Shift Reduce Parser

- A shift-reduce parser has 4 actions:
  - *Shift* the next input symbol is shifted onto the stack
  - *Reduce* the handle that is at top of stack
    - pop handle
    - push appropriate LHS symbol
  - *Accept* and stop parsing & report success
  - *Error* recovery routine is called

# Acceptance

- When the stack has the start symbol and the input is exhausted, the shift reduce parser goes to an accepting state

# Consider a grammar

- 1.  $E \rightarrow E + E$
- 2.  $E \rightarrow E * E$
- 3.  $E \rightarrow \mathbf{id}$

Stack	Action	Input
\$	shift	id + id * id \$
\$ <u>id</u>	Reduce by rule 3	+id*id \$
\$ E	shift	
\$ E+	Shift	id * id \$
\$ E + <u>id</u>	Reduce by rule 3	* id \$
\$ <u>E + E</u>	Reduce by Rule 1	* id \$
\$ E	Shift	* id \$
\$ E *	Shift	id \$

# Parsing action

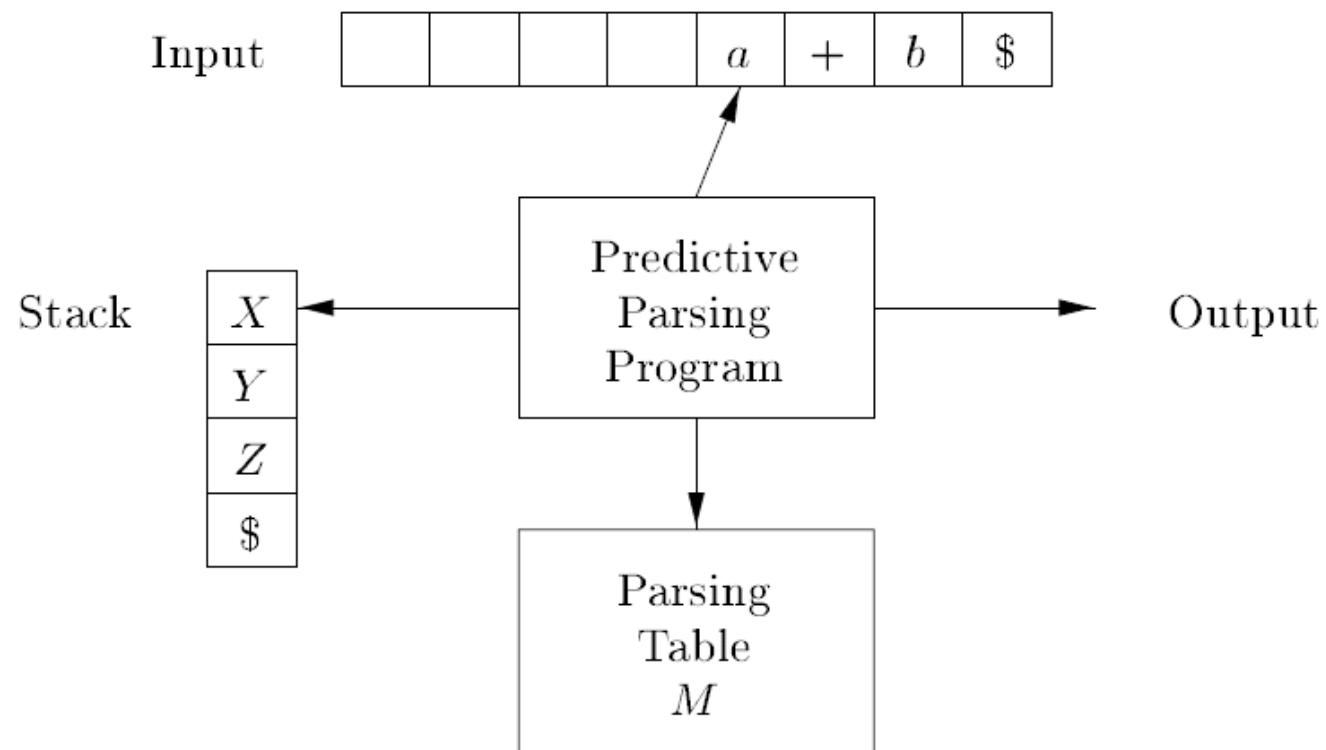
Stack	Action	Input
\$ E *	Shift	id \$
\$ E * <u>id</u>	Reduce by rule 3	\$
\$ <u>E * E</u>	Reduce by rule 2	\$
\$ E	Accept	\$

# Conflicts

- Shift-reduce and reduce-reduce conflicts are caused by
  - The limitations of the parsing method (even when the grammar is unambiguous)
  - Ambiguity of the grammar

# Non-Recursive Predictive Parser

# Non-Recursive Predictive Parsing



# FIRST()

- FIRST function is computed for all terminals and non-terminals
- $\text{FIRST}(\alpha)$  = the set of terminals that begin all strings derived from  $\alpha$

## FIRST ( )

- $\text{FIRST}(a) = \{a\}$  if  $a \in T$
- $\text{FIRST}(\varepsilon) = \{\varepsilon\}$
- $\text{FIRST}(A) = \bigcup_{A \rightarrow \alpha} \text{FIRST}(\alpha)$   
for  $A \rightarrow \alpha \in P$

# FIRST () – Algorithm

- $\text{FIRST}(X_1 X_2 \dots X_k) =$   
**if** for all  $j = 1, \dots, i-1 : \varepsilon \in \text{FIRST}(X_j)$  **then**  
    add non- $\varepsilon$  in  $\text{FIRST}(X_i)$  to  $\text{FIRST}(X_1 X_2 \dots X_k)$   
**if** for all  $j = 1, \dots, k : \varepsilon \in \text{FIRST}(X_j)$  **then**  
    add  $\varepsilon$  to  $\text{FIRST}(X_1 X_2 \dots X_k)$

# FOLLOW

- $\text{FOLLOW}(A)$  = the set of terminals that can immediately follow non-terminal  $A$

# FOLLOW - Algorithm

- FOLLOW( $A$ ) =
  - if**  $A$  is the start symbol  $S$  **then**
    - add  $\$$  to FOLLOW( $A$ )
  - for** all  $(B \rightarrow \alpha A \beta) \in P$  **do**
    - add FIRST( $\beta$ )\{\epsilon\} to FOLLOW( $A$ )
  - for** all  $(B \rightarrow \alpha A \beta) \in P$  and  $\epsilon \in \text{FIRST}(\beta)$  **do**
    - add FOLLOW( $B$ ) to FOLLOW( $A$ )
  - for** all  $(B \rightarrow \alpha A) \in P$  **do**
    - add FOLLOW( $B$ ) to FOLLOW( $A$ )

# Example

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

# FIRST

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

# FOLLOW

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

- $\text{FOLLOW}(E') = \text{FOLLOW}(E)$

$$= \{\$, )\}$$

- $\text{FOLLOW}(T') = \text{FOLLOW}(T)$

$$= \{\$, +, )\}$$

# Another Example

- Ambiguous grammar

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

- $\text{First}(S) = \{i, a\}$
- $\text{First}(S') = \{e, \epsilon\}$
- $\text{First}(C) = \{b\}$
- $\text{Follow}(S) = \{\$, e\}$
- $\text{Follow}(S') = \{\$, e\}$

# Predictive Parsing Table

- Row for each non-terminal
- Column for each terminal symbol
  - Table[NT, symbol] = Production that matches the [NT, symbol]
  - if First(NT) has  $\epsilon$ , then add production  
 $NT \rightarrow \epsilon$  in all [NT, a] for all 'a' in FOLLOW(NT)

# Parsing Table

	<b>id</b>	<b>+</b>	<b>*</b>	<b>(</b>	<b>)</b>	<b>\$</b>
<b>E</b>	$E \rightarrow TE'$			$E \rightarrow TE'$		
<b>E'</b>		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
<b>T</b>	$T \rightarrow FT'$			$T \rightarrow FT'$		
<b>T'</b>		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
<b>F</b>	$F \rightarrow id$			$F \rightarrow (E)$		

	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow i C t S S'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow e S$			$S' \rightarrow \epsilon$
C		$C \rightarrow b$				

# Parsing action

- `push($)`  
`push(S)`  
 $a := \text{lookahead}$
- **repeat**  
 $X := \text{pop()}$   
**if**  $X$  is a terminal or  $X = \$$  **then**  
    `match( $X$ )` // move to next token,  $a := \text{lookahead}$   
**else if**  $M[X,a] = X \rightarrow Y_1 Y_2 \dots Y_k$  **then**  
    `push( $Y_k, Y_{k-1}, \dots, Y_2, Y_1$ )` // such that  $Y_1$  is on top  
    produce output and/or invoke actions  
**else**   `error()`  
**endif**  
**until**  $X = \$$

# Parsing action Example

Stack	Input String	Action
\$E	id + id* id \$	[E, id]
\$E'T	id + id *id \$	[T, id]
\$E'T'F	id + id *id \$	[F, id]
\$E'T'id	id + id *id \$	id, id -> pop stack and move input
\$E'T'	+ id *id\$	[T', +] -> replace with ε
\$E'	+ id *id\$	[E', +]
\$E'T+	+ id *id\$	+, + → pop stack and move
\$E'T	id * id \$	[T, id]
\$E'T'F	id *id\$	[F, id]

Stack	Input String	Action
\$E'T'id	id *id\$	id, id → pop
\$E'T'	*id \$	[T', *]
\$E'T'F*	*id \$	*, * -> pop, and move
\$E'T'F	id\$	[F, id]
\$E'T'id	id \$	id, id → pop
\$E'T'	\$	T', \$ -> replace with ε
\$E'	\$	E', \$ -> replace with ε
\$	\$	Accept

# Error Recovery in LL (1) parser

- *Panic mode*
  - Discard input until a token in a set of designated synchronizing tokens is found
- *Phrase-level recovery*
  - Perform local correction on the input to repair the error
- *Error productions*
  - Augment grammar with productions for erroneous constructs
- *Global correction*
  - Choose a minimal sequence of changes to obtain a global least-cost correction

# Error Recovery

- Panic Mode
  - Add synchronizing actions to undefined entries based on FOLLOW
- Phrase Mode
  - Change input stream by inserting missing +, \*, (, or )  
For example: **id id** is changed into **id \* id or id + id**

# Error Recovery

- Error Production
  - Add productions that will take care of incorrect input combinations

# Error Recovery

	<b>id</b>	<b>+</b>	<b>*</b>	<b>(</b>	<b>)</b>	<b>\$</b>
<b>E</b>	$E \rightarrow TE'$			$E \rightarrow TE'$	<b>synch</b>	<b>synch</b>
<b>E'</b>		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
<b>T</b>	$T \rightarrow FT'$	<b>synch</b>		$T \rightarrow FT'$	<b>synch</b>	<b>synch</b>
<b>T'</b>		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
<b>F</b>	$F \rightarrow id$	<b>synch</b>	<b>synch</b>	$F \rightarrow (E)$	<b>synch</b>	<b>synch</b>

# LL (1)

- A grammar  $G$  is LL(1) if for each collections of productions

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$$

for nonterminal  $A$  the following holds:

1.  $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset$  for all  $i \neq j$
2. if  $\alpha_i \Rightarrow^* \varepsilon$  then
  - 2.a.  $\alpha_j \Rightarrow^* \varepsilon$  for all  $i \neq j$
  - 2.b.  $\text{FIRST}(\alpha_j) \cap \text{FOLLOW}(A) = \emptyset$   
for all  $i \neq j$

# If then Grammar

- The if then grammar has multiple entries in the parsing table.
- So, confusion on which production to apply
- Ambiguous grammar hence not LL (1)

# SLR Parsers, LR (0) items

# Bottom-up Parsers

- Simple Shift-reduce parsers has lot of Shift/Reduce conflicts
- Operator precedence parsers is for a small class of grammars
- Go for LR parsers

# LR Parsers

- LR(1) parsers recognize the languages in which one symbol of look-ahead is sufficient to decide whether to shift or reduce
  - L : for left-to-right scan of the input
  - R : for reverse rightmost derivation
  - 1: for one symbol of look-ahead

# LR Parsers

- Read input, one token at a time
- Use stack to keep track of current state
  - The state at the top of the stack summarizes the information below.
  - The stack contains information about what has been parsed so far.

# LR Parsers

- Use parsing table to determine action based on current state and look-ahead symbol.
- Parsing table construction takes into account the shift, reduce, accept or error action

# LR Parsers

- SLR
  - Simple LR parsing
  - Easy to implement, but not powerful
  - Uses LR(0) items
- Canonical LR
  - Larger parser but powerful
  - Uses LR(1) items

- LALR
  - Condensed version of canonical LR
  - May introduce conflicts
  - Uses LR(1) items

# SLR Parsers - Handle

- As a SLR parser processes the input, it must identify all possible handles.
- For example, consider the usual expression grammar and the input string  
 $a + b.$

# SLR Parsers

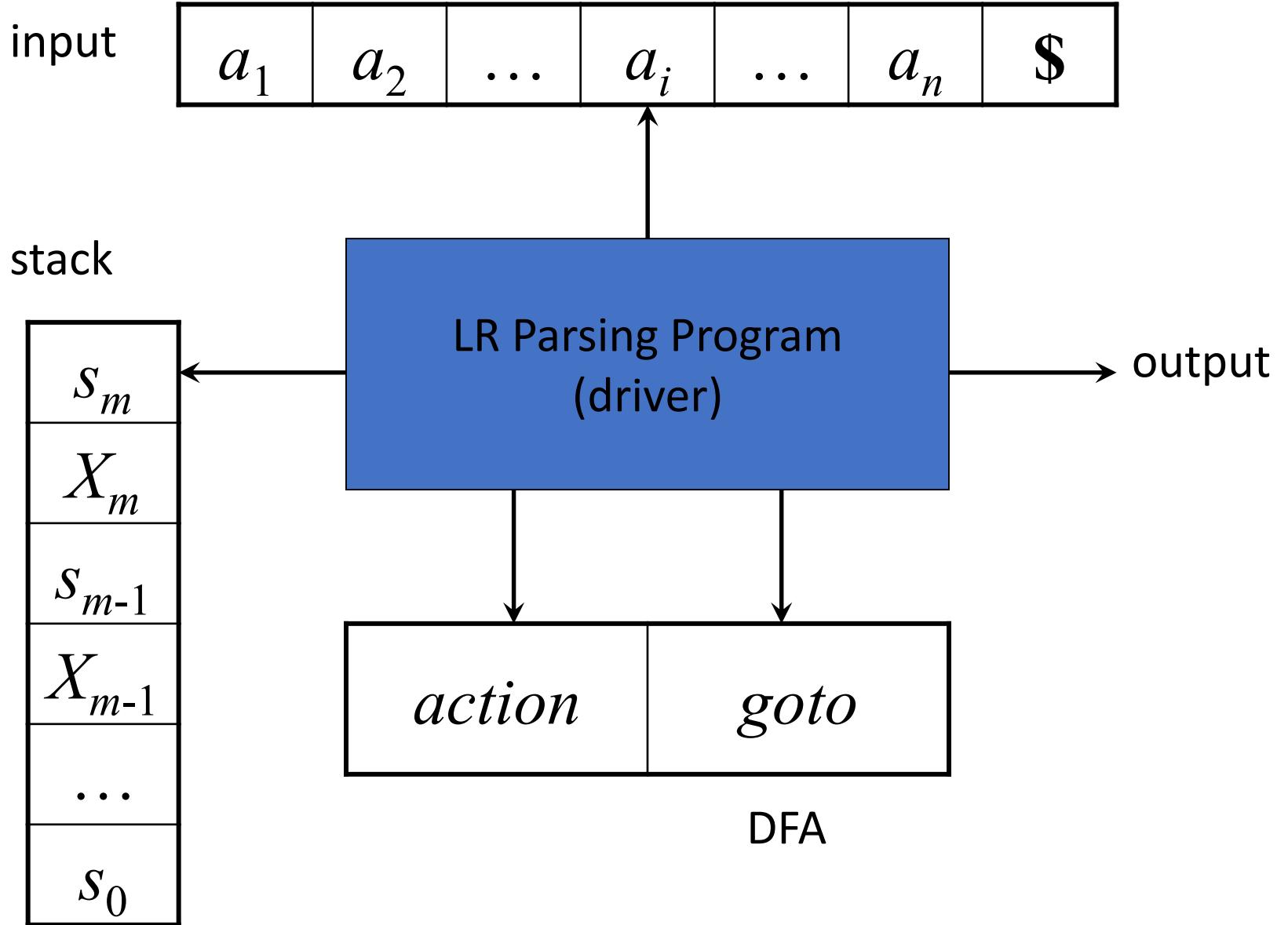
- If the parser has processed ‘a’ and reduced it to E. Then, the current state can be represented by  $E \bullet +E$  where • means
  - E has already been parsed and
  - $+E$  is a potential suffix, which, if determined, yields to a successful parse.

# SLR parsers

- Our ultimate aim is to finally reach state  $E+E\bullet$ , which corresponds to an actual handle yielding to the reduction  $E \rightarrow E+E$

# SLR Parsers

- LR parsing works by building an automata where each state represents what has been parsed so far and what we intend to parse after looking at the current input symbol. This is indicated by productions having a “.” These productions are referred to as items.
- Items that has the “.” at the end leads to the reduction by that production



# SLR (1) Parser

- Form the augmented grammar
- Construction of LR(0) items
- Construct the follow() for all the non-terminals which requires construction of first() for all the terminals and non-terminals

# SLR(1) parser

- Using this and the follow( ) of the grammar, construct the parsing table
- Using the parsing table, a stack and an input parse the input

# LR (0) items

- An *LR(0) item* of a grammar  $G$  is a production of  $G$  with a • at some position of the right-hand side
- Thus, a production

$$A \rightarrow XYZ$$

has four items:

$$[A \rightarrow \bullet XYZ]$$

$$[A \rightarrow X \bullet Y Z]$$

$$[A \rightarrow X Y \bullet Z]$$

$$[A \rightarrow X Y Z \bullet]$$

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

# LR (0) items

- The grammar is augmented with a new start symbol  $S'$  and production  $S' \rightarrow S$
- Initially, set  $C = \text{closure}(\{[S' \rightarrow \bullet S]\})$
- For each set of items  $I \in C$  and each grammar symbol  $X \in (N \cup T)$  such that  $\text{goto}(I, X) \notin C$  and  $\text{goto}(I, X) \neq \emptyset$ ,
  - add the set of items  $\text{goto}(I, X)$  to  $C$
- Repeat until no more sets can be added to  $C$

# Closure ( $I$ )

- Start with  $\text{closure}(I) = I$
- If  $[A \rightarrow \alpha \bullet B\beta] \in \text{closure}(I)$  then for each production  $B \rightarrow \gamma$  in the grammar, add the item  $[B \rightarrow \bullet\gamma]$  to  $\text{closure}(I)$  if it is not already there
- Repeat 2 until no new items can be added to  $\text{closure}(I)$

# Goto ( $I$ , $X$ )

- For each item  $[A \rightarrow \alpha \bullet X\beta] \in I$ , add the set of items  $\text{closure}(\{[A \rightarrow \alpha X \bullet \beta]\})$  to  $\text{goto}(I, X)$  if not already there
- Repeat until no more items can be added to  $\text{goto}(I, X)$
- Intuitively,  $\text{goto}(I, X)$  is the set of items that are valid for the viable prefix  $\gamma X$  when  $I$  is the set of items that are valid for  $\gamma$

# Augmented Grammar

$$E' \rightarrow E$$

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

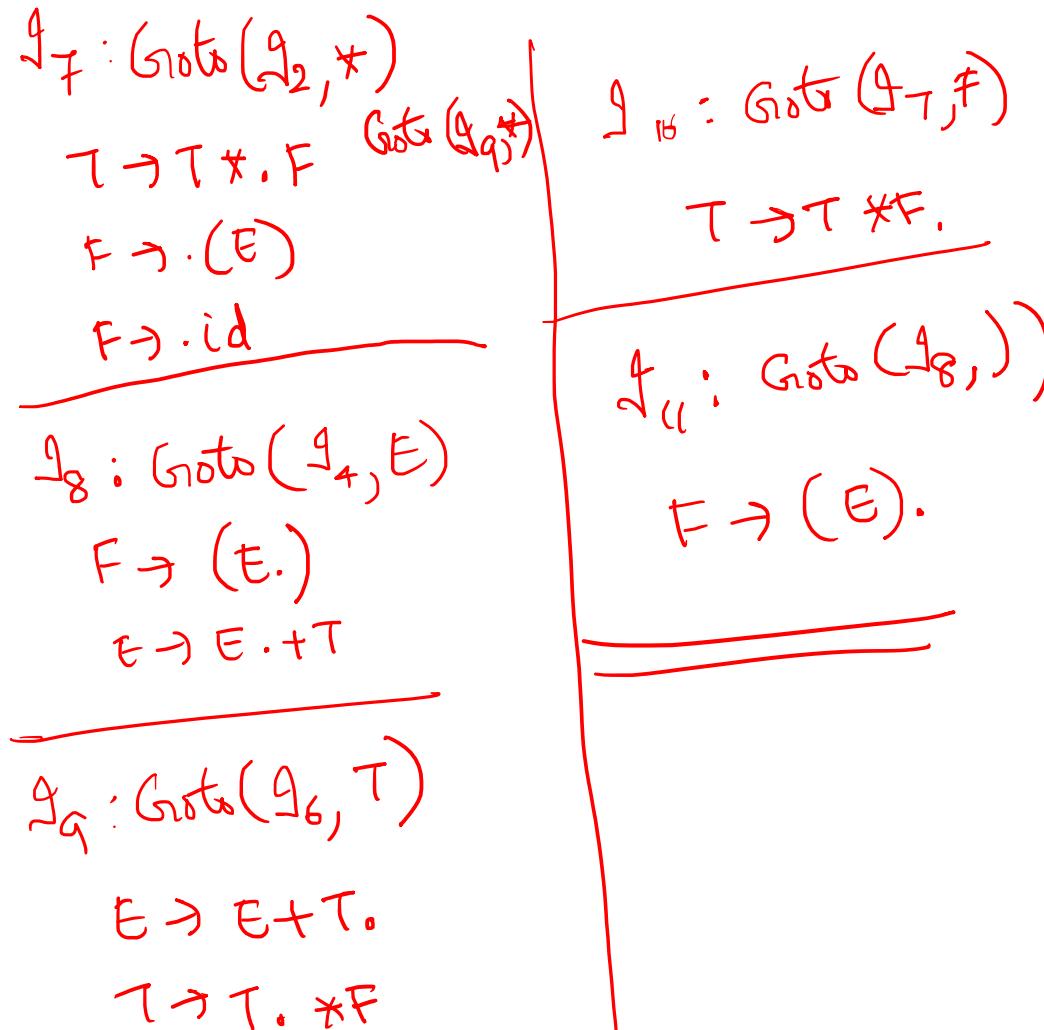
# Augmented Grammar

$$E' \rightarrow E$$

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

$I_0$

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





# Items

- $I_0$
- $E' \rightarrow .E$
- $E \rightarrow .E + T$
- $E \rightarrow .T$
- $T \rightarrow .T * F$
- $T \rightarrow .F$
- $F \rightarrow .(E)$
- $F \rightarrow .id$
- $I_1 = \text{Goto}(I_0, E)$
- $E' \rightarrow E.$
- $E \rightarrow E. + T$
- $I_2 = \text{Goto}(I_0, T), \text{Goto}(I_4, T),$   
 $E \rightarrow T.$
- $T \rightarrow T.*F$

- $I_3 = \text{Goto}(I_0, F), \text{Goto}(I_4, F), \text{Goto}(I_6, F)$
- $T \rightarrow F.$
- $I_5 = \text{Goto}(I_0, \text{id}), \text{Goto}(I_4, \text{id}), \text{Goto}(I_6, \text{id}), \text{Goto}(I_7, \text{id})$   
 $F \rightarrow \text{id}.$
- $I_4 = \text{Goto}(I_0, ( ), \text{Goto}(I_4, (), \text{Goto}(I_6, (), \text{Goto}(I_7, ()$
- $F \rightarrow (.E)$
- $E \rightarrow .E + T$
- $E \rightarrow .T$
- $T \rightarrow .T * F$
- $T \rightarrow .F$
- $F \rightarrow .(E)$
- $F \rightarrow .\text{id}$

# Items

$I_6 = \text{Goto}(I_1, +), \text{Goto}(I_8, +),$

$E \rightarrow E + . T$

$T \rightarrow . T * F$

$T \rightarrow . F$

$F \rightarrow .(E)$

$F \rightarrow .id$

$I_8 = \text{Goto}(I_4, E)$

$F \rightarrow (E.)$

$E \rightarrow E . + T$

$I_9 = \text{Goto}(I_6, T)$

$E \rightarrow E + T.$

$T \rightarrow T . * F$

$I_7 : \text{Goto}(I_2, *), \text{Goto}(I_9, * )$

$T \rightarrow T * . F$

$F \rightarrow .(E)$

$F \rightarrow .id$

$I_{10} : \text{Goto}(I_7, F )$

$T \rightarrow T * F.$

$I_{11} : \text{Goto}(I_8, ) )$

$F \rightarrow (E).$

# SLR Parsing Table

- Input: Augmented Grammar  $G'$
- Output: SLR parsing table with functions, shift, reduce and accept
- Parsing table is between items and Terminals and non-terminals
- The non-terminals correspond to the `goto()` of the items set
- The terminals have the parsing table corresponding to the action – shift / reduce/accept

# SLR Parsing Table

- Augment the grammar with  $S' \rightarrow S$
- Construct the set  $C = \{I_0, I_1, \dots, I_n\}$  of  $LR(0)$  items
- If  $[A \rightarrow \alpha \bullet a\beta] \in I_i$  and  $goto(I_i, a) = I_j$  then set  $action[i, a] = \text{shift } j$ , where  $a$  is a terminal
- If  $[A \rightarrow \alpha \bullet] \in I_i$  then set  $action[i, a] = \text{reduce } A \rightarrow \alpha$  for all  $a \in FOLLOW(A)$  where  $A \neq S'$

# SLR parsing table

- If  $[S' \rightarrow S \bullet]$  is in  $I_i$ , then set  $action[i, \$] = \text{accept}$
- If  $goto(I_i, A) = I_j$  then set  $goto[i, A] = j$
- Repeat for all the items until no more entries added
- The initial state  $i$  is the  $I_i$  holding item  $[S' \rightarrow \bullet S]$
- All other entries are error

# Grammar

- $E' \rightarrow E$
- 1 •  $E \rightarrow E + T$
- 2 •  $E \rightarrow T$
- 3 •  $T \rightarrow T * F$
- 4 •  $T \rightarrow F$
- 5 •  $F \rightarrow (E)$
- 6 •  $F \rightarrow id$

# Follow

- $\text{Follow}(E) = \{\$, +, )\}$
- $\text{Follow}(T) = \{\$, +, *, )\}$
- $\text{Follow}(F) = \{\$, +, *, )\}$

- $s_i$  means shift state  $i$
- $r_j$  means reduce by production numbered  $j$
- Blank means error

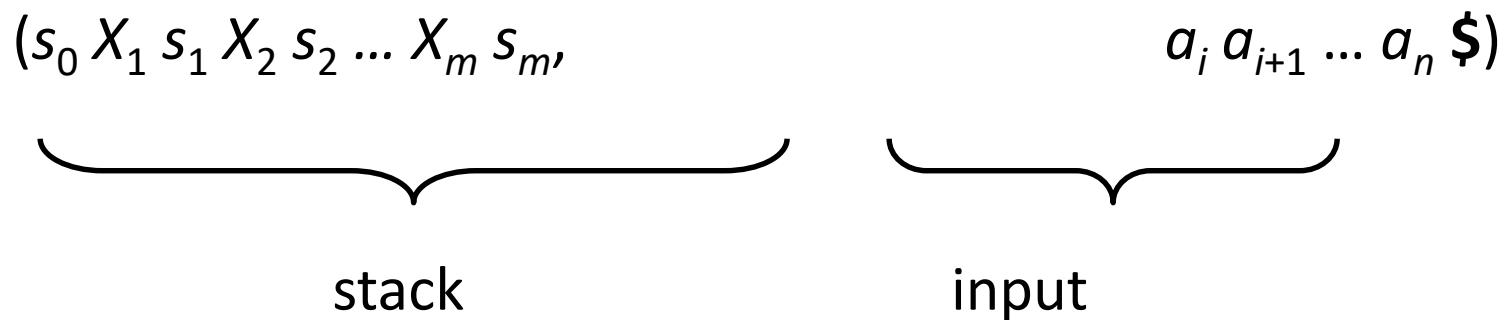
# Shift, Accept, Reduce

State	Action							Goto		
	id	+	*	(	)	\$	E	T	F	
0	s5				s4		1	2	3	
1		s6				accept				
2		r2	s7		r2	r2				
3		r4	r4		r4	r4				
4	s5				s4		8	2	3	
5		r6	r6		r6	r6				
6	s5				s4			9	3	

# Shift, Accept and Reduce

State	Action							Goto		
	id	+	*	(	)	\$	E	T	F	
7	s5				s4					10
8			s6			s11				
9		r1	s7			r1	r1			
10		r3	r3		r3	r3				11
11		r <u>5</u>	r5		r5	r5				

# SLR Parsing



# Parsing action

- If  $\text{action}[s_m, a_i] = \text{shift } s$ , then push  $a_i$ , push  $s$ , and advance input:  
 $(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m a_i s, \ a_{i+1} \dots a_n \$)$
- If  $\text{action}[s_m, a_i] = \text{reduce } A \rightarrow \beta$  and  $\text{goto}[s_{m-r}, A] = s$  with  $r = |\beta|$  then  
pop  $2r$  symbols, push  $A$ , and push  $s$ :  
 $(s_0 X_1 s_1 X_2 s_2 \dots X_{m-r} s_{m-r} A s, \ a_i a_{i+1} \dots a_n \$)$

- If  $\text{action}[s_m, a_i] = \text{accept}$ , then stop
- If  $\text{action}[s_m, a_i] = \text{error}$ , then attempt recovery

# Parsing algorithm

- Set input to point to the first symbol of w\$
- Repeat
  - Let s be the state on the top of the stack
  - Let a be the symbol pointed to by ip
  - If action [s, a] = shift s' then
    - Push a then s' on top of the stack
    - Move input to the next input symbol
  - Else if action [s, a] = reduce A → β then
    - Pop 2 \* | β | symbols off the stack
    - Let s' be the state now on the top of the stack
    - Push A then goto [s', A] on top of the stack
    - Output the production A → β
  - Else if action[s, a] = accept then return;
  - Else error()

# Parsing action

Stack	Input	Action
0	id * id + id \$	[0, id] → s5 , shift
0 id 5	* id + id \$	[5, *] → r6, pop 2 symbols, Goto[0, F] → 4
0 F 3	* id + id \$	[3, *] → r4, pop 2 symbols, Goto[0, T] → 2
0 T 2	* id + id \$	[2, *] → s7, shift
0 T 2 * 7	id + id \$	[7, id] → s5, shift
0 T 2 * 7 id 5	+ id \$	[5, +] → r6, pop 2 symbols, Goto[7, F] → 10
0 T 2	+ id \$	[2, +] → r2, pop 2 symbols and goto [0 , E] → 1

# Parsing action

Stack	Input	Action
0 E 1	+ id \$	[1, +] → s6, shift
0 E 1 + 6	id \$	[6, id] → s5, shift
0 E 1 + 6 id 5	\$	[5, \$] → r6, pop 2 symbols, goto [6, F] → 3
0 E 1 + 6 F 3	\$	[3 , \$] → r4, pop 2 symbols, goto [ 6, T] → 9
0 E 1 + 6 T 9	\$	[9, \$] → r1, pop 6 symbols, goto [0, E] → 1
0 E 1	\$	[1, \$] → accept, hence successful parsing

# Problems with SLR grammar

- Every SLR grammar is unambiguous, but **not** every unambiguous grammar is SLR
- Consider for example the unambiguous grammar

# Example

- $S \rightarrow L = R$
- $S \rightarrow R$
- $L \rightarrow * R$
- $L \rightarrow id$
- $R \rightarrow L$

# Items set

- $I_0$ :  
 $S' \rightarrow \bullet S$ 
  1.  $S \rightarrow \bullet L = R$
  2.  $S \rightarrow \bullet R$
  3.  $L \rightarrow \bullet^* R$
  4.  $L \rightarrow \bullet \text{id}$
  5.  $R \rightarrow \bullet L$
- $I_1$ :  $(I_0, S)$   
 $S' \rightarrow S \bullet$
- $I_2$ :  $(I_0, L)$   
 $S \rightarrow L \bullet = R$   
 $R \rightarrow L \bullet$
- $I_3$ :  $(I_0, R)$   
 $S \rightarrow R \bullet$

# Items set

- $I_4: (I_0, *) (I_4, *) (I_6, *)$   
 $L \rightarrow * \bullet R$   
 $R \rightarrow \bullet L$   
 $L \rightarrow \bullet^* R$   
 $L \rightarrow \bullet \mathbf{id}$
- $I_5: (I_0, \mathbf{id}) (I_4, \mathbf{id}) (I_6, \mathbf{id})$   
 $L \rightarrow \mathbf{id} \bullet$
- $I_9: (I_6, R)$   
 $S \rightarrow L = R \bullet$
- $I_6: (I_2, =)$   
 $S \rightarrow L = \bullet R$   
 $R \rightarrow \bullet L$   
 $L \rightarrow \bullet^* R$   
 $L \rightarrow \bullet \mathbf{id}$
- $I_7: (I_4, R)$   
 $L \rightarrow * R \bullet$
- $I_8: (I_4, L) (I_6, L)$   
 $R \rightarrow L \bullet$

- $\text{Follow}(S) = \{ \$ \}$
- $\text{Follow}(L) = \{ =, \$ \}$
- $\text{Follow}(R) = \{ \$, = \}$

State	Action				Goto		
	id	=	*	\$	S	L	R
0	s5		s4		1	2	3
1				accept			
2			s6 / r5		r5		
3				r2			
4	s5		s4			8	7
5		r4		r4			
6	s5		s4			8	9

State	Action				Goto		
	id	=	*	\$	S	L	R
7		r3		r3			
8		r5		r5			
9				r1			

0  
0 id5

0 L2

0 R3

$$\begin{aligned} id &= *id \$ \\ &= *id \$ \\ &= *id \$ \\ &= *id \$ \end{aligned}$$

r4 L → id  
r5 R → L

Reduce → Error

shift

0 L2 = 6

\* id \$

0 L2 = 6 \* 4

id \$

0 L2 = 6 \* 4 id5

\$ L → id

0 L2 = 6 \* 4 L8

\$ R → L

0 L2 = 6 \* 4 RT

\$ R3

0 L2 = 6 L8

\$ R → L

0 L2 = 6 R9

\$ R1

0 S1

\$ accept

# Conflict

- Shift / reduce conflict arises
- Because the grammar is not SLR(1)
- Follow information alone is not sufficient
- Hence, powerful parser is required

# Summary

- Learnt to parse the SLR(1) grammar using the SLR(1) parsing algorithm
- Some grammar results in Shift / Reduce conflict

# Parser

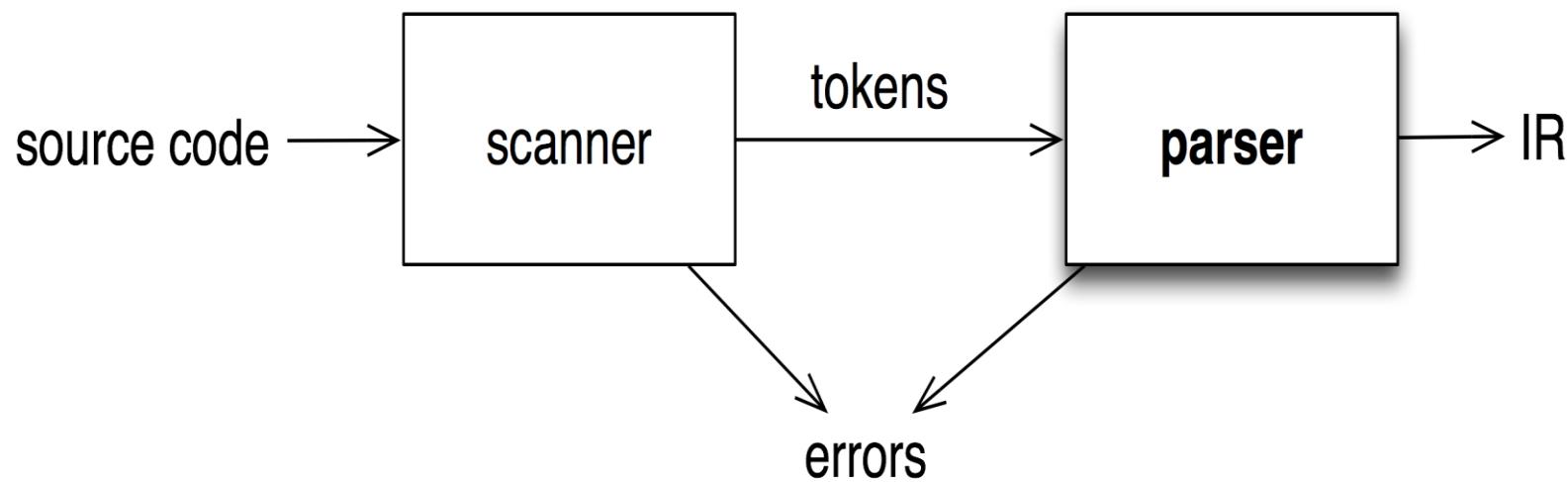
# Second Phase of the compiler

- Parser – Typically integrated with the lexical phase of the compiler
- Top Down Parser
- Bottom Up Parser

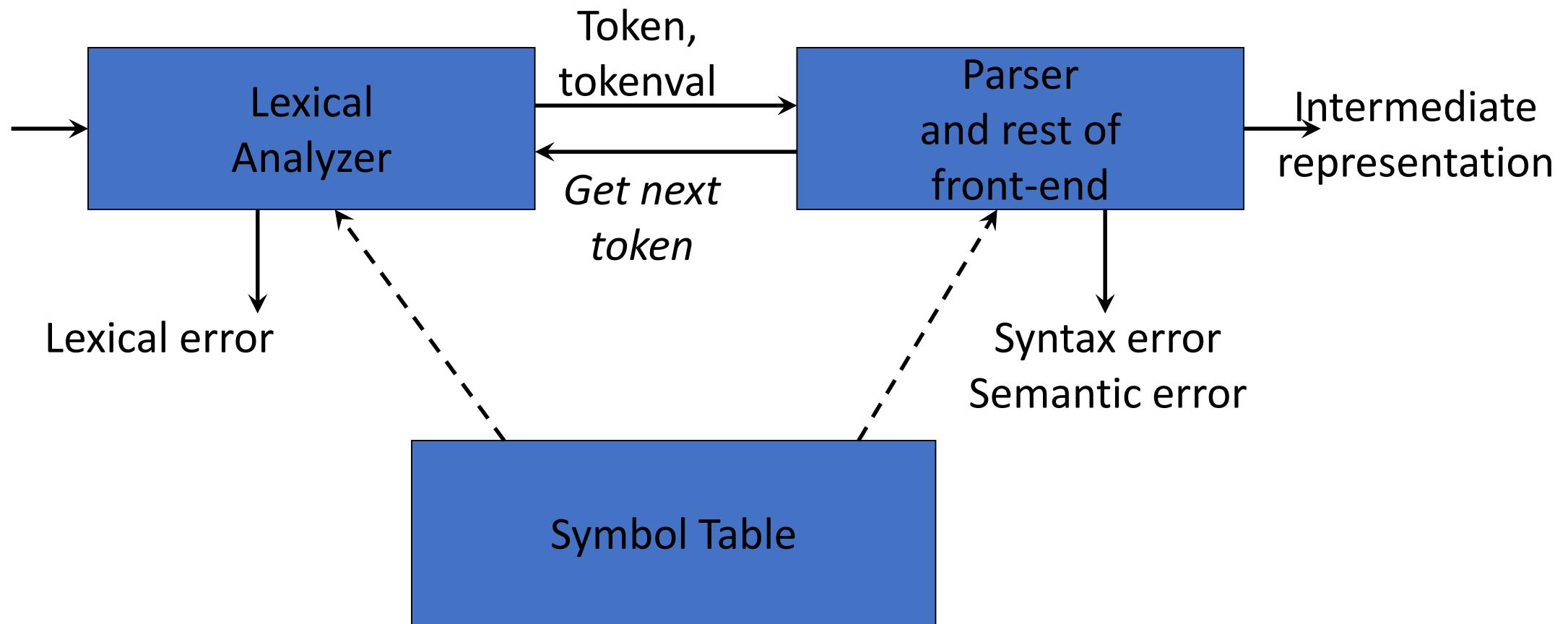
# Functions of the Parser

- Validate the syntax of the programming language
- Points out errors in the statements

# Role of the Parser



# Role of the Parser



# General Types of Parsers

- Universal Parsers
  - Cocke- Younger-Kasami
  - Earley's Algorithm
- Top-Down Parsers
- Bottom Up Parsers

# Universal Parsers

- Can parse any Grammar
- Use in NLP
- But too inefficient in Compilers

# Top Down Parsers

- Build the parse trees from the top to the bottom
- Recursive Descent parsers – requires backtracking
- LL Parsers – No Backtracking

# Example

Consider the Grammar

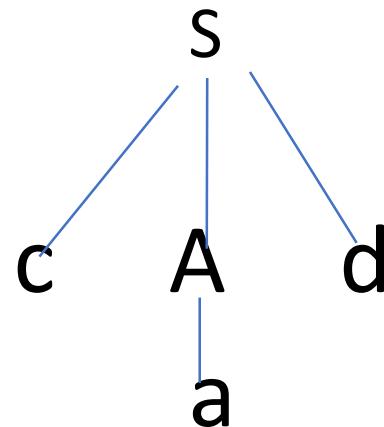
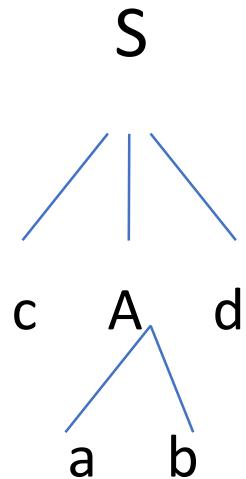
$$S \rightarrow c A d$$

$$A \rightarrow ab \mid a$$

Let the input be “cad”

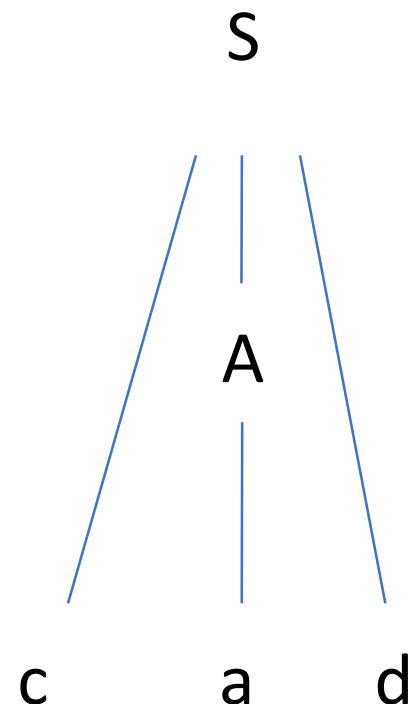
# Parsing (Recursive Descent)

Expand A using **the first alternative**  $A \rightarrow ab$



# Bottom up Parsers

- Start from the bottom and work up to the root for parsing a string
- LR parsers are bottom up parsers



# Parsing

- Both Top Down and Bottom up parsers parse the string based on a viable-prefix property
- This property states that before the string is fully processed, if there is an error, the parser will identify it and recovers

# Context Free Grammars - CFG

- Programming language constructs are defined using context free grammar
- For example

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

Expression grammar involving the operators, +, \*, ( )

# Context Free Grammars

- Defined formally as  $(V, T, P, S)$ 
  - $V$  – Variables / Non-terminals
  - $T$  – Terminals that constitute the string
  - $P$  – Set of Productions that has a LHS and RHS
  - $S$  – Special Symbol, subset of  $V$

# Context Free Grammar

- Example

$\text{stmt} \rightarrow \text{if E then stmt else stmt}$

$\text{stmt} \rightarrow \text{if E then stmt}$

$\text{stmt} \rightarrow a$

$E \rightarrow b$

Here,  $\text{stmt}$ ,  $E$  are Non-terminals,

$\text{if}$ ,  $\text{then}$ ,  $\text{else}$ ,  $a$ ,  $b$  are all terminals

# Grammar - notations

- Terminals -  $a, b, c, \dots \in T$ 
  - specific terminals: **0**, **1**, **id**, **+**
- Non-terminals -  $A, B, C, \dots \in N$ 
  - specific non-terminals: *expr*, *term*, *stmt*
- Grammar symbols -  $X, Y, Z \in (N \cup T)$
- Strings of terminals
  - $u, v, w, x, y, z \in T^*$
- Strings of grammar symbols
  - $\alpha, \beta, \gamma \in (N \cup T)^*$

# Derivation

- The *one-step derivation* is defined by

$$\alpha A \beta \Rightarrow \alpha \gamma \beta$$

where  $A \rightarrow \gamma$  is a production in the grammar

- In addition, we define

- $\Rightarrow$  is *leftmost*  $\Rightarrow_{lm}$  if  $\alpha$  does not contain a nonterminal
- $\Rightarrow$  is *rightmost*  $\Rightarrow_{rm}$  if  $\beta$  does not contain a nonterminal
- Transitive closure  $\Rightarrow^*$  (zero or more steps)
- Positive closure  $\Rightarrow^+$  (one or more steps)

# Derivation

- The *language generated by G* is defined by  
$$L(G) = \{w \mid S \Rightarrow^+ w\}$$

# Derivation

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow ( E )$$

$$E \rightarrow - E$$

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

$$E \Rightarrow - E \Rightarrow - \mathbf{id}$$

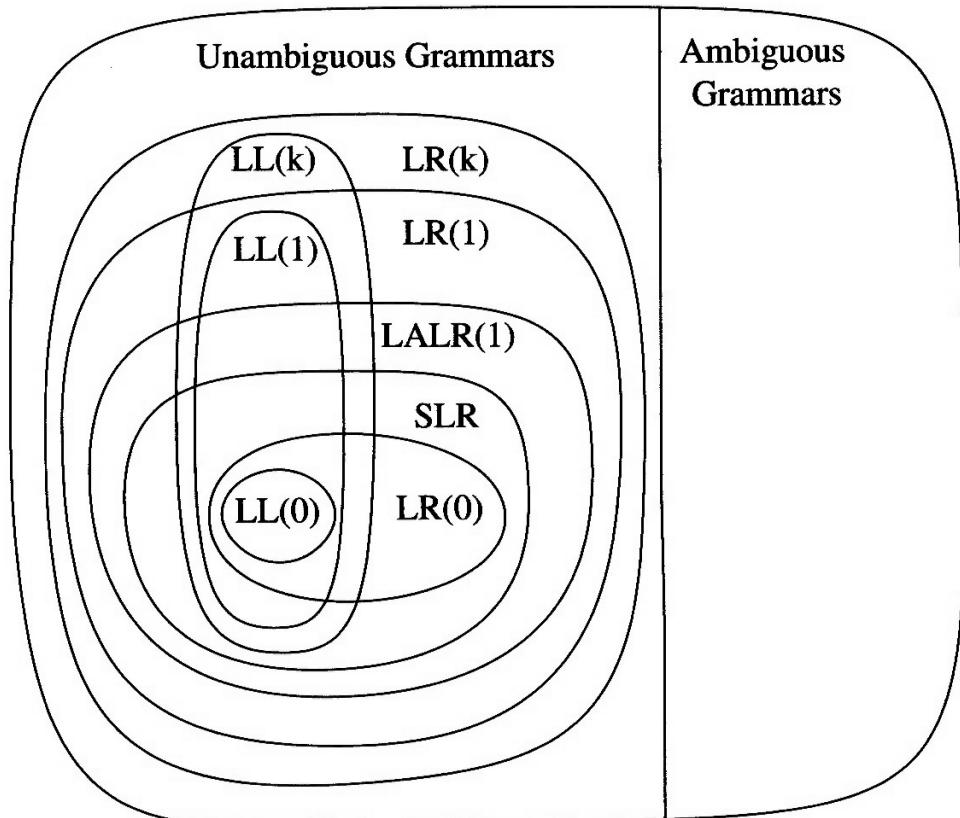
$$E \Rightarrow_{rm} E + E \Rightarrow_{rm} E + \mathbf{id} \Rightarrow_{rm} \mathbf{id} + \mathbf{id}$$

$$E \Rightarrow^* E \quad E \Rightarrow^+ \mathbf{id} * \mathbf{id} + \mathbf{id}$$

# Parsers

- Context Free grammars are already defined for all programming constructs
- All strings that are part of the programming language will be based on this construct
- Hence, parsers are designed keeping in mind the CFG

# Hierarchy of Grammar Classes



# Hierarchy

- **LL( $k$ ):**
  - Left-to-right, Leftmost derivation,  $k$  tokens lookahead
- **LR( $k$ ):**
  - Left-to-right, Rightmost derivation,  $k$  tokens lookahead
- **SLR:**
  - Simple LR (uses “follow sets”)
- **LALR:**
  - LookAhead LR (uses “lookahead sets”)

# Top Down Parsers

- LL methods (Left-to-right, Leftmost derivation) and recursive-descent parsing

Grammar:

$$E \rightarrow T + T$$

$$T \rightarrow ( E )$$

$$T \rightarrow - E$$

$$T \rightarrow \mathbf{id}$$

Leftmost derivation:

$$E \xrightarrow{Im} T + T$$

$$\xrightarrow{Im} \mathbf{id} + T$$

$$\xrightarrow{Im} \mathbf{id} + \mathbf{id}$$

# Top Down Parsers – LL (1) Parsers

- LL parsers cannot handle
  - Left Recursive Grammar
  - Left Factoring

# Left Recursive Grammar

- *Formally, a grammar is left recursive if  $\exists A \in NT$  such that  $\exists$  a derivation  $A \Rightarrow^+ A\alpha$ , for some string  $\alpha \in (NT \cup T)^+$*
- $A \rightarrow A\alpha / \beta / \gamma$

# Left Factor

- When a non-terminal has two or more productions whose right-hand sides start with the same grammar symbols the grammar is said to have left-factor property
- *Example*  $A \rightarrow \alpha \beta_1 / \alpha \beta_2 / \dots / \alpha \beta_n / \gamma$

# Pre-requisites for Top-Down Parser

- Eliminate Left Recursion
- Left Factor the grammar

$$\begin{array}{l} A \xrightarrow{\alpha\beta} \\ A \rightarrow A\alpha \mid \textcircled{B}\beta \\ B \rightarrow A\gamma \mid \epsilon \end{array}$$

# Eliminating Left Recursion

Arrange the non-terminals in some order  $A_1, A_2, \dots, A_n$

**for**  $i = 1, \dots, n$  **do**

**for**  $j = 1, \dots, i-1$  **do**

        replace each

        with  $A_i \rightarrow A_j \gamma$

        where  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$

$A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$

**end**

    eliminate the immediate left recursion in  $A_i$

**end**

# Eliminate Left Recursion

- Rewrite every left-recursive production

$$A \rightarrow A\alpha / \beta | \gamma | A\delta$$

- into a right-recursive production:

$$A \rightarrow \beta A_R / \gamma A_R$$

$$A_R \rightarrow \alpha A_R / \delta A_R / \varepsilon$$

# Example

- $A \rightarrow B C \mid a$   
 $\underline{B} \rightarrow CA \mid A \mathbf{b}$   
 $C \rightarrow A B \mid CC \mid a$

- $i = 1$ : nothing to do

$i = 2, j = 1$ :  $B \rightarrow CA \mid \underline{A} \mathbf{b}$

$\Rightarrow B \rightarrow CA \mid \underline{B} C \mathbf{b} \mid \underline{\mathbf{a}} \mathbf{b}$

$\Rightarrow_{(\text{imm})} B \rightarrow CA B_R \mid \mathbf{a} \mathbf{b} B_R$

$B_R \rightarrow C \mathbf{b} B_R \mid \varepsilon$

$i = 3, j = 1$ :  $C \rightarrow \underline{A} B \mid CC \mid \mathbf{a}$

$\Rightarrow C \rightarrow \underline{B} C B \mid \underline{\mathbf{a}} B \mid CC \mid \mathbf{a}$

- $i = 3, j = 2$ :  $C \rightarrow \underline{B} C B \mid a B \mid C C \mid a$   
 $\Rightarrow C \rightarrow \underline{C A B_R} C B \mid \underline{a b B_R} C B \mid a B \mid C C \mid a$   
 $\Rightarrow_{(imm)} C \rightarrow a b B_R C B C_R \mid a B C_R \mid a C_R$   
 $C_R \rightarrow A B_R C B C_R \mid C C_R \mid \varepsilon$

# Example - Expression Grammar

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

# Modified Grammar

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

# Left Factoring

- Replace productions

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

with

$$A \rightarrow \alpha A_R / \gamma$$

$$A_R \rightarrow \beta_1 / \beta_2 / \dots / \beta_n$$

# Left Factoring

**METHOD:** For each nonterminal  $A$ , find the longest prefix  $\alpha$  common to two or more of its alternatives. If  $\alpha \neq \epsilon$  — i.e., there is a nontrivial common prefix — replace all of the  $A$ -productions  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma$ , where  $\gamma$  represents all alternatives that do not begin with  $\alpha$ , by

$$\begin{aligned} A &\rightarrow \alpha A' \mid \gamma \\ A' &\rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n \end{aligned}$$

Here  $A'$  is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.  $\square$

# Left Factoring - Example

$S \rightarrow \underline{iCtS} \mid \underline{iCtSeS} \mid a$

$C \rightarrow b$

# Left Factoring

- $S \rightarrow i\underset{\text{CT}}{\underline{CTSS'}} \mid a$
- $S' \rightarrow eS \mid \epsilon$
- $C \rightarrow b$

# LL (1) Parser – Predictive parser

- L – input is scanned from left to right
- L – left derivation
- (1) – looking at 1 input symbol

# Predictive Parser LL (1)

- Eliminate left recursion from grammar
- Left factor the grammar
- Compute FIRST and FOLLOW
- Two variants:
  - Recursive (recursive calls)
  - Non-recursive (table-driven)

# Recursive descent with Recursive calls

- Recursive-descent parsing is a top-down method of syntax analysis in which a set of recursive procedures is used to process the input
- One procedure is associated with each nonterminal of a grammar
- A simple form of recursive-descent parsing, called predictive parsing, in which the lookahead symbol unambiguously determines the flow of control through the procedure body for each nonterminal
- The sequence of procedure calls during the analysis of an input string implicitly defines a parse tree for the input, and can be used to build an explicit parse tree, if desired.

```
void A() {  
    Choose an  $A$ -production,  $A \rightarrow X_1 X_2 \cdots X_k$ ;  
    for (  $i = 1$  to  $k$  ) {  
        if (  $X_i$  is a nonterminal )  
            call procedure  $X_i()$ ;  
        else if (  $X_i$  equals the current input symbol  $a$  )  
            advance the input to the next symbol;  
        else /* an error has occurred */;  
    }  
}
```

# Recursive descent with Recursive calls

```
stmt → expr ;
      | if ( expr ) stmt
      | for ( optexpr ; optexpr ; optexpr ) stmt
      | other
```

```
optexpr → ε
        | expr
```

# Recursive descent with Recursive calls

```
void stmt() {
    switch ( lookahead ) {
        case expr:
            match(expr); match(';'); break;
        case if:
            match(if); match('('); match(expr); match(')'); stmt();
            break;
        case for:
            match(for); match('(');
            optexpr(); match(';'); optexpr(); match(';'); optexpr();
            match(')'); stmt(); break;
        case other:
            match(other); break;
        default:
            report("syntax error");
    }
}
```

# Recursive descent with Recursive calls

```
void optexpr() {  
    if ( lookahead == expr ) match(expr);  
}
```

```
void match(terminal t) {  
    if ( lookahead == t ) lookahead = nextTerminal;  
    else report("syntax error");  
}
```