

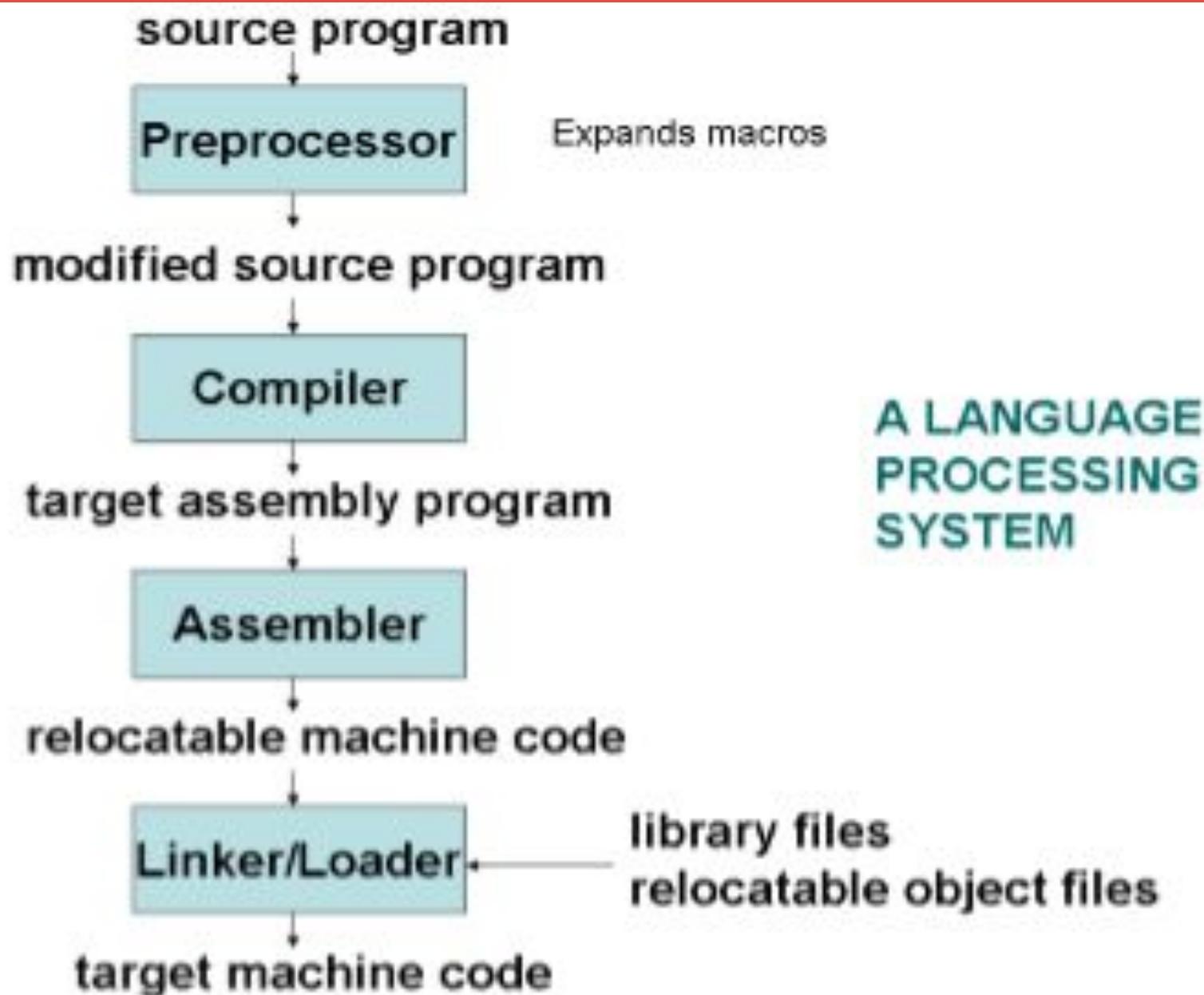
Module 5

Compilers: Analysis Phase

Syllabus

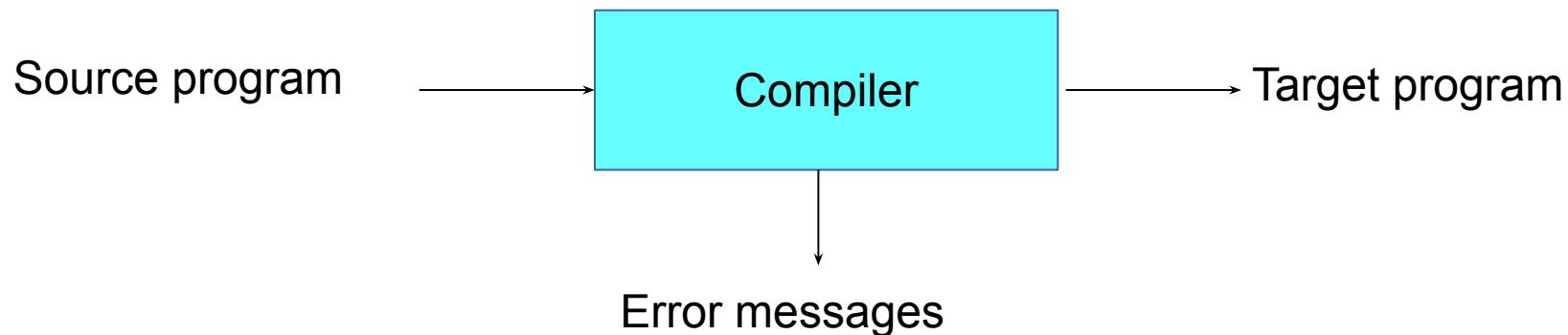
- Introduction to compilers, Phases of compilers:
 - **Lexical Analysis** -Role of Finite State Automata in Lexical Analysis, Design of Lexical analyzer, data structures used.
 - **Syntax Analysis**-Role of Context Free Grammar in Syntax analysis, Types of Parsers: Top down parser-LL(1), Bottom up parser-SR Parser, Operator precedence parser, SLR.
 - **Semantic Analysis**- Syntax directed definitions.

Language Processing System



What is Compiler?

- A compiler is a language translator that converts a high-level program into low-level language program that can be easily understood by the computer



Functions of a Compiler

- Translation of source language into target language
- Reporting Errors and Warnings in the input source language
- Helps in debugging of the execution of code
- Generates extra profiling code to report the statistics on the time taken by specific functions in the input source language

Through profiling one can determine the parts in program code that are time consuming and needs to be re-written.

This helps make your program execution faster which is always desired.

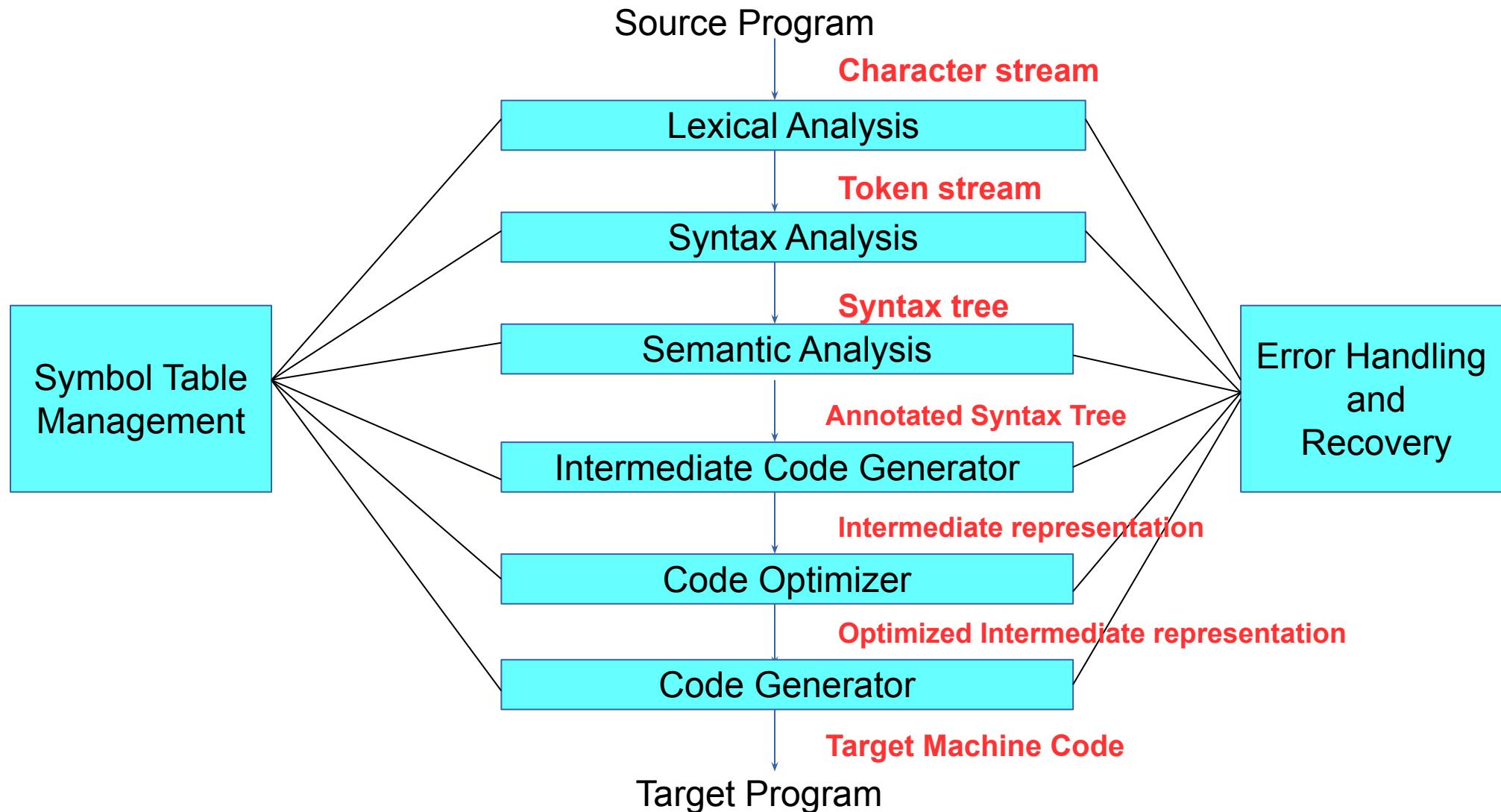
Analysis & Synthesis Model of Compilation

- **Analysis**-Breaking up of source program into constituent pieces and creation of intermediate representation
- **Synthesis**-Construction of desired target program from intermediate representation

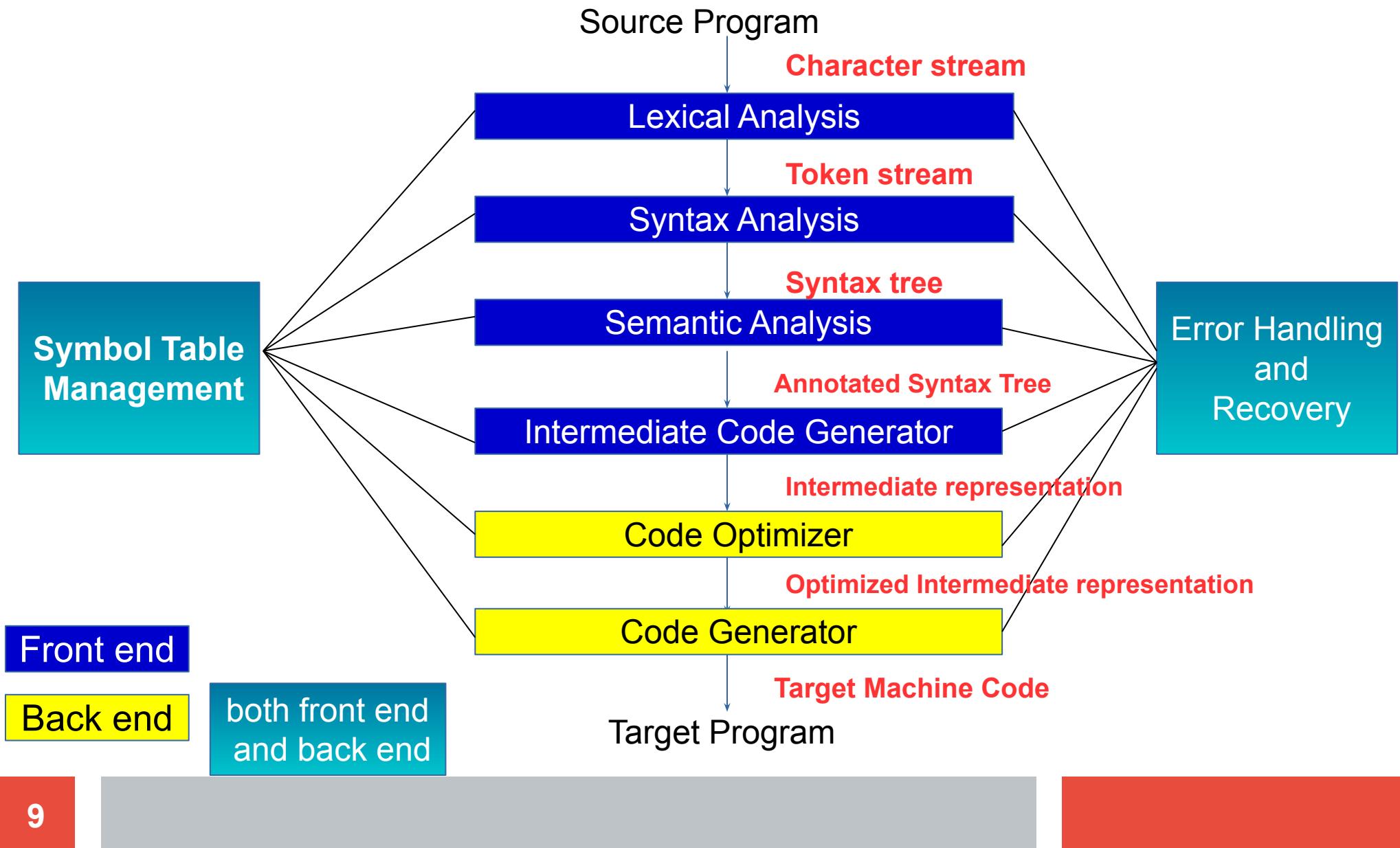
Phases of Compiler

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Intermediate code generation
- Code Optimization
- Code Generation
- Symbol table management
- Error handling and recovery

Phases of compiler



Phases of compiler-Front end-Back end



Lexical Analysis

- This is the 1st phase
- Lexical analysis is also called linear analysis or scanning, processes character from left to right & group the character stream into significant unit called **lexemes**
- These lexemes are mapped into tokens
- This phase does the additional job of removing extra white space, comments added by the user, etc. from the program

Syntax Analysis

- This is the 2nd phase
- The syntax analysis phase takes word/tokens from the lexical analyzer and **checks the syntactic correctness** of the input program
- Syntax analysis identifies and notifies the syntactic errors in the program once the program has been broken into the tokens
- It often referred as hierarchical analysis or **simply parsing**
- The hierarchical structure of the source program can be represented by a **parse tree**

Semantic Analysis

- This is the 3rd phase
- Once the program is syntactically correct means grammatically correct the next task is to check semantic correctness
- This phase performs the **checks on the meaning of the statement** and makes the necessary modification in the parse tree representation
- The word semantic refers to meaning & the semantic analyzer checks the meaning of the program
- This phase checks the source program for semantic error and gathers “type information” about the program element

Intermediate Code Generation

- This is the 4th phase
- This phase generates an **intermediate code** which helps to simplify the complexity of the code
- The intermediate representation should have two important properties:-
 - It should be easy to produce
 - It should be easy to translate into the target program

Code Optimization

- This is the 5th phase
- The code optimization phase attempts to **improve the intermediate code** so that a faster running machine code could be generated
- Code optimization is performed to:
 - Minimize the time taken to execute a program
 - Minimize the amount of memory occupied

Code Generation

- This is the 6th phase
- This phase is responsible for the **generation of target code** (machine language code)
 - Memory locations are selected for each variable
 - Instructions are translated into a sequence of assembly instructions
 - Variables and intermediate results are assigned to memory registers

Symbol Table Management

- Symbol table is a data structure holding information about all the symbols defined in the source program
- It is used as a reference table by all the phases of a compiler
- The typical information stored in the symbol table includes the name of the variables, their types, sizes relative offset within the program and so on
- The generation of this table is normally carried out by the lexical analyzer and syntax analyzer phases

Error Handling and recovery

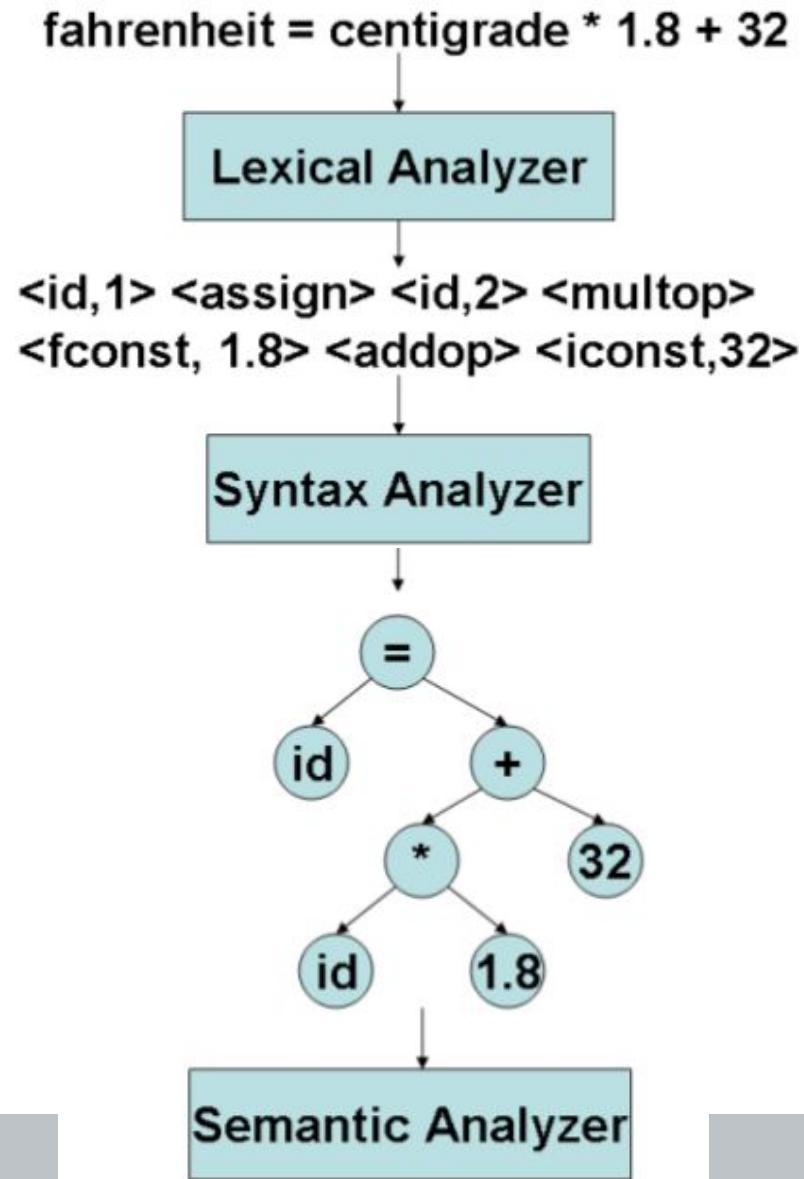
- Error handling is responsible for handling the error which can occur in any of the compilation phase
- After detecting the error, a phase must deal with that error so that compilation can proceed so as to detect more errors

Example

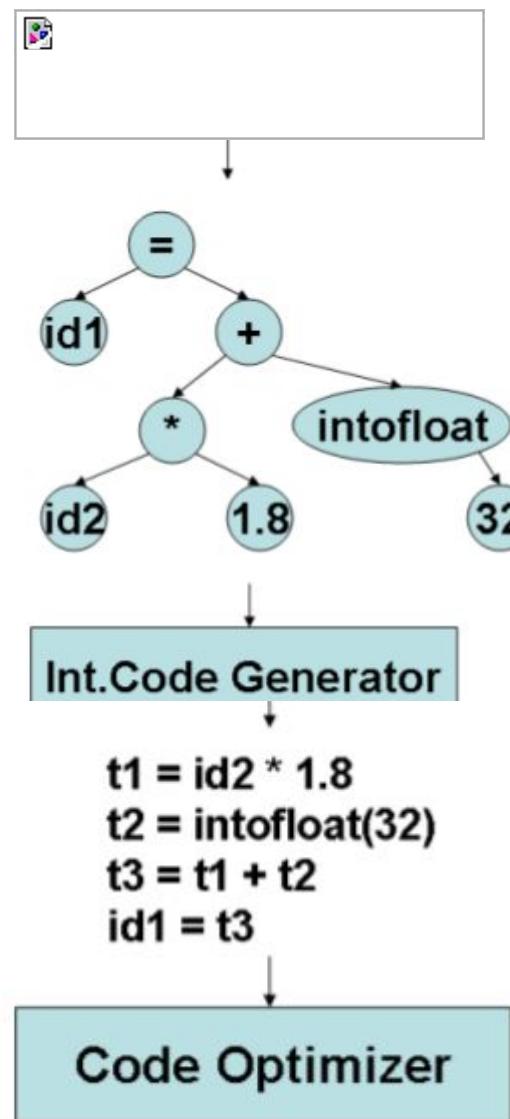
Q. Illustrate compilers internal representation of source program for each phase for the following statement:-

- fahrenheit=centigrade*1.8+32

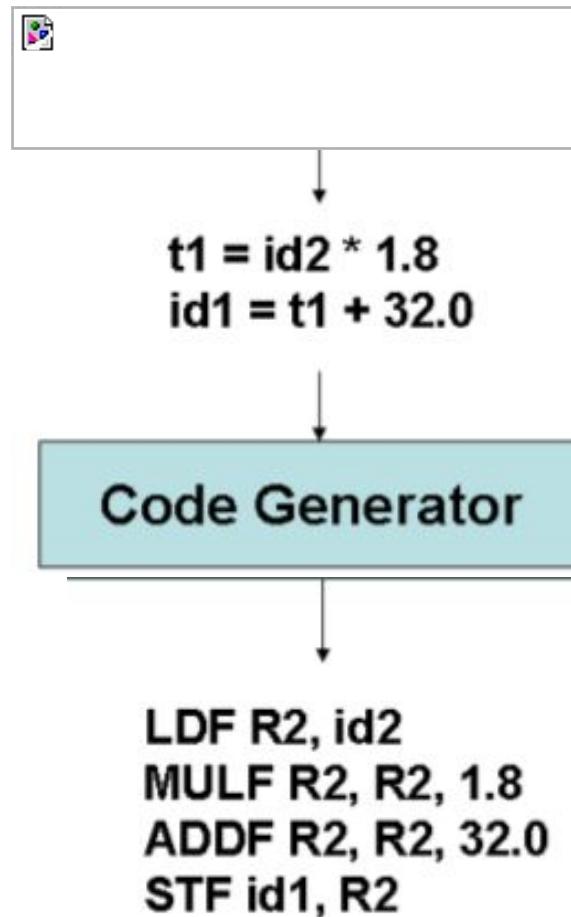
$\text{farenheit} = \text{centigrade} * 1.8 + 32$



$\text{farenheit} = \text{centigrade} * 1.8 + 32$



$\text{farenheit} = \text{centigrade} * 1.8 + 32$



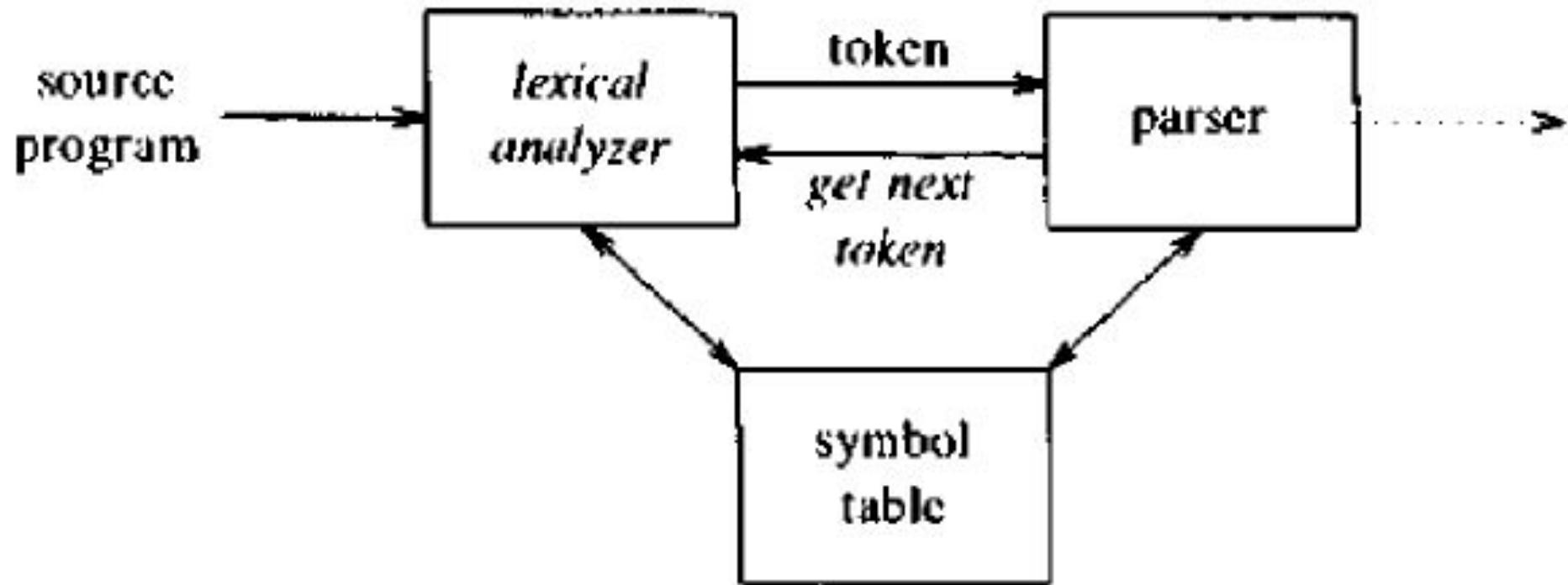
Comparison of compiler and Interpreter

Compilers	Interpreters
It converts the source program (higher level language) to target program (low level language which can be assembly language or machine language)	It converts <u>line by line</u> of the source program written in HLL into its equivalent machine code and executes if the program is error free
Speed of execution is fast	Speed of execution is slow
Translation is done <u>of the entire program</u>	Translation is <u>done line by line</u>
Program needs to be compiled only once and can be executed repeatedly	For every run of the program , the program needs to be translated

Role of Lexical Analyzer

- Lexical analyzer is the first phase whose main task is to read the input characters and produce as output a sequence of tokens
- Removal of comments and white spaces
- Correlating error messages from the compiler with the source program
- If source language supports some macro processor functions, then these preprocessor functions may also be implemented as lexical analysis takes place

Role of Lexical Analyzer



Role of Lexical Analyzer

Issues in Lexical Analysis

- Simpler design is the most important consideration
- Compiler efficiency is improved
- Compiler portability is enhanced

Tokens, patterns and Lexemes

Example: float abs_zero_Kelvin = -273;

Token- A string of characters which logically belong together
•e.g.-float, identifier, equal, minus, number, semicolon
•Tokens are treated as terminal symbols of the grammar specifying the source language

Pattern-The set of strings for which the same token is produced
•The pattern is said to match each string in the set
 float, l(l+d+_)*, =, -, d+, ;

Tokens, patterns and Lexemes

Example: float abs_zero_Kelvin = -273;

Lexeme- The sequence of characters matched by a pattern to form the corresponding token

“float”, “abs_zero_Kelvin”, “=”, “-”, “273”, “;”

Tokens, patterns and Lexemes

Examples of Token

Token	Sample Lexemes	Informal Description
const	const	const
if	if	if
relation	<, <=, =, <>, >, >=	< or <= or = or <> or > or >=
id	pi, count, D2	letter followed by letters and digits
num	3.1416, 0, 6.02E23	Any numeric constant
literal	“core dumped”	Any characters between “ and “ except “

Tokens, patterns and Lexemes

Attributes for Token

- ✓ The lexical analyzer collects information about tokens into their associated attributes
- ✓ Attributes help in translation of tokens
- ✓ A token has a single attribute – A pointer to the symbol table entry in which the information about the token is kept
- ✓ The pointer becomes attribute for the token

Tokens, patterns and Lexemes

Attributes for Token

Tokens, patterns and Lexemes

Tokens, patterns and Lexemes

Lexical Errors

- Lexical analyzer can't distinguish between misspelling of a keyword and undeclared function identifier

instead of 'if'
'fi'

if (a = -f(x))
then _____

fi (a = -f(x))
then _____

- It can't proceed further until the pattern for token matches a prefix of the remaining input

Tokens, patterns and Lexemes

Lexical Errors

Recovery Strategies

- Panic Mode Recovery- Delete successive characters from the remaining input until the lexical analyzer finds a well-formed token
- Deleting an extraneous character
- Inserting a missing character
- Replacing an incorrect character by a correct character
- Transposing two adjacent characters

Specification of Tokens

Strings and Languages

- Alphabet is any finite set of symbols.
- Letters, digits and punctuation
- $\{0,1\}$ – binary alphabet

- String over an alphabet is a finite sequence of symbols drawn from that alphabet.
- “Compiler” is a string of length eight.
- The empty string, denoted ϵ , is the string of length zero.

- Language is any countable set of strings over some fixed alphabet.
- Abstract languages are \emptyset , the empty set, or $\{\epsilon\}$

Specification of Tokens

String and Languages

TERM	DEFINITION
<i>prefix of s</i>	A string obtained by removing zero or more trailing symbols of string s ; e.g., <code>ban</code> is a prefix of <code>banana</code> .
<i>suffix of s</i>	A string formed by deleting zero or more of the leading symbols of s ; e.g., <code>nana</code> is a suffix of <code>banana</code> .
<i>substring of s</i>	A string obtained by deleting a prefix and a suffix from s ; e.g., <code>nan</code> is a substring of <code>banana</code> . Every prefix and every suffix of s is a substring of s , but not every substring of s is a prefix or a suffix of s . For every string s , both s and ϵ are prefixes, suffixes, and substrings of s .
<i>proper prefix, suffix, or substring of s</i>	Any nonempty string x that is, respectively, a prefix, suffix, or substring of s such that $s \neq x$.
<i>subsequence of s</i>	Any string formed by deleting zero or more not necessarily contiguous symbols from s ; e.g., <code>baaa</code> is a subsequence of <code>banana</code> .

Specification of Tokens

Operations on Languages

- Union
- Concatenation
- Closure
- Exponentiation

Specification of Tokens

Operations on Languages

- .Let L be the set {A, B,....Z,a,b,....z}and D be the set{0,1,...,9}
 - .L is the alphabet consisting of the set of upper and lower case letters
 - .D is the alphabet consisting of the set of the ten decimal digits
1. $L \cup D$ is the set of letters and digits.
 2. LD is the set of strings consisting of a letter followed by a digit.
 3. L^4 is the set of all four-letter strings.
 4. L^* is the set of all strings of letters, including ϵ , the empty string.
 5. $L(L \cup D)^*$ is the set of all strings of letters and digits beginning with a letter.
 6. D^+ is the set of all strings of one or more digits. □

Specification of Tokens

Operations on Languages

OPERATION	DEFINITION
<i>union of L and M</i> written $L \cup M$	$L \cup M = \{ s \mid s \text{ is in } L \text{ or } s \text{ is in } M \}$
<i>concatenation of L and M</i> written LM	$LM = \{ sr \mid s \text{ is in } L \text{ and } r \text{ is in } M \}$
<i>Kleene closure of L</i> written L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$ <p>L^* denotes "zero or more concatenations of" L.</p>
<i>positive closure of L</i> written L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$ <p>L^+ denotes "one or more concatenations of" L.</p>

Specification of Tokens

Regular Expressions



1. The regular expression $a|b$ denotes the set $\{a, b\}$.
2. The regular expression $(a|b)(a|b)$ denotes $\{aa, ab, ba, bb\}$, the set of all strings of a 's and b 's of length two. Another regular expression for this same set is $aa \mid ab \mid ba \mid bb$.
3. The regular expression a^* denotes the set of all strings of zero or more a 's, i.e., $\{\epsilon, a, aa, aaa, \dots\}$.
4. The regular expression $(a|b)^*$ denotes the set of all strings containing zero or more instances of an a or b , that is, the set of all strings of a 's and b 's. Another regular expression for this set is $(a^*b^*)^*$.
5. The regular expression $a \mid a^*b$ denotes the set containing the string a and all strings consisting of zero or more a 's followed by a b . □

If two regular expressions r and s denote the same language, we say r and s are *equivalent* and write $r = s$. For example, $(a|b) = (b|a)$.

Recognition of Tokens

REGULAR EXPRESSION	TOKEN	ATTRIBUTE-VALUE
ws	-	-
if	if	-
then	then	-
else	else	-
id	id	pointer to table entry
num	num	pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE



Recognition of Tokens

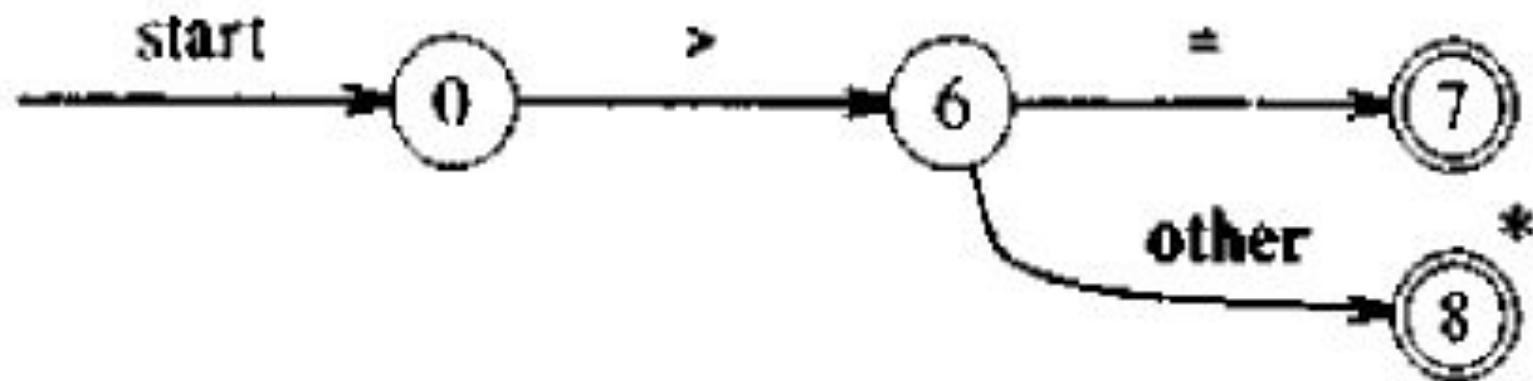
Transition Diagrams

- .We convert patterns into stylized flowcharts, called "**transition diagrams**"
- .Transition diagrams have a collection of nodes or circles, called **states**
- .Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns
- .**Edges** are directed from one state of the transition diagram to another
- .Each edge is labelled by a symbol or set of symbols

Recognition of Tokens

Transition Diagrams

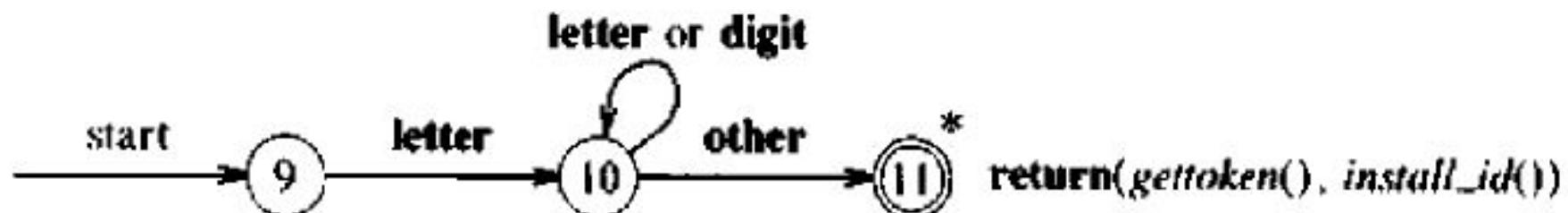
- Transition diagram for >=



Recognition of Tokens

Transition Diagrams

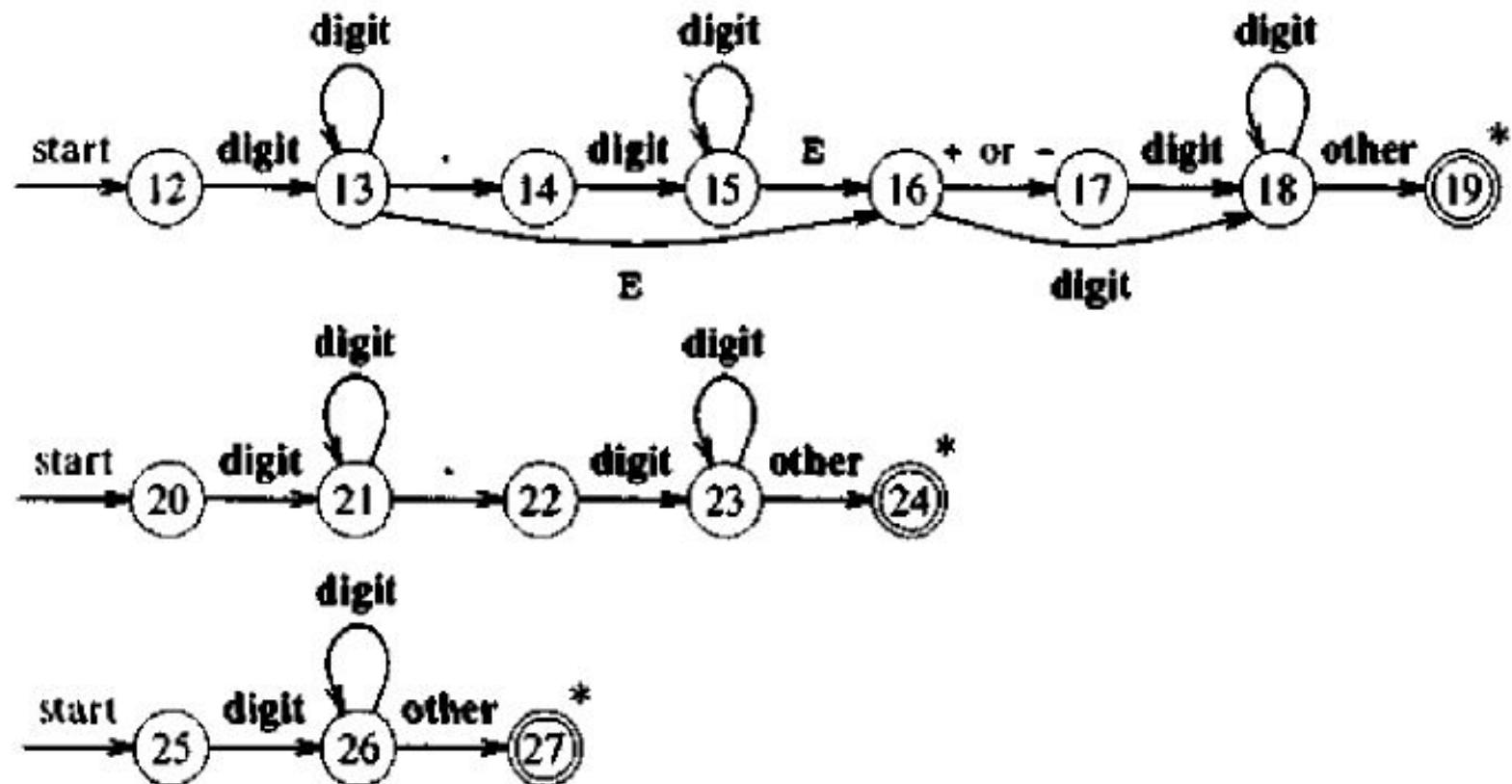
- Transition diagram for identifiers



Recognition of Tokens

Transition Diagrams

- Transition diagram for unsigned numbers



Recognition of Tokens

Transition Diagrams

- Transition diagram for white spaces



Recognition of Tokens

Implementing a Transition diagram

- Each state gets a segment of code
- If there are edges leaving a state, then the code reads a character and selects an edge to follow
- A function nextchar() is used to read the next character from the input buffer
- The forward pointer is advanced at each call and the character that is read is returned

Recognition of Tokens

Implementing a Transition diagram

- .If there is an edge labeled by the read character, then control is transferred to the code for the state pointed to by that edge
- .If there is no such edge, and current state is not one that indicates a token has been found then a routine fail() is invoked
- .This routine retracts the forward pointer to the position of the beginning pointer and initiates a search for a token specified by the next transition diagram

Recognition of Tokens

Implementing a Transition diagram

- .If there are no transition diagrams to try,then fail() calls an error recovery routine
- .To return tokens a global variable lexical_value
- .The pointers returned by functions install_id() and install_num() are assigned to the global variable when an identifier or number is found
- .The token class is returned by the main procedure of the lexical analyzer known as nexttoken()

Recognition of Tokens

C code for Lexical Analyzer

```
token nexttoken()
{
    while(1) {
        switch (state) {
            case 0:   c = nextchar();
                /* c is lookahead character */
                if (c==blank || c==tab || c==newline) {
                    state = 0;
                    lexeme_beginning++;
                    /* advance beginning of lexeme */
                }
                else if (c == '<') state = 1;
                else if (c == '=') state = 5;
                else if (c == '>') state = 6;
                else state = fail();
                break;

                .../* cases 1-8 here */

            case 9:   c = nextchar();
                if (isletter(c)) state = 10;
                else state = fail();
                break;
        }
    }
}
```

Recognition of Tokens

```
case 10:  c = nextchar();
    if (isletter(c)) state = 10;
    else if (isdigit(c)) state = 10;
    else state = 11;
    break;
case 11:  retract(1); install_id();
    return ( gettoken() );
... /* cases 12-24 here */

case 25:  c = nextchar();
    if (isdigit(c)) state = 26;
    else state = fail();
    break;
case 26:  c = nextchar();
    if (isdigit(c)) state = 26;
    else state = 27;
    break;
case 27:  retract(1); install_num();
    return ( NUM );
}
}
```

Recognition of Tokens

```
case 10:  c = nextchar();
    if (isletter(c)) state = 10;
    else if (isdigit(c)) state = 10;
    else state = 11;
    break;
case 11:  retract(1); install_id();
    return ( gettoken() );
... /* cases 12-24 here */

case 25:  c = nextchar();
    if (isdigit(c)) state = 26;
    else state = fail();
    break;
case 26:  c = nextchar();
    if (isdigit(c)) state = 26;
    else state = 27;
    break;
case 27:  retract(1); install_num();
    return ( NUM );
}
}
```

Module 5

Compilers: Analysis Phase

Role of Lexical Analyzer

- Lexical analyzer is the first phase whose main task is to *read the input characters and produce as output a sequence of tokens*
- *Removal of comments and white spaces*
- *Correlating error messages from the compiler with the source program*
- If source language supports some macro processor functions, then *these preprocessor functions may also be implemented* as lexical analysis takes place

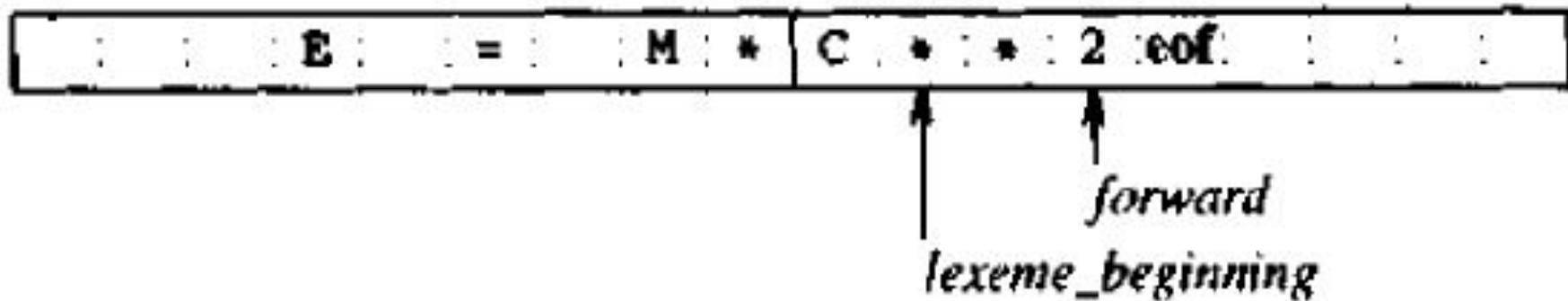
Input Buffering

- The Lexical Analyzer scans the characters of the source program *one at a time* to discover tokens
- A lot of time is consumed scanning the characters. So, specialized buffering techniques have been developed to *reduce the amount of overhead required* to process an input character

Input Buffering

Buffer Pairs

- Two pointers to the input are maintained:
- Pointer *lexeme_beginning* marks the beginning of the current lexeme, whose extent we are attempting to determine.
- Pointer *forward* scans ahead until a pattern match is found.



Input Buffering

Code to advance forward pointer

if *forward* at end of first half then begin

 reload second half;

forward:=forward + 1

end

else if *forward* at end of second half then begin

 reload first half;

 move *forward* to beginning of first half

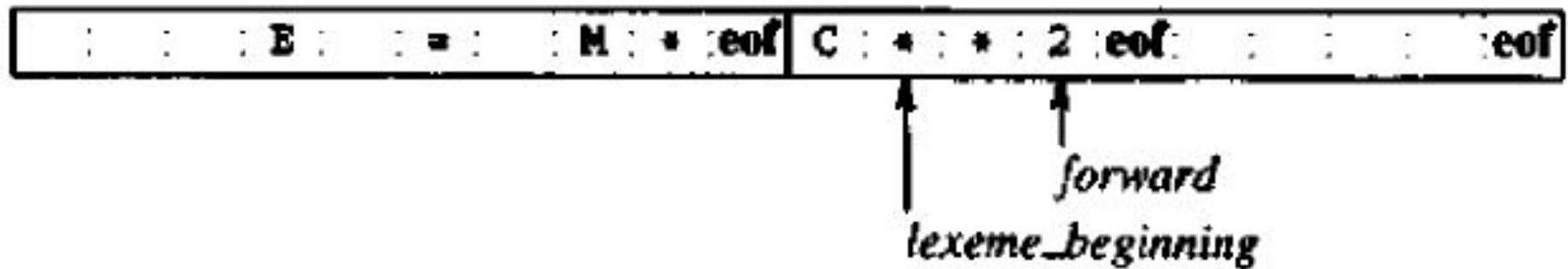
end

else *forward:=forward + 1;*

Input Buffering

Sentinels

- An extra key is inserted at the end of the array
- It is a special, dummy character that can't be part of source program and works as 'eof'



Input Buffering

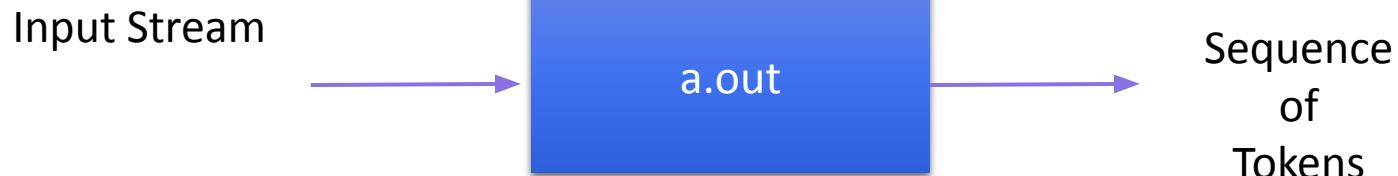
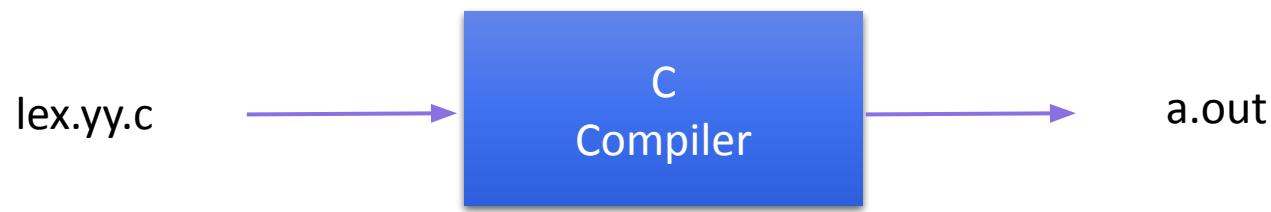
Lookahead code with sentinels

```
forward:=forward+1
if forward= eof then begin
    if forward at end of first half then begin
        reload second half;
        forward:=forward + 1
    end
    else if forward at end of second half then begin
        reload first half;
        move forward to beginning of first half
    end
    else /* eof within a buffer signifying end of input */
        terminate
end
```

Lexical Analyzer

- **Design of the Lexical Analyzer**
- **Data structures used**

Lex Compiler



Lex Specifications

Lex program consists of three parts:-

declarations

%%

translation rules

%%

auxiliary procedures

Lex specifications

Declarations

It consists of variables, constants, regular definitions

Lex specifications

Translational Rules are of the form:-

$$\begin{array}{ll} p_1 & \{ \text{action}_1 \} \\ p_2 & \{ \text{action}_2 \} \\ \dots & \dots \\ p_n & \{ \text{action}_n \} \end{array}$$

- .where each P_i is a regular expression and
- .each action_i is a program fragment describing what action the lexical analyzer should take when pattern p_i matches a lexeme

Lex specifications

Auxiliary Procedures

- This section holds whatever auxiliary functions are used in the actions
- These functions can be compiled separately and loaded with the lexical analyzer

Lex Program Example 1

```
%{
```

```
%}
```

```
%%
```

```
\n { printf("\n Hello Good Morning");
```

```
%%
```

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

It recognizes tokens from the input stream and returns them to the parser

Lex Program Example 2

```
%{  
char name[10];  
%}  
%%  
[\n] {printf("\n Hi.....%s.....Good Morning\n",name); return 1;}  
%%
```

```
void main()  
{  
    char opt; do  
{  
        printf("\nWhat is your name?"); scanf("%s",name);  
        yylex();  
        printf("\nPress y to continue"); scanf("%c",&opt);  
    }  
    while(opt=='y');
```

Lex Program Example 3

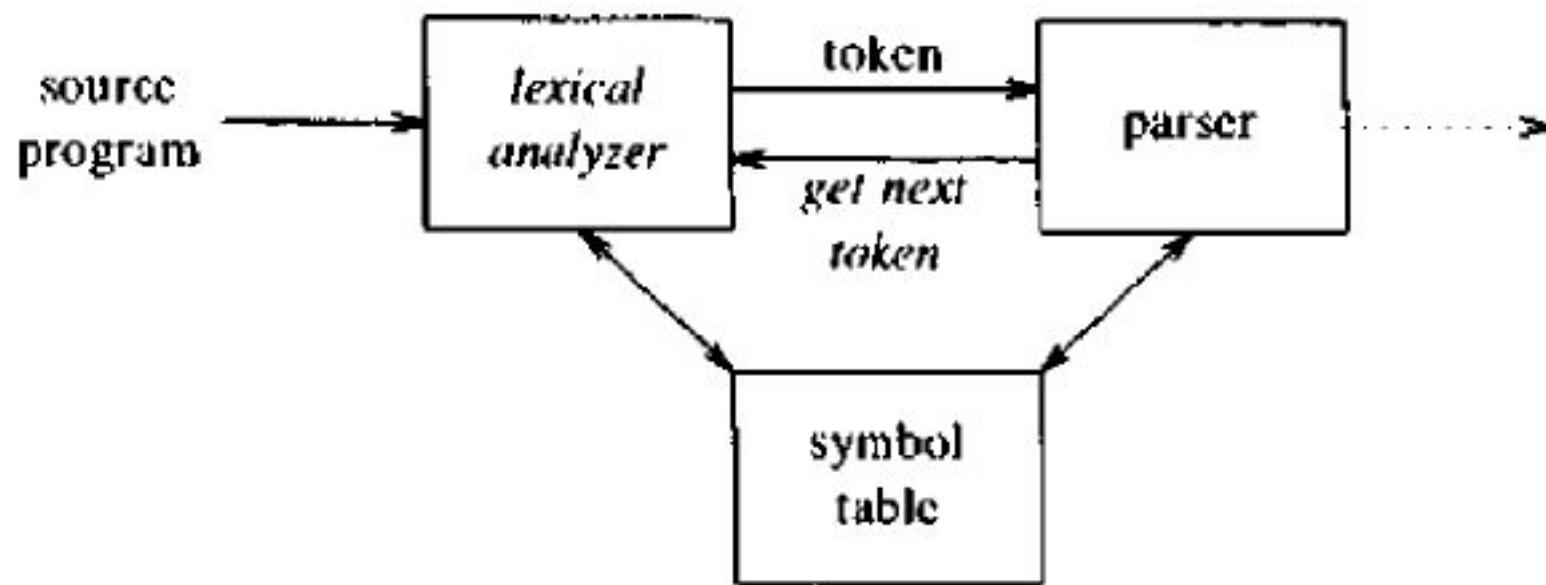
```
%{  
int linecount = 0, charcount = 0;  
%}  
  
%%  
. {charcount++;}  
\n {linecount++; charcount++;}  
%%  
  
void main()  
{  
    yylex();  
    printf("# of lines = %d, # of chars = %d", linecount, charcount);  
}
```

Lex Program Example 4

```
%{  
void display(int, char *);  
int flag;  
%}  
  
%%  
[a|e|i|o|u] {flag =1; display(flag,yytext);}  
. {flag =0; display(flag,yytext);}  
%%  
  
void main()  
{  
printf("\nEnter the word:");  
yylex();  
}
```

```
void display(int flag,char *t)  
{  
if(flag==1)  
{  
printf("\nThe given character %s is vowel\n",t);  
}  
else  
{  
printf("\nThe given character %s is not vowel\n",t);  
}  
}
```

Working of Lexical analyzer with parser



- When called by the parser, the lexical analyzer begins reading its remaining input, one character at a time, until it finds the longest prefix of the input that matches one of the patterns

Working of Lexical analyzer with parser

- .It then executes action, which returns the control to the parser
- .But if it does not then the lexical analyzer proceeds to find additional lexemes, until one of the corresponding actions causes a return to the parser.
- .The lexical analyzer returns a single quantity i.e the token to the parser
- .To pass an attribute value with information about the lexeme, a global variable is set known as **yylval**

Lex Program for tokens

```
%{  
    /* definitions of manifest constants  
    LT, LE, EQ, NE, GT, GE,  
    IF, THEN, ELSE, ID, NUMBER, RELOP */  
}  
  
/* regular definitions */  
delim      [ \t\n]  
ws         {delim}+  
letter     [A-Za-z]  
digit      [0-9]  
id          {letter}{(letter)|(digit)}*  
number     {(digit)*(\.(digit)*)? (E[+\-]? (digit)*)?}
```

Lex Program for tokens

```
install_id() {
    /* procedure to install the lexeme, whose
       first character is pointed to by yytext and
       whose length is yyleng, into the symbol table
       and return a pointer thereto */
}

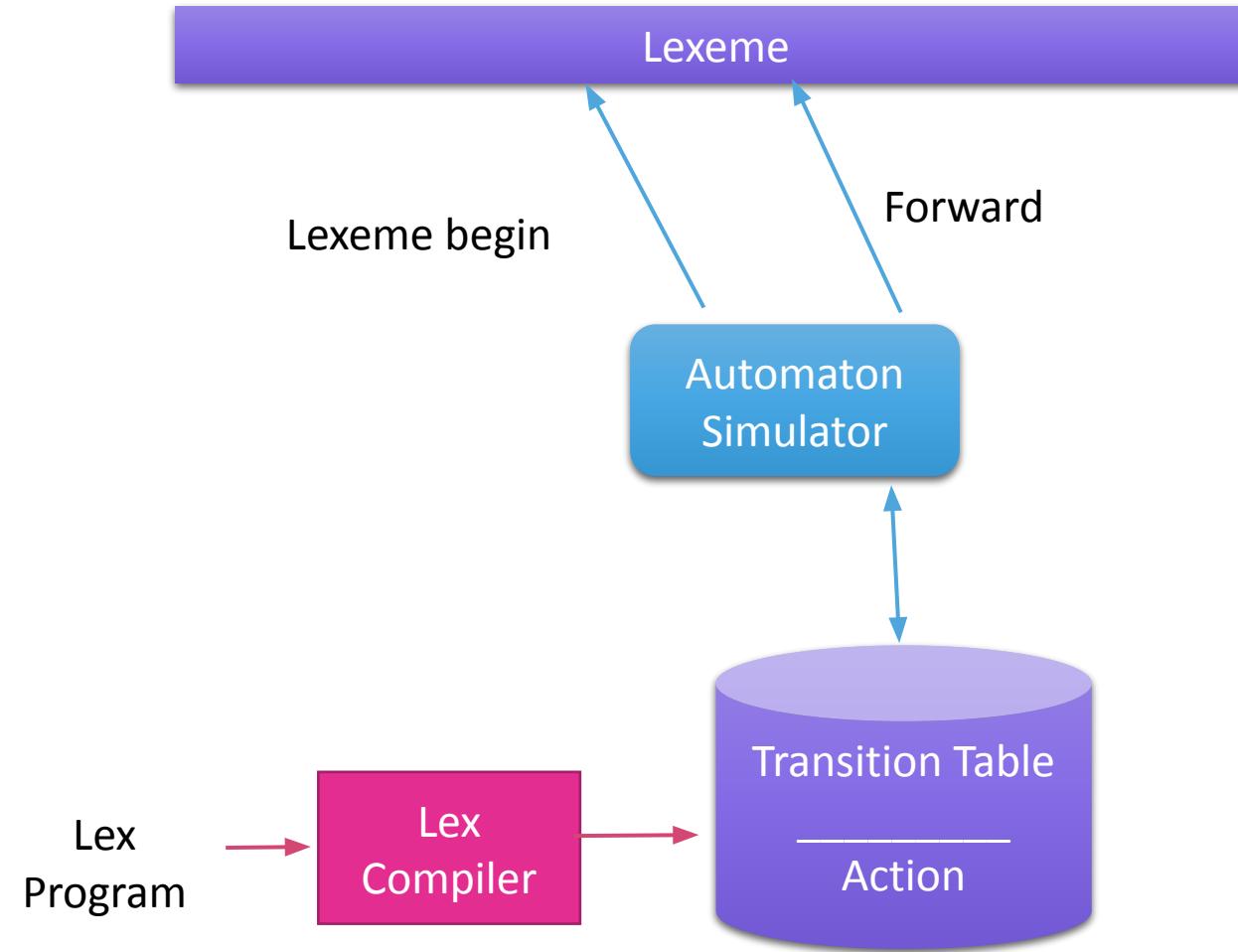
install_num() {
    /* similar procedure to install a lexeme that
       is a number */
}
```

Design of Lexical analyzer generator

Aim: Design a software tool that automatically constructs a lexical analyzer from a program in the Lex language

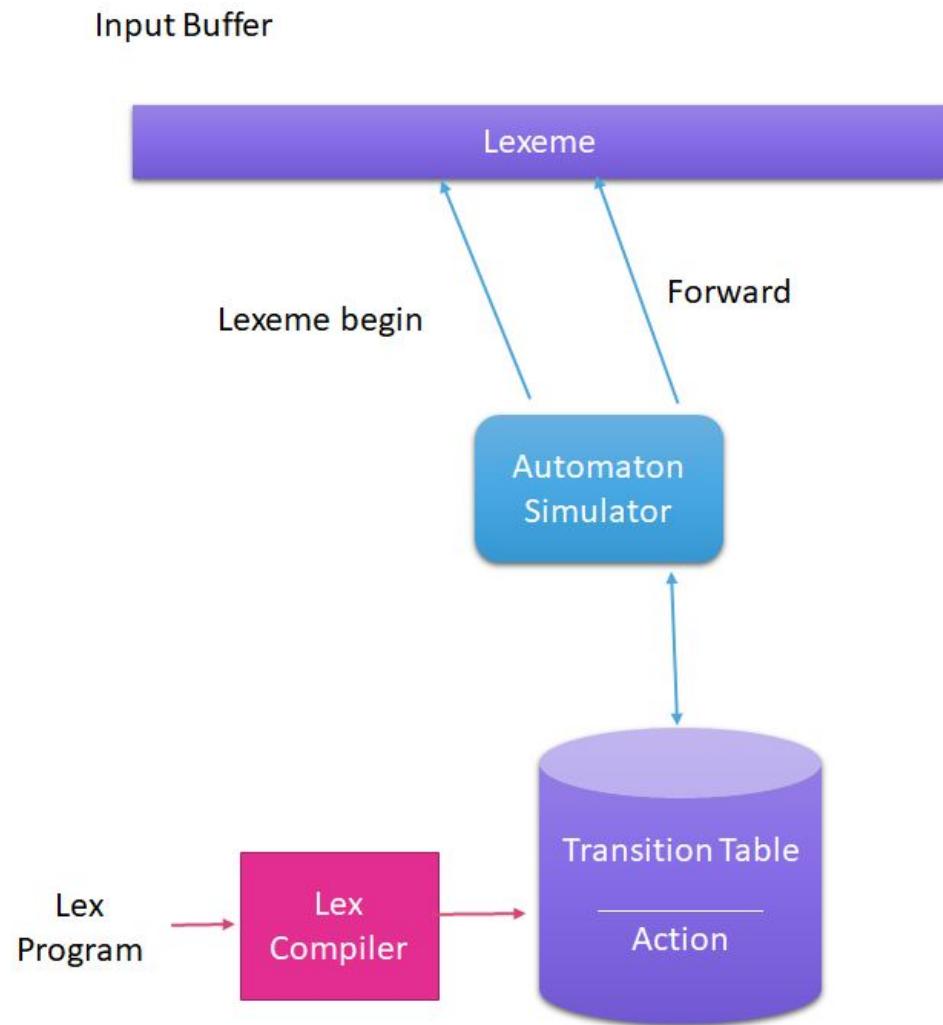
Architecture of a lexical analyzer generated by Lex

Input Buffer



Design of a Lexical analyzer

- The program that serves as the lexical analyzer includes a fixed program that simulates an automaton
- The rest of the lexical analyzer consists of components that are created from the Lex program by Lex itself
- A Lex program is turned into a transition table and actions which are used by a finite automaton simulator



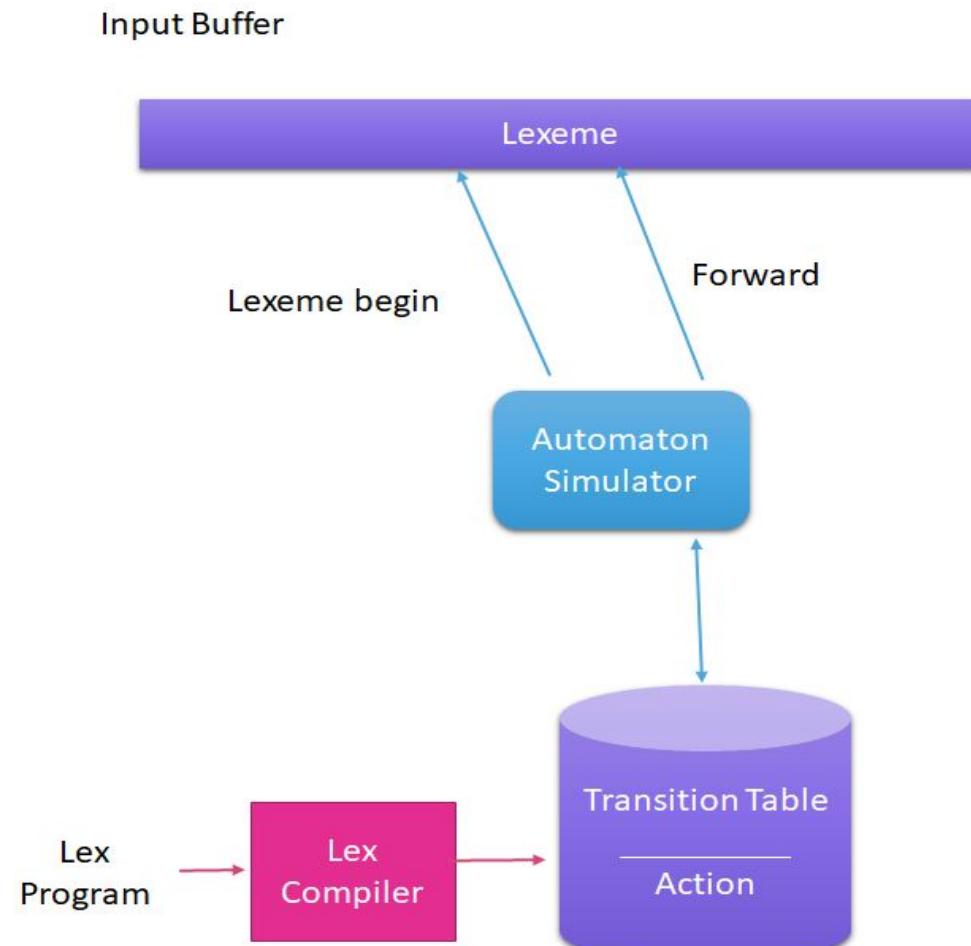
Design of a Lexical analyzer

The components are:

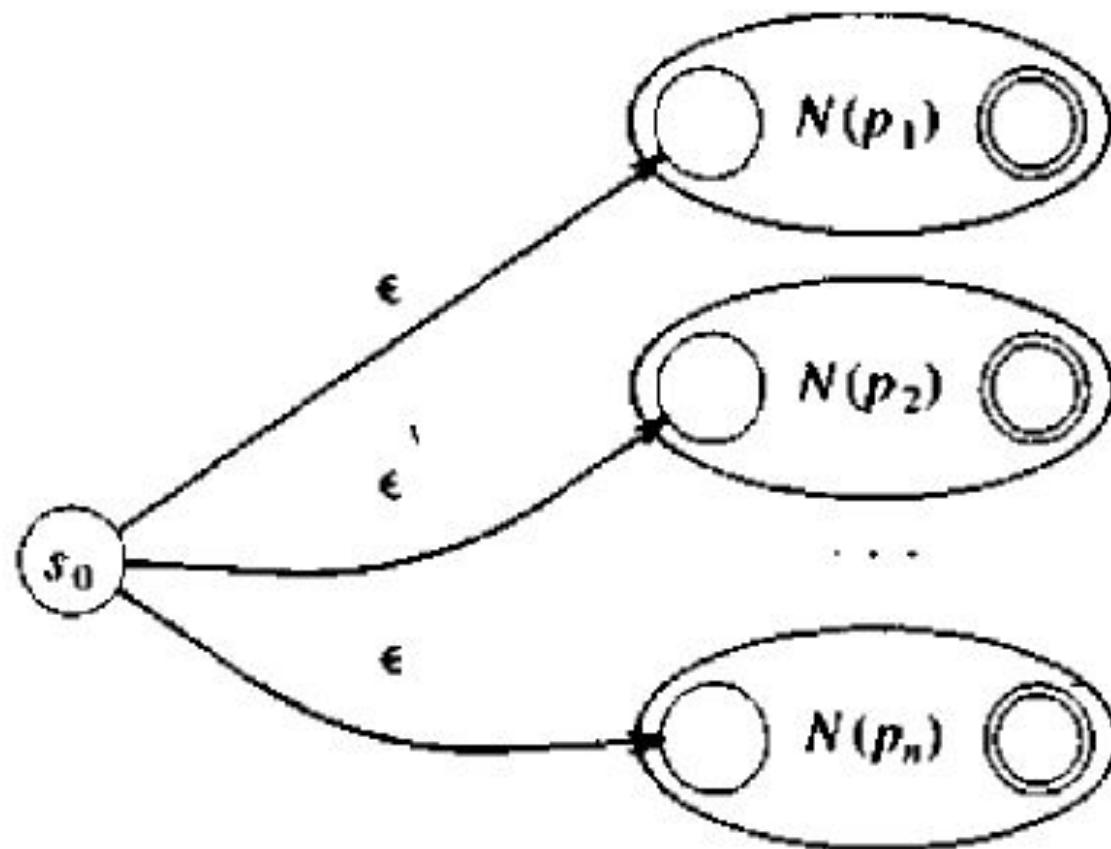
- A *transition table* for automaton
- Those *functions* that are passed directly through Lex to the output
- The *actions* from the input program which appear as fragments of code to be invoked at the appropriate time by the automaton stimulator

Design of a Lexical analyzer

- To construct the automaton take each regular expression pattern in the Lex program and convert it using Algorithm to an NFA
- We need a single automaton that will recognize lexemes matching any of the patterns in the program
- Hence combine all the NFA's into one by introducing a new start state with transitions to each of the start states of the NFA's N_i for pattern p_i



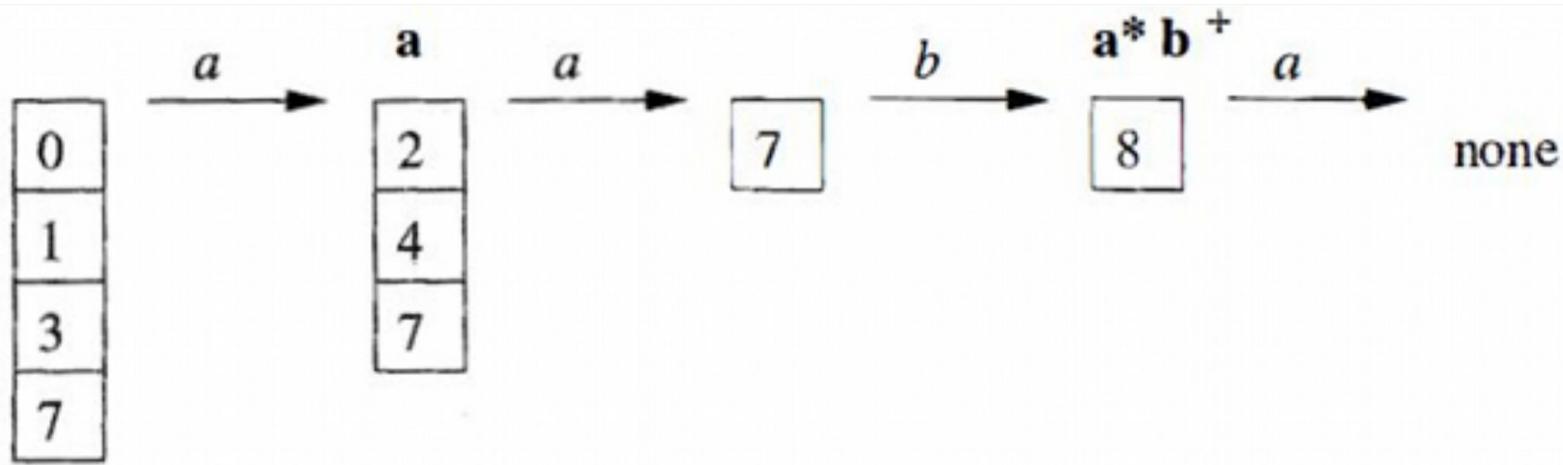
Pattern Matching Based on NFA's



Pattern Matching Based on NFA's

```
a    { } /* actions are omitted here */
abb  { }
a*b+ { }
```

Example



Sequence of sets of states entered when processing input *aaba*

Data structure used in Lexical Analyzer

- *Symbol*: Identifier used in the Source Program
- *Examples*: Names of variables, functions and Procedures
- *Symbol Table*: To maintain information about attributes of symbol

Operations on Symbol Table:

- Add a symbol and its attributes
- Locate a symbol's entry
- Delete a symbol's entry
- Access a symbol's entry

Data structure used in Lexical Analyzer

Design Goal of Symbol Table

1. The Table's organization should facilitate efficient search
2. Table should be compact (Less Memory)

Data structure used in Lexical Analyzer

To improve search efficiency, allocate more memory to symbol table

Organization of Entries:

1. Linear Data Structure
2. Non-Linear Data Structure

Data structure used in Lexical Analyzer

Symbol Table Entry Format

- Number of Fields to accommodate attributes of one symbol
- Symbol Field: The symbol to be stored
- Key Field: Basis for the search in table
- Entry of Symbol is called record
- Each entry can be of type fixed length, variable length or hybrid

Data structure used in Lexical Analyzer

Module 5

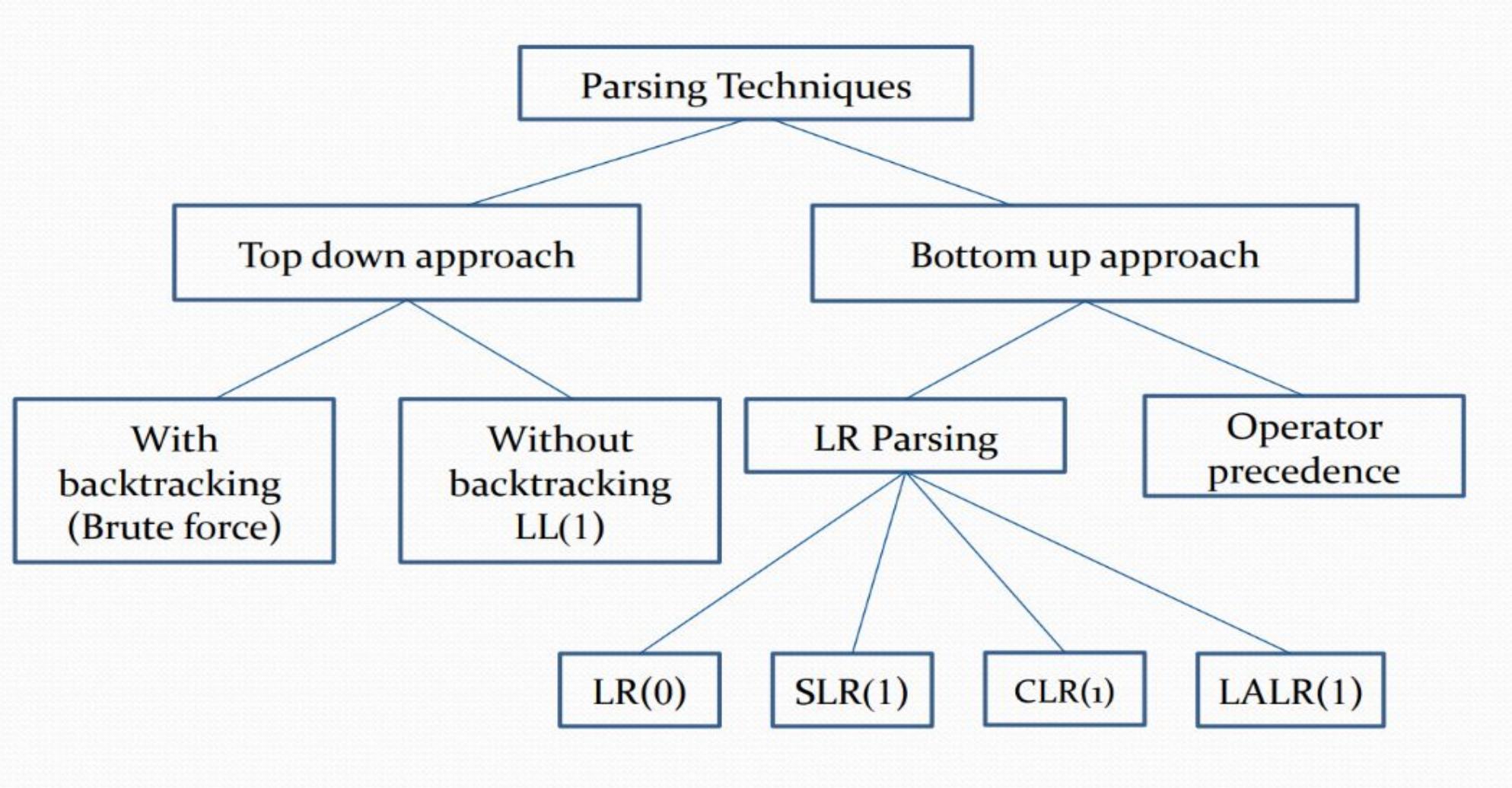
Compilers: Analysis Phase

Syntax Analysis

Syntax Analysis

- Role of Context Free Grammar in Syntax analysis
- Top-Down Parsing
 - LL1 Parsers
- Bottom-Up Parsing
 - Operator Precedence Parsing
 - SLR Parsers

Syntax Analysis



Syntax Analysis

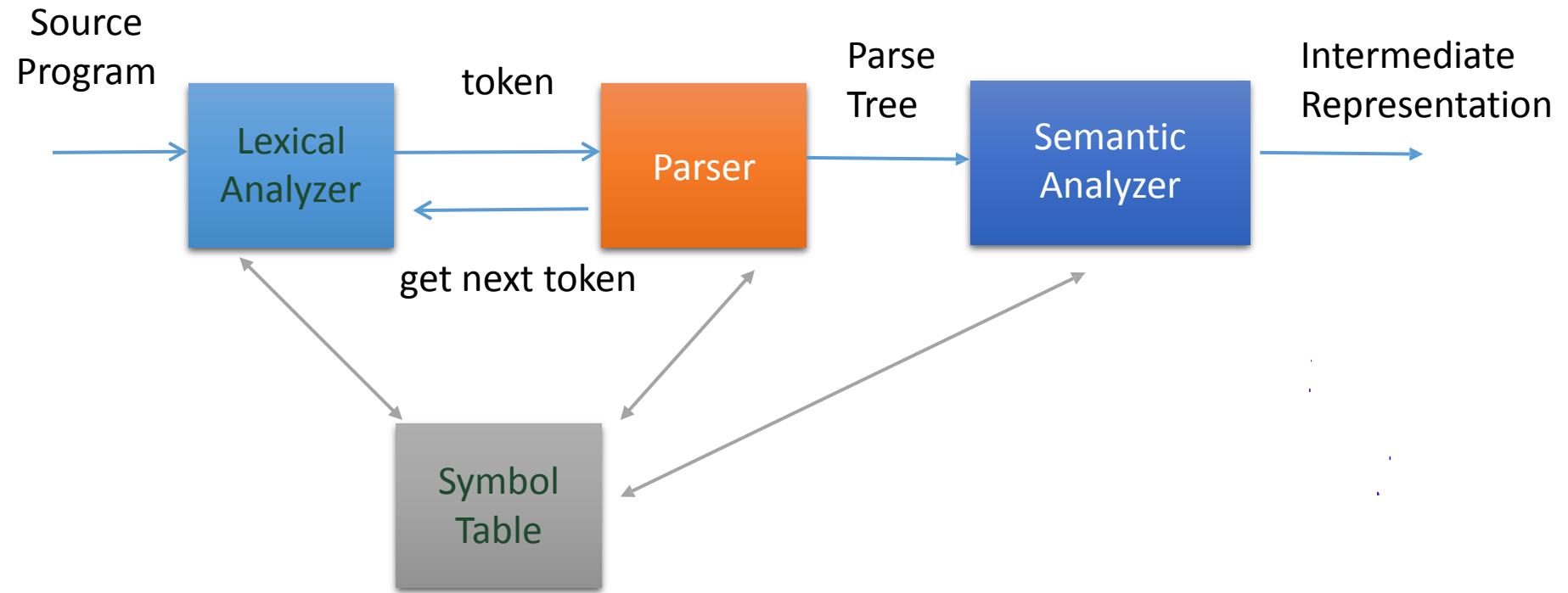
Top-down parser

- Recursive-Descent Parsing
 - Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
 - It is a general parsing technique, but not widely used.
 - Not efficient
- Predictive Parsing
 - no backtracking
 - efficient
 - needs a special form of grammars (LL(1) grammars).
 - Recursive Predictive Parsing is a special form of Recursive Descent parsing without backtracking.
 - Non-Recursive (Table Driven) Predictive Parser is also known as LL(1) parser

Syntax Analysis

- Syntax analysis is the second phase of the compiler
- It is also called as parsing and it generates parse tree
- Parser:
 - It is the program that takes *tokens and grammar* (context-free grammar -CFG) as input and validates the input token against the grammar

Syntax Analysis



The Role of Parser

- It obtains a string of tokens from the lexical analyzer and *verifies* that the string can be generated by the grammar for the source language
- It also reports any *syntax errors*
- It also *recovers* from commonly occurring errors so that it can continue processing the remaining input

Syntax Error Handling

- .Lexical errors: such as misspelling a keyword.
- .Syntactic errors: such as an arithmetic expression with unbalanced parentheses.
- .Semantic errors: such as an operator applied to an incompatible operand.
- .Logical errors: such as an infinitely recursive call.

Error Recovery Techniques

- The main error recovery strategies are as follows:
- Panic Mode recovery
- Phrase Level
- Error Production
- Global Correction

Panic Mode Recovery

- Panic mode error recovery is based on the idea of *discarding input symbols one at a time* until one of the designated set of synchronized tokens is found
- The synchronizing tokens are usually delimiters, such as semicolon or end, whose role in the source program is clear
- It has the advantage of simplicity and does not go into an infinite loop.
- When multiple errors in the same statement are rare, this method is quite useful.

Phrase Level Recovery

- On discovering error, a parser may perform a *local fix* to allow the parser to continue and simultaneously report error
- In phrase level recovery mode each empty entry in the parsing table is filled with a *pointer to a specific error routine* to take care of that error
- These error routine may be:
 - Change , insert, or delete input symbol
 - Issue an appropriate error message
 - Pop items from the stack

Error Production

- Error production *adds rules to the grammar* that describes the erroneous syntax . This strategy can resolve many , but not all potential errors
- It includes production for common errors and we can augment the grammar for the production rules that generates the erroneous constructs
- A parser constructed from an augmented grammar by these error productions detects the anticipated error production is used during parsing
- Since it is almost impossible to know all the errors that can be made by the programmers , this method is not practical

Global Correction

- Replace *incorrect input with input that is correct* and require the fewer changes to create
- This requires expensive techniques that are costly in terms of time and space
- The algorithm states that:
 - For a given grammar G give an incorrect input string X
 - Now find a parse tree for a related string y with the help of an algorithm such that the number of insertions , deletion and changes of token require to change x into y is as small as possible

Context Free Grammar

(CFG)

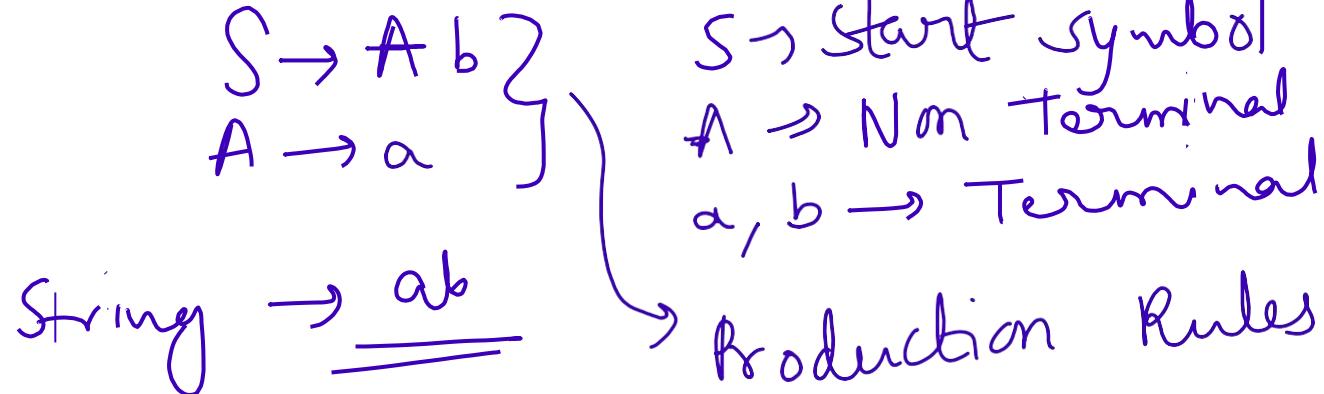
CFG is used to define the syntactic structure of a programming language. It also contains a set of rules called as production rules or productions.

Terminals, Non Terminals, Start symbol, Productions.

Context Free Grammar

- A context free grammar has four tuple $\langle V, T, P, S \rangle$ where,
 - V : set of non terminal symbols for writing the grammar
 - T : Set of terminal symbol T , used as token for the language
 - P : Set of production rule
 - S : A special non terminal which is start symbol

Context Free Grammar



string : abb

$S \rightarrow A B b$
 $A \rightarrow a$
 $B \rightarrow b$

$S \rightarrow \underbrace{A}_{a} \underbrace{B b}_{a b} b$

Example

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

To generate a valid string: **- (id + id * id)**

Example

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

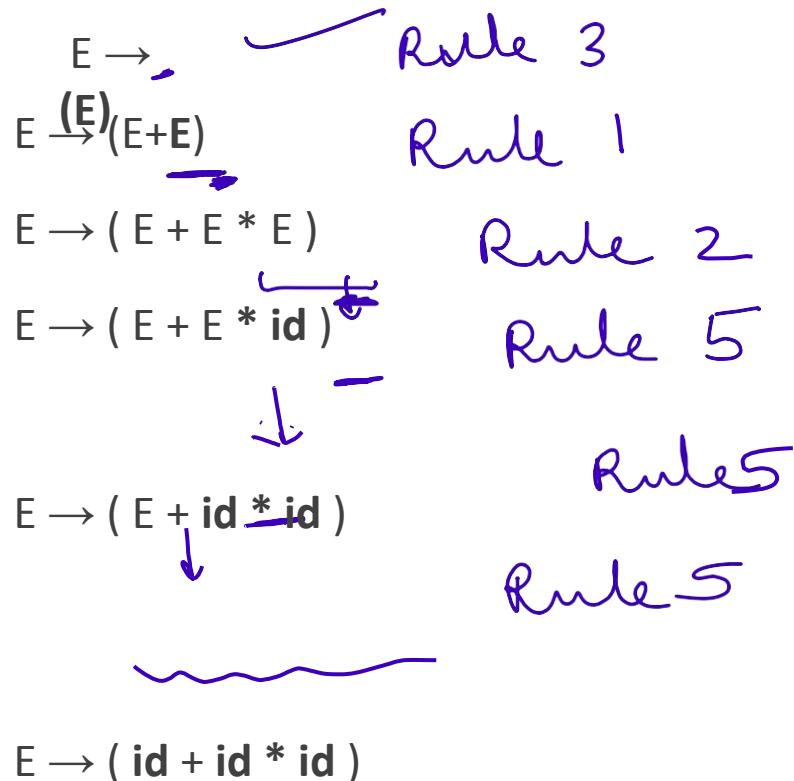
To generate a valid string: $- (id + id * id)$

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

Example

$\bar{E} \rightarrow \text{Start Symbol}$, $\text{id} \rightarrow \text{Terminal}$

$E \rightarrow E + E \mid E * E \mid (E) \mid - E \mid \text{id}$



Production Rules

- 1) $E \rightarrow E + E$
- 2) $E \rightarrow E * E$
- 3) $E \rightarrow (E)$
- 4) $E \rightarrow - E$
- 5) $E \rightarrow \text{id}$

Example

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

To generate a valid string: $- (id + id * id)$

$$E \rightarrow (E) \quad \text{Rule 3}$$

$$E \rightarrow (E * E) \quad \text{Rule 2}$$

$$E \rightarrow (E + E * E) \quad \text{Rule 1}$$

$$E \rightarrow (E + E * id) \quad \text{Rule 5}$$

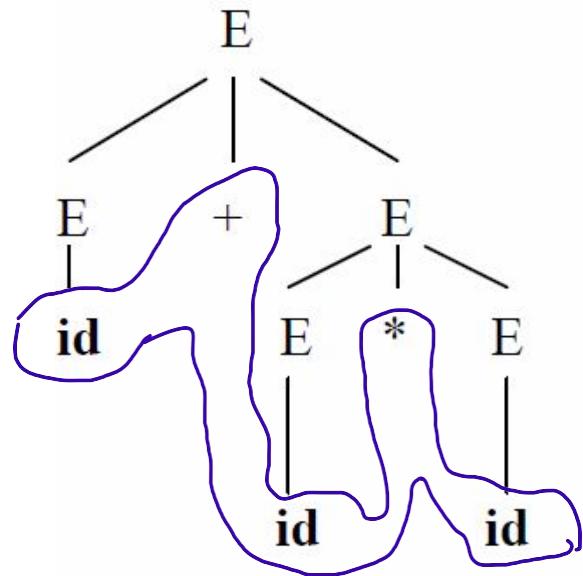
$$E \rightarrow (E + id * id) \quad \text{Rule 5}$$

$$E \rightarrow (id + id * id) \quad \text{Rule 5}$$

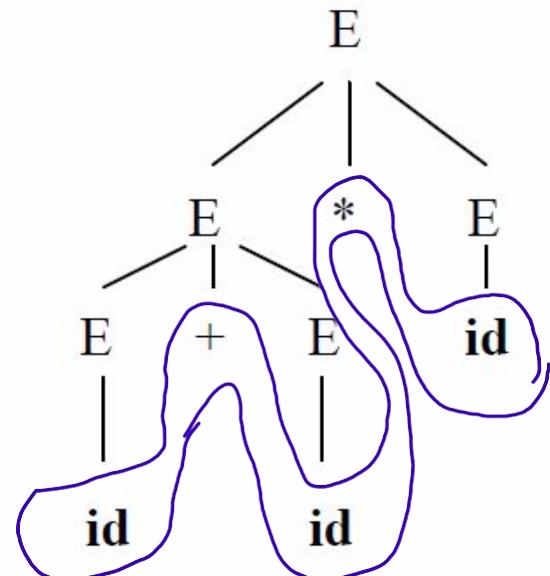
)

Example

Syntax Tree



$\text{id} + \text{id} * \text{id}$



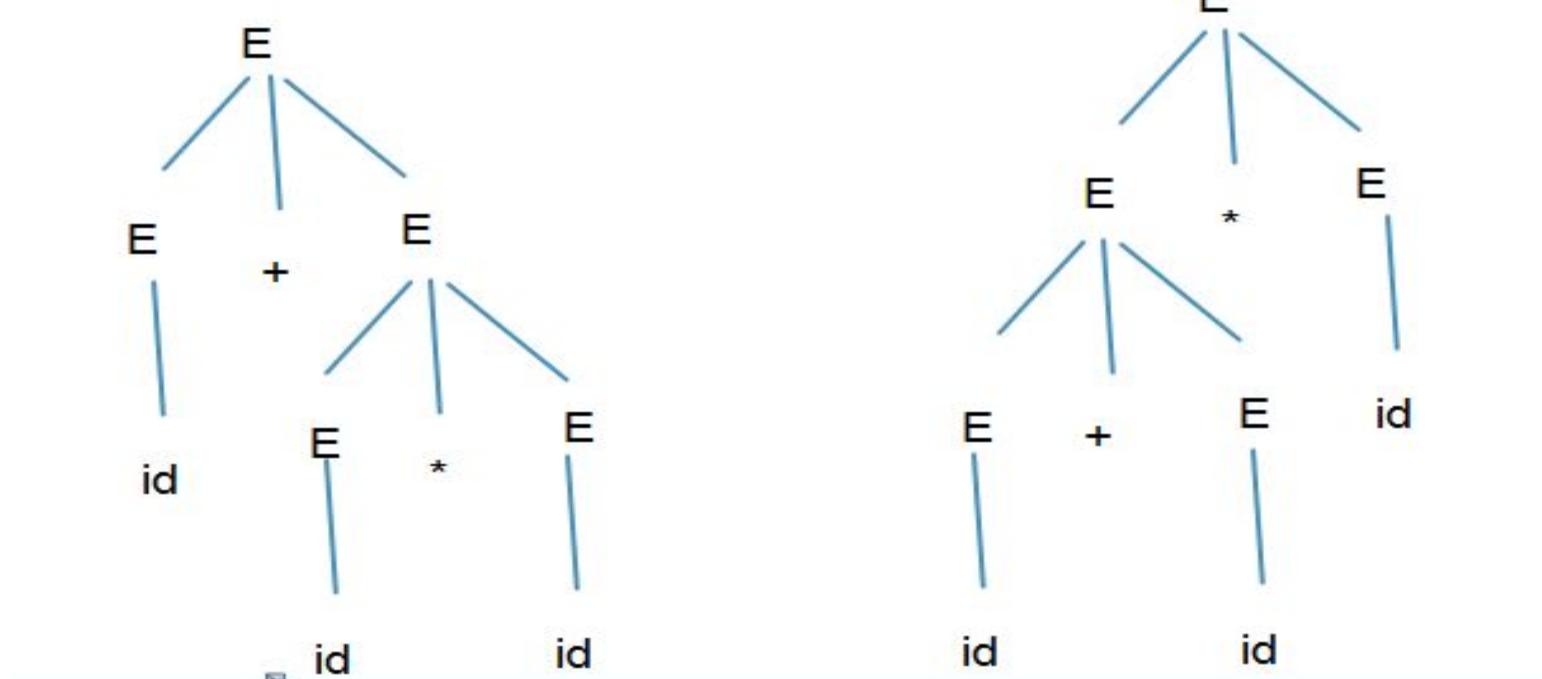
$\text{id} + \text{id} * \text{id}$

CFG

$$E \rightarrow E + E \mid E * E \mid (E) \mid - E \mid \text{id}$$

To generate a valid string: **- id + id * id**

Parse Trees



Ambiguous and Unambiguous Grammar

Ambiguous Grammar

For a given grammar, we can generate at least one string which can be presented using more than one parse tree then such grammar is called ambiguous grammar

Ambiguous and Unambiguous Grammar

Unambiguous Grammar

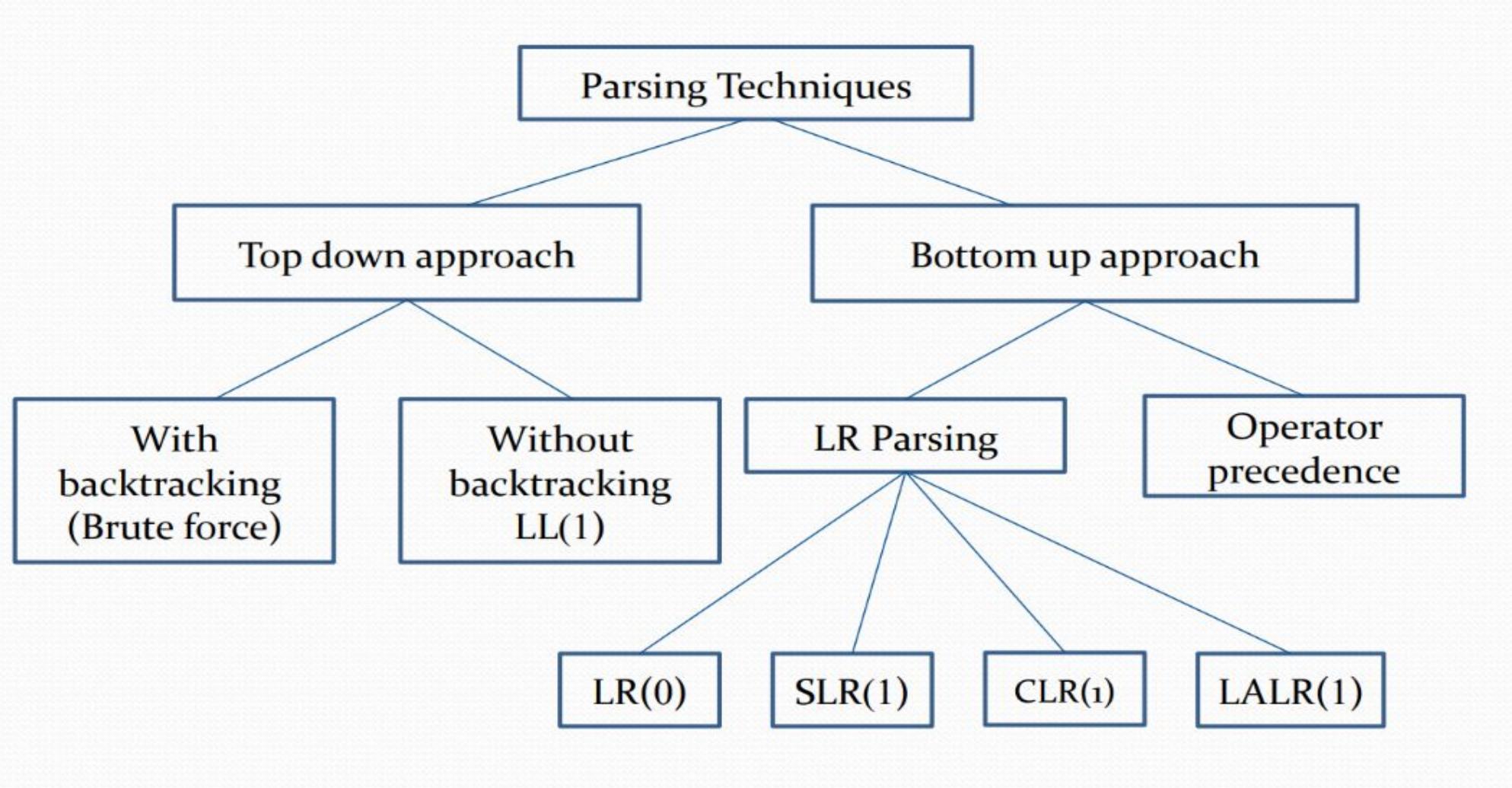
For a given grammar, all possible strings which can be generated using it have only one representation of Parse Tree, such grammar is called Unambiguous Grammar

Module 5

Compilers: Analysis Phase

Syntax Analysis

Syntax Analysis



Ambiguous and Unambiguous Grammar

Ambiguous Grammar

For a given grammar, we can generate at least one string which can be presented using more than one parse tree then such grammar is called ambiguous grammar

Ambiguous and Unambiguous Grammar

Unambiguous Grammar

For a given grammar, all possible strings which can be generated using it have only one representation of Parse Tree, such grammar is called Unambiguous Grammar

Eliminating Ambiguity

- Eliminating Left Recursion
- Left Factoring

Eliminating Left Recursion

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

$$A \rightarrow A \alpha \text{ for some string } \alpha.$$

Top-down parsing methods cannot handle left-recursive grammars.

Hence, left recursion can be eliminated as follows:

Left Recursion: $A \rightarrow A \alpha \mid \beta$

Then,

$$A \rightarrow \beta A'$$

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

Eliminating Left Recursion

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

for $i := 1$ to n do begin

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

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

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

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

 end

end

Eliminating Left Recursion

Example 1:

$S \rightarrow A$

$A \rightarrow Ad \mid Ae \mid aB \mid ac$

$B \rightarrow bBc \mid f$

Solution: The grammar after eliminating left recursion is-

$S \rightarrow A$

$A \rightarrow aBA' \mid acA'$

$A' \rightarrow dA' \mid eA' \mid \epsilon$

$B \rightarrow bBc \mid f$

Eliminating Left Recursion

Example 2:

$$A \rightarrow ABd \mid Aa \mid a$$

$$B \rightarrow Be \mid b$$

Solution- The grammar after eliminating left recursion is-

$$A \rightarrow aA'$$

$$A' \rightarrow BdA' \mid aA' \mid \in$$

$$B \rightarrow bB'$$

$$B' \rightarrow eB' \mid \in$$

Eliminating Ambiguity

Example 3:

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T^* F \mid F$$
$$F \rightarrow (E) \mid \text{id}$$

Eliminating Left Recursion

First eliminate the left recursion for E as

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

$$E \rightarrow T E'$$

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

Then eliminate for T as

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

$$T \rightarrow F T'$$

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

Eliminating Left Recursion

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Left Factoring

- .Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.
- .When it is not clear which of two alternative productions to use to expand a non-terminal A, we can rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

If there is any production $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$, then the left factored grammar is:

$$A \rightarrow \alpha A'$$

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

Left Factoring

Example 1:

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

Solution: Step-01:

$$S \rightarrow aS' \mid b$$

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

Again, this is a grammar with common prefixes

Step-02:

$$S \rightarrow aS' \mid b$$

$$S' \rightarrow SA' \mid bb$$

$$A' \rightarrow SbS \mid aSb$$

Left Factoring

Example 2:

$S \rightarrow a \mid ab \mid abc \mid abcd$

Solution.- Step-1:

$S \rightarrow aS'$

$S' \rightarrow b \mid bc \mid bcd \mid \in$

Again, this is a grammar with common prefixes

Left Factoring

Example 2: contd.

Step-02:

$$S \rightarrow aS'$$

$$S' \rightarrow bA' \mid \in$$

$$A' \rightarrow c \mid cd \mid \in$$

Again, this is a grammar with common

Step-03:

$$S \rightarrow aS'$$

$$S' \rightarrow bA' \mid \in$$

$$A' \rightarrow cB' \mid \in$$

$$B' \rightarrow d \mid \in$$

LL(1) Grammar

Used to construct Predictive Parser

- Predictive Parser – Recursive Descent Parser with no need of Backtracking
- First 'L' - scanning input from Left to Right
- Second 'L' - Leftmost Derivation
- "1" - One input symbol of Look ahead at each step to make parsing action decisions
- Left Recursive and Ambiguous grammar is NOT LL(1)

FIRST ()

FIRST()

- FIRST (α) is set of terminal symbol that are the first symbol appearing in R.H.S in derivation of α
- If $\alpha \rightarrow \in$ then \in is also in FIRST (α)
 - If the terminal symbol a then $\text{FIRST}(a) = \{a\}$
 - If there is a rule $X \rightarrow \in$ then $\text{FIRST}(X) = \{\in\}$
 - For the rule $A \rightarrow X_1 X_2 X_3 X_4 \dots X_k$ $\text{FIRST}(A) = (\text{FIRST}(X_1) \cup \text{FIRST}(X_2) \cup \text{FIRST}(X_3) \dots \text{FIRST}(X_k))$
 - Where $k \leq n$ such that $1 \leq j \leq k - 1$

Follow ()

- FOLLOW(A) is defined as the set of terminal symbols that appear immediately to the right of A
- In other words

$\text{FOLLOW}(A)=\{a \mid S \rightarrow^* Aa \beta \text{ where } \alpha \text{ and } \beta \text{ are some grammar symbols may be terminal or non terminal}$

- The rules for computing FOLLOW function are as follows
 - For the start symbol S place \$ in FOLLOW(S)
 - If there is a production $A \rightarrow \alpha B \beta$ then everything in FIRST(β) without \in is to be placed in FOLLOW(B)
 - If there is a production $A \rightarrow \alpha B \beta$ or $A \rightarrow \alpha B$ and the $\text{FIRST}(\beta)=\{\in\}$ then $\text{FOLLOW}(A)=\text{FOLLOW}(B)$ or $\text{FOLLOW}(B)=\text{FOLLOW}(A)$
 - That means everything in FOLLOW(A) is in FOLLOW(B)

Syntax analysis

First & Follow

FIRST()

- FIRST (α) is set of terminal symbol that are the first symbol appearing in R.H.S in derivation of α
- If $\alpha \rightarrow \in$ then \in is also in FIRST (α)
 - If the terminal symbol a then $\text{FIRST}(a) = \{a\}$
 - If there is a rule $X \rightarrow \in$ then $\text{FIRST}(X) = \{\in\}$
 - For the rule $A \rightarrow X_1 X_2 X_3 X_4 \dots X_k$ $\text{FIRST}(A) = (\text{FIRST}(X_1) \cup \text{FIRST}(X_2) \cup \text{FIRST}(X_3) \dots \text{FIRST}(X_k))$
 - Where $k \leq n$ such that $1 \leq j \leq k - 1$

FIRST of Grammar

Example 1

- $A \rightarrow BC$
- $B \rightarrow Ax \mid x$
- $C \rightarrow yC \mid y$

Solution

- In $A \rightarrow BC$
 $\text{FIRST}(A) = \{\text{FIRST}(B) \cup \text{FIRST}(C)\}$ if $B \rightarrow \epsilon$ is true
- $\text{FIRST}(A) = \{\text{FIRST}(B)\}$ if $B \rightarrow \epsilon$ is false
- $\text{FIRST}(A) = \{x\}$
- $\text{FIRST}(B) = \{x\}$
- $\text{FIRST}(C) = \{y\}$

FIRST of Grammar

Example 2:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

Solution:

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

FOLLOW()

- FOLLOW(A) is defined as the set of terminal symbols that appear immediately to the right of A
- In other words

$\text{FOLLOW}(A) = \{a \mid S \rightarrow^* Aa\beta \text{ where } \alpha \text{ and } \beta \text{ are some grammar symbols}\}$
may be terminal or non terminal

- The rules for computing FOLLOW function are as follows
 - For the start symbol S place \$ in FOLLOW(S)
 - If there is a production $A \rightarrow \alpha B \beta$ then everything in FIRST(β) without \in is to be placed in FOLLOW(B)
 - If there is a production $A \rightarrow \alpha B \beta$ or $A \rightarrow \alpha B$ and the $\text{FIRST}(\beta) = \{\in\}$ then $\text{FOLLOW}(A) = \text{FOLLOW}(B)$ or $\text{FOLLOW}(B) = \text{FOLLOW}(A)$
 - That means everything in FOLLOW(A) is in FOLLOW(B)

Follow of Grammar

Rule 1:

Place \$ in FOLLOW (S) where S is the start symbol and \$ is the input right endmarker

Rule 2:

If there is a production $A \rightarrow \alpha B \beta$ then everything in FIRST(β) except ϵ is in FOLLOW (B)

Rule 3:

If there is a production $A \rightarrow \alpha B$ or
a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains ϵ
then everything in FOLLOW (A) is in FOLLOW (B)

APPLY ABOVE RULES UNTIL THERE IS NO UPDATION IN FOLLOW LIST

Follow of Grammar

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

Example 1:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

Solution:

1. FOLLOW (E) = { \$,) }

Since E is start symbol so \$

For Production Rule F → (E) First()) = {) }

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

By Rule 3: if E' contains ε then everything in FOLLOW(E') will be in FOLLOW(E) i.e FOLLOW (E') = FOLLOW (E)

Follow of Grammar

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

Example 1:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

Everything in First(E) except ϵ

Solution:

$$3. FOLLOW(T) = \{ FIRST(E') \}$$

$$= \{ +, \text{Follow}(E) \}$$

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

$$4. FOLLOW(T') = FOLLOW(T)$$

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

Substituting ϵ in place of E' , we get : $E \rightarrow T$, so Follow(E) needs to be obtained

Follow of Grammar

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

Example 1:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

Solution:

$$\begin{aligned}5. FOLLOW(F) &= \{ FIRST(T') \} \\&= \{ * , FOLLOW(T) \} \\&= \{ * , + , \$,) \}\end{aligned}$$

Everything in First(T') except ϵ

Substituting ϵ in place of T', we get : $T \rightarrow F$ so Follow(T) needs to be obtained

Follow of Grammar

	FIRST	FOLLOW
A	x	\$, x
B	x	y
C	y	\$, x

Example 2:

$$A \rightarrow BC$$

$$B \rightarrow Ax \mid x$$

$$C \rightarrow yC \mid y$$

Solution:

$$1. \text{ FOLLOW}(A) = \{ \$ \} \cup \text{FIRST}(x) = \{ \$, x \}$$

$$2. \text{ FOLLOW}(B) = \text{FIRST}(C)$$

$$= \{ y \}$$

$$3. \text{ FOLLOW}(C) = \text{FOLLOW}(A)$$

$$= \{ \$, x \}$$

Follow of Grammar

	FIRST	FOLLOW
S	d , g , h , ϵ , b, a	
A	d , g , h , ϵ	
B	g , ϵ	
C	h , ϵ	

Example 3:

$$S \rightarrow ACB \mid Cbb \mid Ba$$

$$A \rightarrow da \mid BC$$

$$B \rightarrow g \mid \epsilon$$

$$C \rightarrow h \mid \epsilon$$

Solution:

$$\begin{aligned}1. \text{ FIRST}(S) &= \{\text{FIRST}(A) \cup \text{FIRST}(C) \cup \text{FIRST}(B)\} \cup \text{FIRST}(b) \\&\quad \cup \text{FIRST}(a)\end{aligned}$$

$$\begin{aligned}&= \{ d, g, h, \epsilon \} \cup \{ h, \epsilon \} \cup \{ g, \epsilon \} \cup \{ b \} \cup \{ a \} \\&= \{ d, g, h, \epsilon, b, a \}\end{aligned}$$

$$\begin{aligned}2. \text{ FIRST}(A) &= \text{FIRST}(d) \cup \text{FIRST}(B) \cup \text{FIRST}(C) \\&= \{ d, g, h, \epsilon \}\end{aligned}$$

$$3. \text{ FIRST}(B) = \{ g, \epsilon \}$$

$$4. \text{ FIRST}(C) = \{ h, \epsilon \}$$

Follow of Grammar

	FIRST	FOLLOW
S	d , g , h , ϵ , b, a	\$
A	d , g , h , ϵ	h , g , \$
B	g , ϵ	a , h , g , \$
C	h , ϵ	b , h , g , \$

Example 3:

$$S \rightarrow ACB \mid Cbb \mid Ba$$

$$A \rightarrow da \mid BC$$

$$B \rightarrow g \mid \epsilon$$

$$C \rightarrow h \mid \epsilon$$

Solution:

1. FOLLOW (S) = { \$ } ... Since S is start symbol
2. FOLLOW (A) = { FIRST (C) - ϵ } U { FIRST (B) - ϵ } U FOLLOW (S)
= { h , g , \$ }
3. FOLLOW (B) = FOLLOW(S) U FIRST (a) U { FIRST (C) - ϵ } U FOLLOW (A)
= { a , h , g , \$ }
3. FOLLOW (C) = FIRST (b) U { FIRST(B) - ϵ } U FOLLOW (A)
= { b , h , g , \$ }

Follow of Grammar

	FIRST	FOLLOW
S	a , b , d , ϵ	
A	a , b , d , ϵ	
B	b , d , ϵ	
D	d , ϵ	

Example 4:

$$S \rightarrow ABD$$

$$A \rightarrow a \mid BSB$$

$$B \rightarrow b \mid D$$

$$D \rightarrow d \mid \epsilon$$

Solution:

1. FIRST (S) = FIRST (A)

$$= \{ a \} \cup \text{FIRST}(B)$$

$$= \{ a \} \cup \{ b \} \cup \text{FIRST}(D)$$

$$= \{ a , b , d , \epsilon \}$$

2. FIRST (A) = { a } U FIRST (B)

$$= \{ a \} \cup \{ b \} \cup \text{FIRST}(D)$$

$$= \{ a , b , d , \epsilon \}$$

3. FIRST (B) = { b } U FIRST (D)

$$= \{ b , d , \epsilon \}$$

4. FIRST (D) = { d , ϵ }

Follow of Grammar

	FIRST	FOLLOW
S	a , b , d , ϵ	b , d , \$
A	a , b , d , ϵ	b , d , \$
B	b , d , ϵ	a , b , d , \$
D	d , ϵ	a , b , d , \$

Example 4:

$$S \rightarrow ABD$$

$$A \rightarrow a \mid BSB$$

$$B \rightarrow b \mid D$$

$$D \rightarrow d \mid \epsilon$$

Solution:

1. FOLLOW (S) = { \$ } U { FIRST (B) - ϵ } U FOLLOW (A)
= { \$, b , d } U { FIRST (B) - ϵ } U { FIRST (D) - ϵ }
= { b , d , \$ }
2. FOLLOW (A) = { FIRST (B) - ϵ } U { FIRST (D) - ϵ }
= { b , d , \$ }
3. FOLLOW (B) = { FIRST (S) - ϵ } U FOLLOW (A)
= { a , b , d , \$ }
3. FOLLOW (D) = FOLLOW (B)
= { a , b , d , \$ }

SYNTAX ANALYSIS



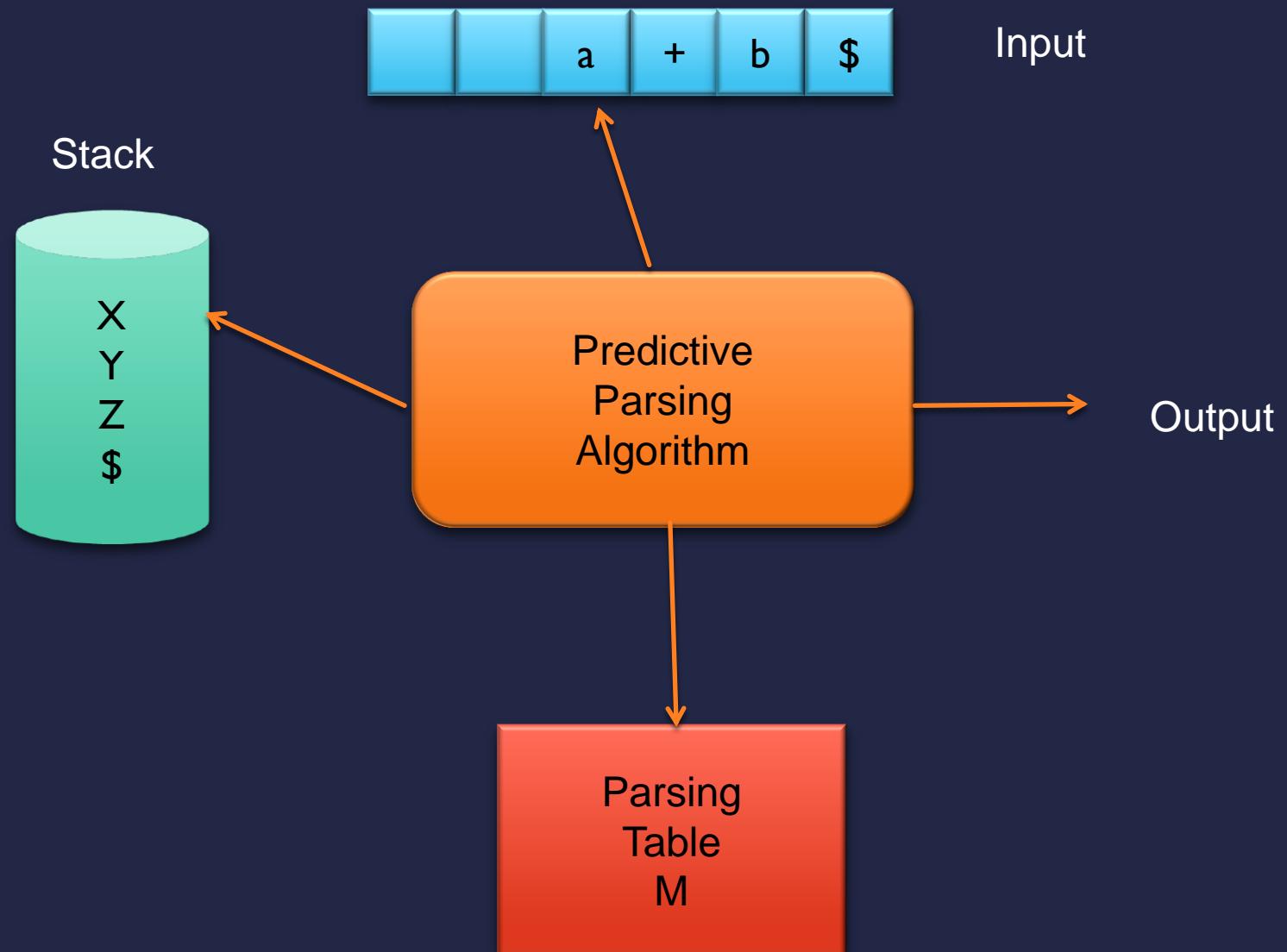
CONTENT

- LL (1) Parser
- Model of Non-Recursive Predictive Parser
- Construction of Predictive Parser Table
- Parsing a string

LL (1) GRAMMAR

- Used to construct Predictive Parser
- Predictive Parser – Recursive Descent Parser with no need of Backtracking
- First 'L' - scanning input from Left to Right
- Second 'L' - Leftmost Derivation
- "1" - One input symbol of Look ahead at each step to make parsing action decisions
- Left Recursive and Ambiguous grammar is NOT LL(1)

Model Of Non-Recursive Predictive Parser



Example:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

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

Construction Of Predictive Parsing Table

Input: Grammar G
Output: Parsing Table M

Table M has Non-Terminals as row and Terminals as columns

For Each Production $A \rightarrow \alpha$ of grammar do step 1 and 2

Step 1: For each terminal 'a' in FIRST (α)

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

Step 2:

Case 1:

If ϵ is in FIRST (α) then

for each terminal b in FOLLOW (A)

Add $A \rightarrow \alpha$ to M [A , b]

Case 2:

If ϵ is in FIRST (α) and \$ is in FOLLOW (A) then

for each terminal b in FOLLOW (A)

Add $A \rightarrow \alpha$ to M [A , b]

Step 3: Make each undefined entry of M be an error

Example:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

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

Consider Production $F \rightarrow id$

FIRST (id) = { id }

Add $F \rightarrow id$ to M [F, id]

Construction Of Predictive Parsing Table

- For every LL grammar each parsing table entry uniquely identifies a production or signals an error
- For some grammars however M may have some entries that are multiply defined
- Such grammars are not LL(1) Grammar

Predictive Parsing Algorithm

```
Let a be the first symbol of w
Let X be the top of the Stack symbol
while ( X != $)
{
    if ( X == a)
        pop the stack and let 'a' be the next symbol of w
    else if ( X is a terminal ) // X != a and X is terminal
        Error ( )
    else if ( M [ X , a ] is an error entry )
        Error ( )
    else if ( M [ X , a ] = Y1 Y2 ... Yk )
        Output the production X → Y1 Y2 ... Yk
        Pop the stack
        Push Yk Yk-1 ... Y1 onto the stack with Y1 on top
    Let X be the top stack symbol
}
```

Predictive Parsing Algorithm

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

Matched	Stack	Input	Action
	E \$	id + id * id \$	
	TE' \$	id + id * id \$	Output E → TE'
	FT'E' \$	id + id * id \$	Output T → FT'
	idT'E' \$	id + id * id \$	Output F → id
id	T'E' \$	+ id * id \$	match id
id	E' \$	+ id * id \$	Output T' → ε
id	+TE' \$	+ id * id \$	Output E' → +TE'

Predictive Parsing Algorithm

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

Matched	Stack	Input	Action
$id +$	$TE' \$$	$id * id \$$	match +
$id +$	$FT'E' \$$	$id * id \$$	Output $T \rightarrow FT'$
$id +$	$idT'E' \$$	$id * id \$$	Output $F \rightarrow id$
$id + id$	$T'E' \$$	$* id \$$	match id
$id + id$	$*FT'E' \$$	$* id \$$	Output $T' \rightarrow *FT'$
$id + id *$	$FT'E' \$$	$id \$$	match *
$id + id *$	$idT'E' \$$	$id \$$	Output $F \rightarrow id$

Predictive Parsing Algorithm

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

Matched	Stack	Input	Action
$id + id * id$	$T'E' \$$	$\$$	match id
$id + id * id$	$E' \$$	$\$$	Output $T' \rightarrow \epsilon$
$id + id * id$	$\$$	$\$$	Output $E' \rightarrow \epsilon$

SYNTAX ANALYSIS

BOTTOM-UP
PARSING



Parsing Techniques

Top down approach

With
backtracking
(Brute force)

Without
backtracking
 $LL(1)$

Bottom up approach

LR Parsing

Operator
precedence

LR(0)

SLR(1)

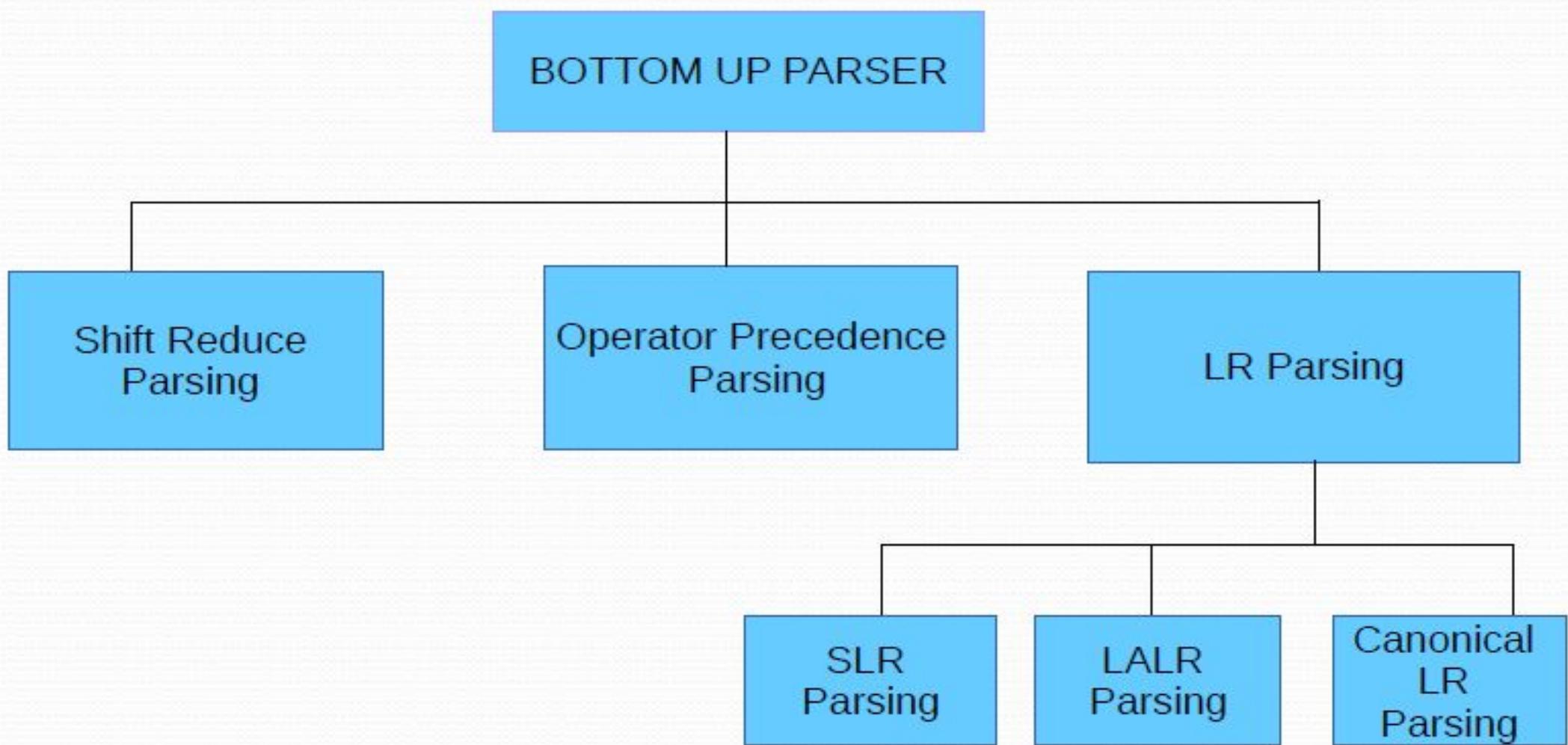
CLR(1)

LALR(1)

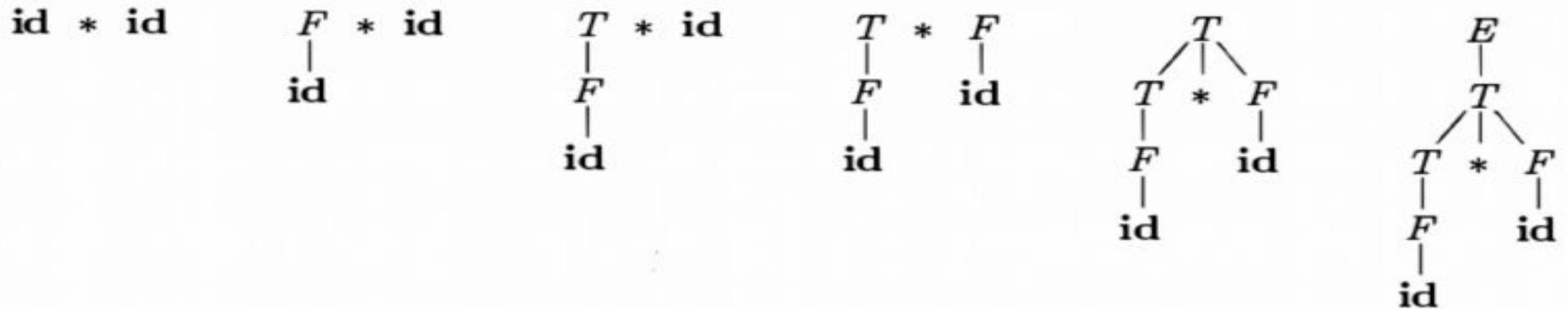
Bottom-Up Parsing

- Construction of Parse Tree for an input string beginning at leaves and working towards the root
- Can be visualized as reducing a string "w" to the start symbol of Grammar
- At each reduction step, a specific substring matching the body of production is replaced by Non-Terminal at the head of the production

Bottom-Up Parser



Bottom-Up Parser



Construction of Parse Tree for an input string beginning at leaves and working towards the root

Bottom-Up Parsing

To reduce the string and move towards root symbol

$$\begin{aligned} \text{id} * \text{id} &\Rightarrow F * \text{id} \\ &\Rightarrow T * \text{id} \\ &\Rightarrow T * F \\ &\Rightarrow T \\ &\Rightarrow E \end{aligned}$$

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

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

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

Now if we write above derivation from bottom to top, we get rightmost derivation.

Conclusion: Bottom-Up parsing during left to right scan of the input constructs a rightmost derivation in reverse

Handle Pruning

A **Handle** is a substring that matches the body of the production and whose reduction represents one step along the reverse of Rightmost Derivation

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid id \end{aligned}$$

Right Sentential Form	Handle	Reducing Production
$id1 * id2$	$id1$	$F \rightarrow id$
$F * id2$	F	$T \rightarrow F$
$T * id2$	$id2$	$F \rightarrow id$
$T * F$	$T * F$	$T \rightarrow T * F$
T	T	$E \rightarrow T$

Handle Pruning

Handle Pruning:

- We start with a string of terminals w to be parsed
- If w is a sentence of the grammar, then let w = γ_n

Where γ_n is the nth right sentential form of some unknown right derivation as

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n$$

- To reconstruct this derivation in reverse order, we locate the handle β_n in γ_n by relevant head of the production $A \rightarrow \beta_n$
- The β_n will be replaced by A to get previous right sentential form γ_{n-1}
- This process we called as Handle Pruning

Shift Reduce Parser

The Process:

Stack	Input
\$	w \$
...	
...	
\$ S	\$

- Initially stack is empty and string w is on the input
- The parser operates by shifting zero or more input symbols onto stack until a handle is on top of stack
- The parser then reduces the handle to the left side of the appropriate production
- The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty

Shift Reduce Parser

Actions of a Shift Reduce Parser:-

1. **Shift:** Shift the next input symbol onto the top of the stack.
2. **Reduce:**
 - a. The right end of the string to be reduced must be at the top of the stack.
 - b. Locate the left end of the string within the stack and decide with what non-terminal to replace the string.
3. **Accept:** Announce successful completion of parsing.
4. **Error:** Discover a syntax error and call an error recovery routine.

Shift Reduce Parser

Input: id * id

Q.1

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Stack	Input	Action
\$	id * id \$	Shift

Shift Reduce Parser

Input: id * id

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Stack	Input	Action
\$	id * id \$	Shift
\$ id	* id \$	Reduce by $F \rightarrow id$

Shift Reduce Parser

Input: id * id

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Stack	Input	Action
\$	id * id \$	Shift
\$ id	* id \$	Reduce by $F \rightarrow id$
\$ F	* id \$	Reduce by $T \rightarrow F$

Shift Reduce Parser

Input: id * id

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Stack	Input	Action
\$	id * id \$	Shift
\$ id	* id \$	Reduce by $F \rightarrow id$
\$ F	* id \$	Reduce by $T \rightarrow F$
\$ T	* id \$	Shift

Shift Reduce Parser

Input: id * id

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Stack	Input	Action
\$	id * id \$	Shift
\$ id	* id \$	Reduce by $F \rightarrow id$
\$ F	* id \$	Reduce by $T \rightarrow F$
\$ T	* id \$	Shift
\$ T *	id \$	Shift

Shift Reduce Parser

Input: id * id

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Stack	Input	Action
\$	id * id \$	Shift
\$ id	* id \$	Reduce by $F \rightarrow id$
\$ F	* id \$	Reduce by $T \rightarrow F$
\$ T	* id \$	Shift
\$ T *	id \$	Shift
\$ T * id	\$	Reduce by $F \rightarrow id$

Shift Reduce Parser

Input: id * id

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Stack	Input	Action
\$	id * id \$	Shift
\$ id	* id \$	Reduce by $F \rightarrow id$
\$ F	* id \$	Reduce by $T \rightarrow F$
\$ T	* id \$	Shift
\$ T *	id \$	Shift
\$ T * id	\$	Reduce by $F \rightarrow id$
\$ T * F	\$	Reduce by $T \rightarrow T * F$

Shift Reduce Parser

Input: id * id

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Stack	Input	Action
\$	id * id \$	Shift
\$ id	* id \$	Reduce by F → id
\$ F	* id \$	Reduce by T → F
\$ T	* id \$	Shift
\$ T *	id \$	Shift
\$ T * id	\$	Reduce by F → id
\$ T * F	\$	Reduce by T → T * F
\$ T	\$	Reduce by E → T

Shift Reduce Parser

Input: id * id

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Stack	Input	Action
\$	id * id \$	Shift
\$ id	* id \$	Reduce by F → id
\$ F	* id \$	Reduce by T → F
\$ T	* id \$	Shift
\$ T *	id \$	Shift
\$ T * id	\$	Reduce by F → id
\$ T * F	\$	Reduce by T → T * F
\$ T	\$	Reduce by E → T
\$ E	\$	Accept

Shift Reduce Parser

Input: id + id * id

Q.2

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

Stack	Input	Action
\$	id + id * id \$	Shift

Shift Reduce Parser

Input: id + id * id

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

Stack	Input	Action
\$	id + id * id \$	Shift
\$ id	+ id * id \$	Reduce by $E \rightarrow id$

Shift Reduce Parser

Input: id + id * id

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

Stack	Input	Action
\$	id + id * id \$	Shift
\$ id	+ id * id \$	Reduce by $E \rightarrow id$
\$ E	+ id * id \$	Shift

Shift Reduce Parser

Input: id + id * id

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

Stack	Input	Action
\$	id + id * id \$	Shift
\$ id	+ id * id \$	Reduce by $E \rightarrow id$
\$ E	+ id * id \$	Shift
\$ E +	id * id \$	Shift

Shift Reduce Parser

Input: id + id * id

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

Stack	Input	Action
\$	id + id * id \$	Shift
\$ id	+ id * id \$	Reduce by $E \rightarrow id$
\$ E	+ id * id \$	Shift
\$ E +	id * id \$	Shift
\$ E + id	* id \$	Reduce by $E \rightarrow id$

Shift Reduce Parser

Input: id + id * id

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

Stack	Input	Action
\$	id + id * id \$	Shift
\$ id	+ id * id \$	Reduce by $E \rightarrow id$
\$ E	+ id * id \$	Shift
\$ E +	id * id \$	Shift
\$ E + id	* id \$	Reduce by $E \rightarrow id$
\$ E + E	* id \$	Shift (Here Shift-Reduce Conflict)

Shift Reduce Parser

Input: id + id * id

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

Stack	Input	Action
\$	id + id * id \$	Shift
\$ id	+ id * id \$	Reduce by $E \rightarrow id$
\$ E	+ id * id \$	Shift
\$ E +	id * id \$	Shift
\$ E + id	* id \$	Reduce by $E \rightarrow id$
\$ E + E	* id \$	Shift (Here Shift-Reduce Conflict)
\$ E + E *	id \$	Shift

Shift Reduce Parser

Input: id + id * id

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

Stack	Input	Action
\$	id + id * id \$	Shift
\$ id	+ id * id \$	Reduce by $E \rightarrow id$
\$ E	+ id * id \$	Shift
\$ E +	id * id \$	Shift
\$ E + id	* id \$	Reduce by $E \rightarrow id$
\$ E + E	* id \$	Shift (Here Shift-Reduce Conflict)
\$ E + E * id	id \$	Shift
	\$	Reduce by $E \rightarrow id$

Shift Reduce Parser

Input: id + id * id

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

Stack	Input	Action
\$	id + id * id \$	Shift
\$ id	+ id * id \$	Reduce by $E \rightarrow id$
\$ E	+ id * id \$	Shift
\$ E +	id * id \$	Shift
\$ E + id	* id \$	Reduce by $E \rightarrow id$
\$ E + E	* id \$	Shift (Here Shift-Reduce Conflict)
\$ E + E *	id \$	Shift
\$ E + E * id	\$	Reduce by $E \rightarrow id$
\$ E + E * E	\$	Reduce by $E \rightarrow E * E$

Shift Reduce Parser

Input: id + id * id

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

Stack	Input	Action
\$	id + id * id \$	Shift
\$ id	+ id * id \$	Reduce by $E \rightarrow id$
\$ E	+ id * id \$	Shift
\$ E +	id * id \$	Shift
\$ E + id	* id \$	Reduce by $E \rightarrow id$
\$ E + E	* id \$	Shift (Here Shift-Reduce Conflict)
\$ E + E *	id \$	Shift
\$ E + E * id	\$	Reduce by $E \rightarrow id$
\$ E + <u>E * E</u>	\$	Reduce by $E \rightarrow E * E$
\$ E + E	\$	Reduce by $E \rightarrow E + E$

Shift Reduce Parser

Input: id + id * id

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

Stack	Input	Action
\$	id + id * id \$	Shift
\$ id	+ id * id \$	Reduce by $E \rightarrow id$
\$ E	+ id * id \$	Shift
\$ E +	id * id \$	Shift
\$ E + id	* id \$	Reduce by $E \rightarrow id$
\$ E + E	* id \$	Shift (Here Shift-Reduce Conflict)
\$ E + E *	id \$	Shift
\$ E + E * id	\$	Reduce by $E \rightarrow id$
\$ E + <u>E * E</u>	\$	Reduce by $E \rightarrow E * E$
\$ E + E	\$	Reduce by $E \rightarrow E + E$
\$ E	\$	Accept

SYNTAX ANALYSIS



BOTTOM-UP
PARSING

Operator Precedence Parser

- Operator precedence parser can be constructed from Operator Grammar
- Operator Grammar: The grammar which has property that no production on right side is ϵ or has two adjacent non-terminal
- Example:

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

Operator Precedance Parser

- There are three disjoint precedence relations
 - < less than
 - = Equal to
 - > Greater than
- Suppose there are two operators a and b then relations give following meaning:
 - $a < b$ – a gives precedence to b
 - $a = b$ – a has same precedence as of b
 - $a > b$ – a takes precedence over b

Operator Precedance Parser

Rules for finding operator precedance relations if a₁ & a₂ are operators

1. If a₁ has higher precedance than a₂ then make a₁ > a₂ and a₂ < a₁
2. If a₁ has equal precedance with a₂ and if operators are left associative then make a₁ > a₂ and a₂ > a₁
3. If a₁ has equal precedance with a₂ and if operators are right associative then make a₁ < a₂ and a₂ < a₁

Operator Precedance Parser

Rules for finding operator precedance relations if a₁ & a₂ are operators

4. For all operators a,

a < id and id > a

a < (and (< a

a >) and) > a

a > \$ and \$ < a

5. Operator ↑ has highest precedance and right associativity

6. Operator * and / has next higher precedance and left associativity

Operator Precedance Parser

Rules for finding operator precedance relations if a₁ & a₂ are operators

7. **Operator + and - has lowest precedance and left associativity**
8. The blank entries in operator precedance relation table indicates an error

Operator Precedence Relation Table

Operator Precedence Algorithm

Input: A string w and table of precedence relations

Output: If w is well formed, a Skeletal parse tree, with a placeholder non-terminal E labeling all interior nodes otherwise an error indication

Method: Initially the stack contains $\$$ and the input buffer the string $w \$$

Operator Precedence Algorithm

```
Set ip to point to the first symbol of w$  
Repeat forever  
    If $ is on top of stack and ip points to $ then  
        Return  
    else begin  
        Let a be the topmost terminal symbol on top of stack  
        And let b be the symbol pointed to by ip;  
        If a < b or a = b then begin  
            Push b on the top of the stack  
            Advance ip to the next symbol  
        End  
        Else if a > b then  
            Repeat  
                Pop of the stack  
                Until the top stack terminal is related by <  
                to the terminal most recently popped  
            Else  
                Error( )  
        End
```

Operator Precedence Parser

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

Input: id + id * id

	id	+	*	\$
id		v	v	v
+	<	v	<	v
*	<	v	>	v
\$	<	<	<	A

Operator Precedence Parser

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

Input: id + id * id

	id	+	*	\$
id		>	>	>
+	<		<	>
*	<	>	>	>
\$	<	<	<	A

Stack	Relation	Input	Comment
\$	<	id + id * id \$	Push id

Operator Precedence Parser

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

Input: id + id * id

	id	+	*	\$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
\$	<	<	<	A

Stack	Relation	Input	Comment
\$	<	id + id * id \$	Push id
\$ id	>	+ id * id \$	Pop id

Operator Precedence Parser

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

Input: id + id * id

	id	+	*	\$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
\$	<	<	<	A

Stack	Relation	Input	Comment
\$	<	id + id * id \$	Push id
\$ id	>	+ id * id \$	Pop id
\$	<	+ id * id \$	Push +

Operator Precedence Parser

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

Input: id + id * id

	id	+	*	\$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
\$	<	<	<	A

Stack	Relation	Input	Comment
\$	<	id + id * id \$	Push id
\$ id	>	+ id * id \$	Pop id
\$	<	+ id * id \$	Push +
\$ +	<	id * id \$	Push id

Operator Precedence Parser

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

Input: id + id * id

	id	+	*	\$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
\$	<	<	<	A

Stack	Relation	Input	Comment
\$	<	id + id * id \$	Push id
\$ id	>	+ id * id \$	Pop id
\$	<	+ id * id \$	Push +
\$ +	<	id * id \$	Push id
\$ + id	>	* id \$	Pop id

Operator Precedence Parser

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

Input: id + id * id

	id	+	*	\$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
\$	<	<	<	A

Stack	Relation	Input	Comment
\$	<	id + id * id \$	Push id
\$ id	>	+ id * id \$	Pop id
\$	<	+ id * id \$	Push +
\$ +	<	id * id \$	Push id
\$ + id	>	* id \$	Pop id
\$ +	<	* id \$	Push *

Operator Precedence Parser

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

Input: id + id * id

	id	+	*	\$
id		>	>	>
+	<		>	<
*	<		>	>
\$	<		<	A

Stack	Relation	Input	Comment
\$	<	id + id * id \$	Push id
\$ id	>	+ id * id \$	Pop id
\$	<	+ id * id \$	Push +
\$ +	<	id * id \$	Push id
\$ + id	>	* id \$	Pop id
\$ +	<	* id \$	Push *
\$ + *	<	id \$	Push id

Operator Precedence Parser

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

Input: id + id * id

	id	+	*	\$
id		>	>	>
+	<		>	<
*	<		>	>
\$	<		<	A

Stack	Relation	Input	Comment
\$	<	id + id * id \$	Push id
\$ id	>	+ id * id \$	Pop id
\$	<	+ id * id \$	Push +
\$ +	<	id * id \$	Push id
\$ + id	>	* id \$	Pop id
\$ +	<	* id \$	Push *
\$ + *	<	id \$	Push id
\$ + * id	>	\$	Pop id

Operator Precedence Parser

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

Input: id + id * id

	id	+	*	\$
id		>	>	>
+	<		>	<
*	<		>	>
\$	<		<	A

Stack	Relation	Input	Comment
\$	<	id + id * id \$	Push id
\$ id	>	+ id * id \$	Pop id
\$	<	+ id * id \$	Push +
\$ +	<	id * id \$	Push id
\$ + id	>	* id \$	Pop id
\$ +	<	* id \$	Push *
\$ + *	<	id \$	Push id
\$ + * id	>	\$	Pop id
\$ + *	>	\$	Pop *

Operator Precedence Parser

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

Input: id + id * id

	id	+	*	\$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
\$	<	<	<	A

Stack	Relation	Input	Comment
\$	<	id + id * id \$	Push id
\$ id	>	+ id * id \$	Pop id
\$	<	+ id * id \$	Push +
\$ +	<	id * id \$	Push id
\$ + id	>	* id \$	Pop id
\$ +	<	* id \$	Push *
\$ + *	<	id \$	Push id
\$ + * id	>	\$	Pop id
\$ + *	>	\$	Pop *
\$ +	>	\$	Pop +

Operator Precedence Parser

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

Input: id + id * id

	id	+	*	\$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
\$	<	<	<	A

Stack	Relation	Input	Comment
\$	<	id + id * id \$	Push id
\$ id	>	+ id * id \$	Pop id
\$	<	+ id * id \$	Push +
\$ +	<	id * id \$	Push id
\$ + id	>	* id \$	Pop id
\$ +	<	* id \$	Push *
\$ + *	<	id \$	Push id
\$ + * id	>	\$	Pop id
\$ + *	>	\$	Pop *
\$ +	>	\$	Pop +
\$	Accept	\$	Accept

Operator Precedence Parser

Steps to construct syntax tree (Expression Tree):

1. Keep track of elements that are popped in the same order.
Consider that sequence as input string for processing using stack.
2. Read the processing sequence from left to right
3. If operand (identifier) is found, then push that onto stack
4. If an operator is found, then pop out top 2 elements and construct subtree with operator as root node.
5. Push this newly constructed subtree on top of stack
6. Repeat step 3 to 5 until all the symbols in input string are processed.
7. When input string is completed then pop out topmost element of stack.
8. If stack is not empty after pop then declare ERROR otherwise POPPED ELEMENT is final SYNTAX TREE

Operator Precedence Parser

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

Input: id + id * id

Comment
Push id
Pop id
Push +
Push id
Pop id
Push *
Push id
Pop id
Pop *
Pop +
Accept

Operator Precedence Parser

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

Input: id + id * id

Comment
Pop id (1)
Pop id (2)
Pop id (3)
Pop * (4)
Pop + (5)
Accept

Sequence to be processed:

id , id , id, *, +

Operator Precedence Parser

$E \rightarrow E + E \mid E - E \mid E^* E \mid E/E \mid E \uparrow E \mid (E) \mid id$

Input: id + id * id

Stack

Sequence to be processed:

id , id , id, *, +

Operator Precedence Parser

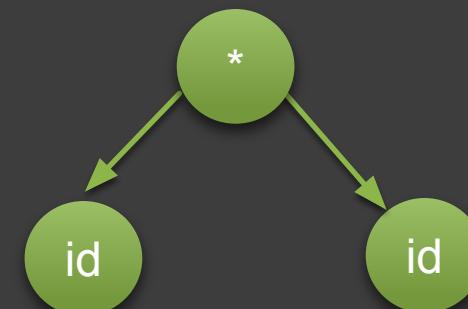
$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid id$

Input: id + id * id

Stack

Sequence to be processed:

id , id , id, *, +



Sub Tree 1

Operator Precedence Parser

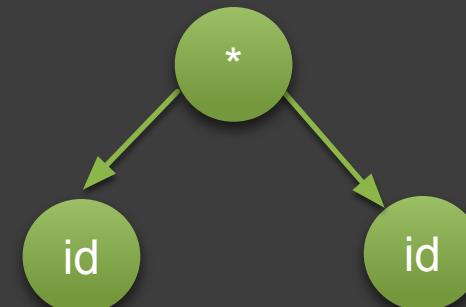
$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid id$

Input: id + id * id

Stack
Sub Tree 1
id
\$

Sequence to be processed:

id , id , id, *, +



Sub Tree 1

Operator Precedence Parser

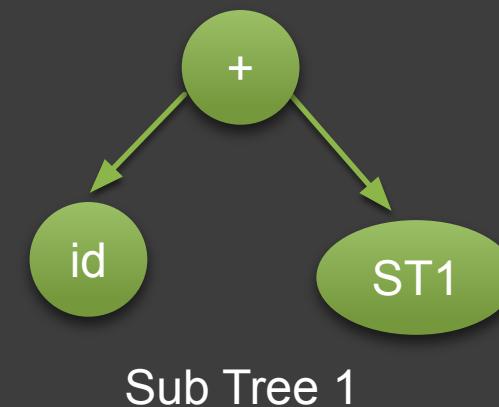
$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid id$

Input: id + id * id

Stack
\$

Sequence to be processed:

id , id , id, *, +



Operator Precedence Parser

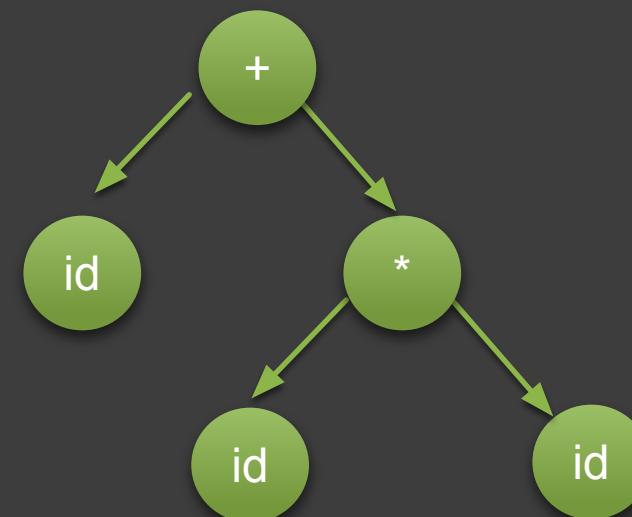
$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid id$

Input: id + id * id

Stack
\$

Sequence to be processed:

id , id , id, *, +



Sub Tree 2

Operator Precedence Parser

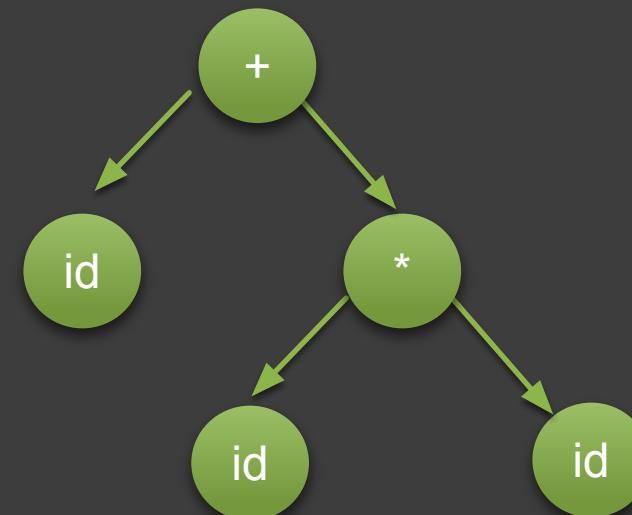
$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid id$

Input: id + id * id

Stack
Sub Tree 2
\$

Sequence to be processed:

id , id , id, *, +



Sub Tree 2

Operator Precedence Parser

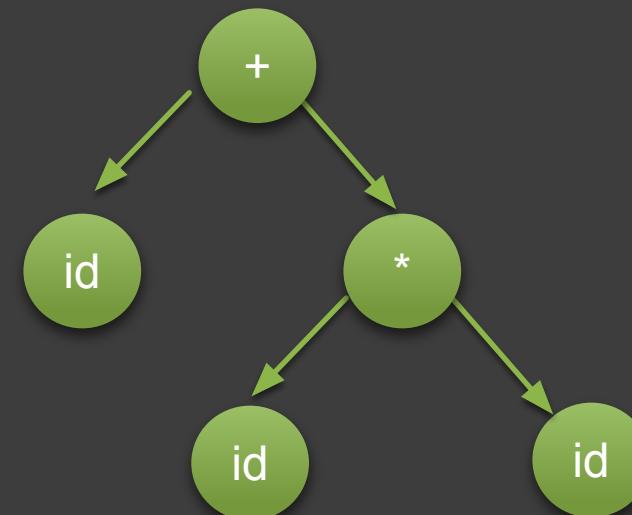
$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid id$

Input: id + id * id

Stack
\$

Sequence to be processed:

id , id , id, *, +



Final Tree

Operator Precedence Parser

Steps to construct Entire Derivation Tree:

1. Keep track of elements that are popped in the reverse order. (Start from Bottom and move in Top Direction)
2. Select start symbol as a root node.
3. Select the Next unprocessed symbol from list of popped elements.
4. Generate sub tree in which this symbol act as a leaf node by choosing one (or set) of the suitable production
5. Select rightmost node of this newly generated subtree such that it is non terminal. This node will act as a root node.
6. Repeat step 3 to 5 until all the symbols in popped sequence are processed.
7. The final tree is Required Derivation Tree

Operator Precedence Parser

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

Input: id + id * id

Construction of Entire Tree:

Comment
Pop id (5)
Pop id (4)
Pop id (3)
Pop * (2)
Pop + (1)
Accept

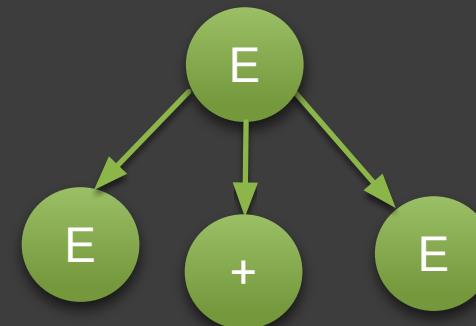
Operator Precedence Parser

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

Input: id + id * id

Comment
Pop id (5)
Pop id (4)
Pop id (3)
Pop * (2)
Pop + (1): $E \rightarrow E + E$
Accept

Construction of Entire Tree:



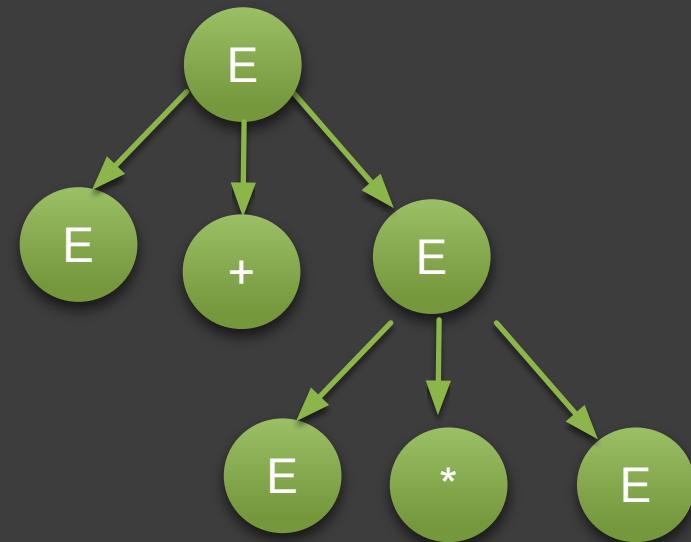
Operator Precedence Parser

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

Input: id + id * id

Comment
Pop id (5)
Pop id (4)
Pop id (3)
Pop * (2) : $E \rightarrow E * E$
Pop + (1): $E \rightarrow E + E$
Accept

Construction of Entire Tree:



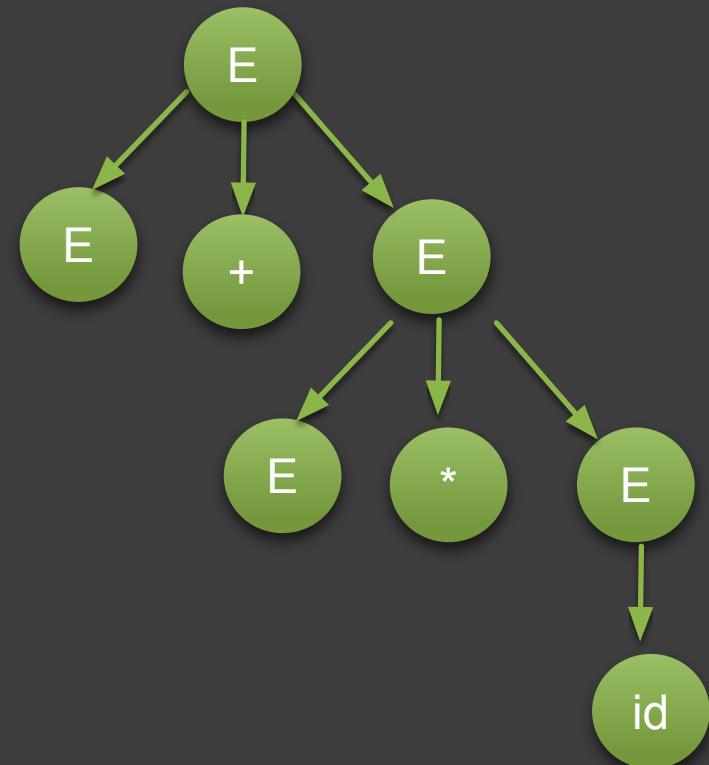
Operator Precedence Parser

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

Input: id + id * id

Comment
Pop id (5)
Pop id (4)
Pop id (3) : $E \rightarrow id$
Pop * (2) : $E \rightarrow E * E$
Pop + (1): $E \rightarrow E + E$
Accept

Construction of Entire Tree:



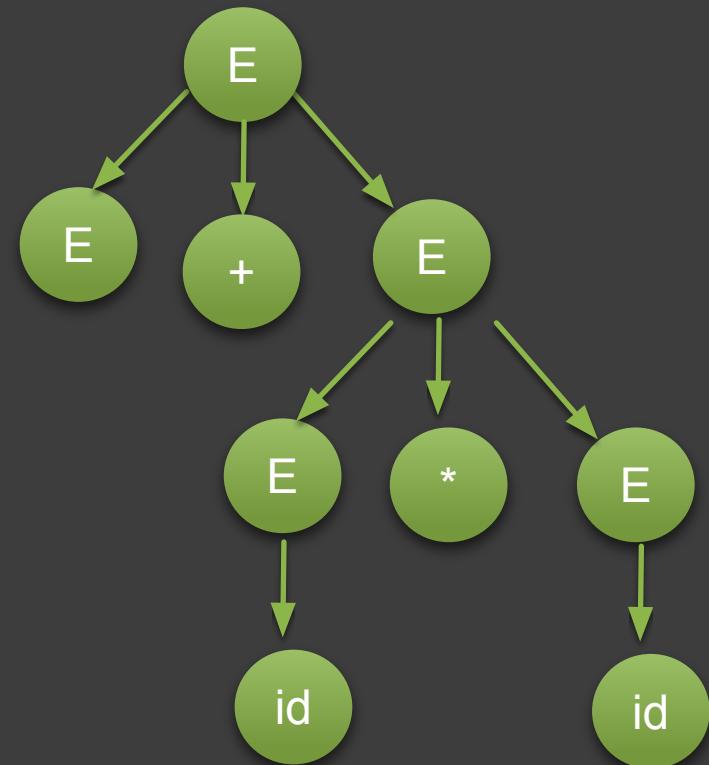
Operator Precedence Parser

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

Input: id + id * id

Comment
Pop id (5)
Pop id (4): $E \rightarrow id$
Pop id (3) : $E \rightarrow id$
Pop * (2) : $E \rightarrow E * E$
Pop + (1): $E \rightarrow E + E$
Accept

Construction of Entire Tree:



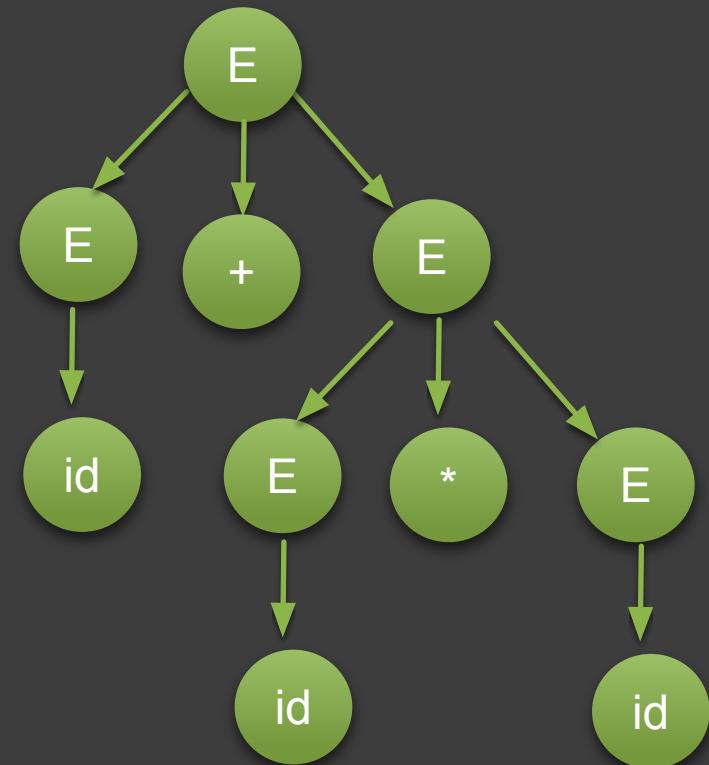
Operator Precedence Parser

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

Input: id + id * id

Comment
Pop id (5): $E \rightarrow id$
Pop id (4): $E \rightarrow id$
Pop id (3) : $E \rightarrow id$
Pop * (2) : $E \rightarrow E * E$
Pop + (1): $E \rightarrow E + E$
Accept

Construction of Entire Tree:



Operator Precedence Parser

$$E \rightarrow E + T | T$$

$$T \rightarrow T^*V \sqcup V$$

$V \rightarrow a | b | c | d$

Input: a + b * c * d

	a	b	c	d	+	*	\$
a					v	v	v
b					v	v	v
c					v	v	v
d					v	v	v
+	v	v	v	v	v	v	v
*	v	v	v	v	v	v	v
\$	v	v	v	v	v	v	A

Operator Precedence Parser

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * V \mid V$$
$$V \rightarrow a \mid b \mid c \mid d$$

Input: $a + b * c * d$

Stack	Relation	Input	Comment
\$	<	$a + b * c * d \$$	Push a

Operator Precedence Parser

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * V \mid V$$
$$V \rightarrow a \mid b \mid c \mid d$$

Input: $a + b * c * d$

Stack	Relation	Input	Comment
\$	<	$a + b * c * d \$$	Push a
\$ a	>	$+ b * c * d \$$	Pop a

Operator Precedence Parser

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T^* V \mid V$$
$$V \rightarrow a \mid b \mid c \mid d$$

Input: $a + b * c * d$

Stack	Relation	Input	Comment
\$	<	$a + b * c * d \$$	Push a
\$ a	>	$+ b * c * d \$$	Pop a
\$	<	$+ b * c * d \$$	Push +

Operator Precedence Parser

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * V \mid V$$
$$V \rightarrow a \mid b \mid c \mid d$$

Input: $a + b * c * d$

Stack	Relation	Input	Comment
\$	<	$a + b * c * d \$$	Push a
\$ a	>	$+ b * c * d \$$	Pop a
\$	<	$+ b * c * d \$$	Push +
\$ +	<	$b * c * d \$$	Push b

Operator Precedence Parser

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * V \mid V$$
$$V \rightarrow a \mid b \mid c \mid d$$

Input: $a + b * c * d$

Stack	Relation	Input	Comment
\$	<	$a + b * c * d \$$	Push a
\$ a	>	$+ b * c * d \$$	Pop a
\$	<	$+ b * c * d \$$	Push +
\$ +	<	$b * c * d \$$	Push b
\$ + b	>	$* c * d \$$	Pop b

Operator Precedence Parser

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * V \mid V$$
$$V \rightarrow a \mid b \mid c \mid d$$

Input: $a + b * c * d$

Stack	Relation	Input	Comment
\$	<	$a + b * c * d \$$	Push a
\$ a	>	$+ b * c * d \$$	Pop a
\$	<	$+ b * c * d \$$	Push +
\$ +	<	$b * c * d \$$	Push b
\$ + b	>	$* c * d \$$	Pop b
\$ +	<	$* c * d \$$	Push *

Operator Precedence Parser

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * V \mid V$$
$$V \rightarrow a \mid b \mid c \mid d$$

Input: $a + b * c * d$

Stack	Relation	Input	Comment
\$	<	$a + b * c * d \$$	Push a
\$ a	>	$+ b * c * d \$$	Pop a
\$	<	$+ b * c * d \$$	Push +
\$ +	<	$b * c * d \$$	Push b
\$ + b	>	$* c * d \$$	Pop b
\$ +	<	$* c * d \$$	Push *
\$ + *	<	$c * d \$$	Push c

Operator Precedence Parser

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * V \mid V$$
$$V \rightarrow a \mid b \mid c \mid d$$

Input: $a + b * c * d$

Stack	Relation	Input	Comment
\$	<	$a + b * c * d \$$	Push a
\$ a	>	$+ b * c * d \$$	Pop a
\$	<	$+ b * c * d \$$	Push +
\$ +	<	$b * c * d \$$	Push b
\$ + b	>	$* c * d \$$	Pop b
\$ +	<	$* c * d \$$	Push *
\$ + *	<	$c * d \$$	Push c
\$ + * c	>	$* d \$$	Pop c

Operator Precedence Parser

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * V \mid V$$
$$V \rightarrow a \mid b \mid c \mid d$$

Input: $a + b * c * d$

Stack	Relation	Input	Comment
\$	<	$a + b * c * d \$$	Push a
\$ a	>	$+ b * c * d \$$	Pop a
\$	<	$+ b * c * d \$$	Push +
\$ +	<	$b * c * d \$$	Push b
\$ + b	>	$* c * d \$$	Pop b
\$ +	<	$* c * d \$$	Push *
\$ + *	<	$c * d \$$	Push c
\$ + * c	>	$* d \$$	Pop c
\$ + *	>	$* d \$$	Pop *

Operator Precedence Parser

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * V \mid V$$
$$V \rightarrow a \mid b \mid c \mid d$$

Input: $a + b * c * d$

Stack	Relation	Input	Comment
\$	<	$a + b * c * d \$$	Push a
\$ a	>	$+ b * c * d \$$	Pop a
\$	<	$+ b * c * d \$$	Push +
\$ +	<	$b * c * d \$$	Push b
\$ + b	>	$* c * d \$$	Pop b
\$ +	<	$* c * d \$$	Push *
\$ + *	<	$c * d \$$	Push c
\$ + * c	>	$* d \$$	Pop c
\$ + *	>	$* d \$$	Pop *
\$ +	<	$* d \$$	Push *

Operator Precedence Parser

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * V \mid V$$
$$V \rightarrow a \mid b \mid c \mid d$$

Input: $a + b * c * d$

Stack	Relation	Input	Comment
\$	<	$a + b * c * d \$$	Push a
\$ a	>	$+ b * c * d \$$	Pop a
\$	<	$+ b * c * d \$$	Push +
\$ +	<	$b * c * d \$$	Push b
\$ + b	>	$* c * d \$$	Pop b
\$ +	<	$* c * d \$$	Push *
\$ + *	<	$c * d \$$	Push c
\$ + * c	>	$* d \$$	Pop c
\$ + *	>	$* d \$$	Pop *
\$ +	<	$* d \$$	Push *
\$ + *	<	$d \$$	Push d

Operator Precedence Parser

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

$$T \rightarrow T * V \mid V$$

$$V \rightarrow a \mid b \mid c \mid d$$

Input: $a + b * c * d$

Stack	Relation	Input	Comment
\$	<	$a + b * c * d \$$	Push a
\$ a	>	$+ b * c * d \$$	Pop a
\$	<	$+ b * c * d \$$	Push +
\$ +	<	$b * c * d \$$	Push b
\$ + b	>	$* c * d \$$	Pop b
\$ +	<	$* c * d \$$	Push *
\$ + *	<	$c * d \$$	Push c
\$ + * c	>	$* d \$$	Pop c
\$ + *	>	$* d \$$	Pop *
\$ +	<	$* d \$$	Push *
\$ + *	<	$d \$$	Push d
\$ + * d	>	\$	Pop d

Operator Precedence Parser

 $E \rightarrow E + T \mid T$
 $T \rightarrow T * V \mid V$
 $V \rightarrow a \mid b \mid c \mid d$

 Input: $a + b * c * d$

Stack	Relation	Input	Comment
\$	<	$a + b * c * d \$$	Push a
\$ a	>	$+ b * c * d \$$	Pop a
\$	<	$+ b * c * d \$$	Push +
\$ +	<	$b * c * d \$$	Push b
\$ + b	>	$* c * d \$$	Pop b
\$ +	<	$* c * d \$$	Push *
\$ + *	<	$c * d \$$	Push c
\$ + * c	>	$* d \$$	Pop c
\$ + *	>	$* d \$$	Pop *
\$ +	<	$* d \$$	Push *
\$ + *	<	$d \$$	Push d
\$ + * d	>	\$	Pop d
\$ + *	>	\$	Pop *

$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Stack	Relation	Input	Comment
\$	<	$a + b * c * d \$$	Push a
\$ a	>	$+ b * c * d \$$	Pop a
\$	<	$+ b * c * d \$$	Push +
\$ +	<	$b * c * d \$$	Push b
\$ + b	>	$* c * d \$$	Pop b
\$ +	<	$* c * d \$$	Push *
\$ + *	<	$c * d \$$	Push c
\$ + * c	>	$* d \$$	Pop c
\$ + *	>	$* d \$$	Pop *
\$ +	<	$* d \$$	Push *
\$ + *	<	$d \$$	Push d
\$ + * d	>	\$	Pop d
\$ + *	>	\$	Pop *
\$ +	>	\$	Pop +

$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b^* c^* d$

Stack	Relation	Input	Comment
\$	<	$a + b^* c^* d \$$	Push a
\$ a	>	$+ b^* c^* d \$$	Pop a
\$	<	$+ b^* c^* d \$$	Push +
\$ +	<	$b^* c^* d \$$	Push b
\$ + b	>	$* c^* d \$$	Pop b
\$ +	<	$* c^* d \$$	Push *
\$ + *	<	$c^* d \$$	Push c
\$ + * c	>	$* d \$$	Pop c
\$ + *	>	$* d \$$	Pop *
\$ +	<	$* d \$$	Push *
\$ + *	<	$d \$$	Push d
\$ + * d	>	\$	Pop d
\$ + *	>	\$	Pop *
\$ +	>	\$	Pop +
\$	Accept	\$	Accept

$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$

Input: a + b * c * d

Comment
Push a
Pop a
Push +
Push b
Pop b
Push *
Push c
Pop c
Pop *
Push *
Push d
Pop d
Pop *
Pop +
Accept

$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$

Input: a + b * c * d

Comment
Pop a (1)
Pop b (2)
Pop c (3)
Pop * (4)
Pop d (5)
Pop * (6)
Pop + (7)
Accept

Sequence to be processed:

a , b , c , * , d , * , +

$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Sequence to be processed:

 $a , b , c , ^* , d , ^* , +$

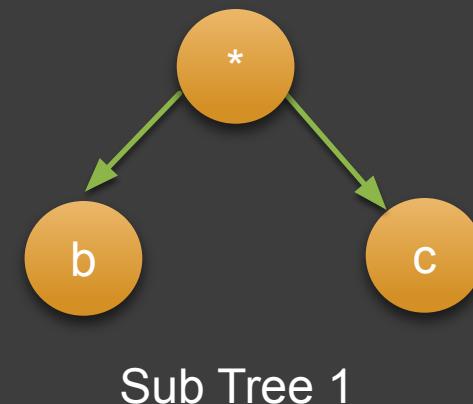
Stack
c
b
a
\$

$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Sequence to be processed:

 $a, b, c, *, d, *, +$

Stack
a
$\$$

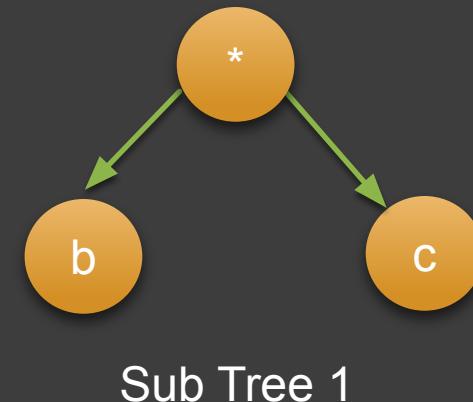


$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Sequence to be processed:

 $a, b, c, *, d, *, +$

Stack
Sub Tree 1
a
\$

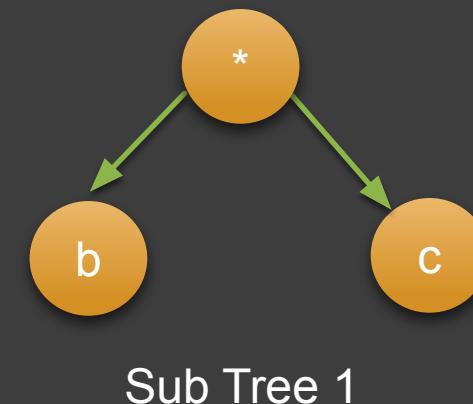


$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Sequence to be processed:

 $a, b, c, *, d, *, +$

Stack
d
Sub Tree 1
a
\$

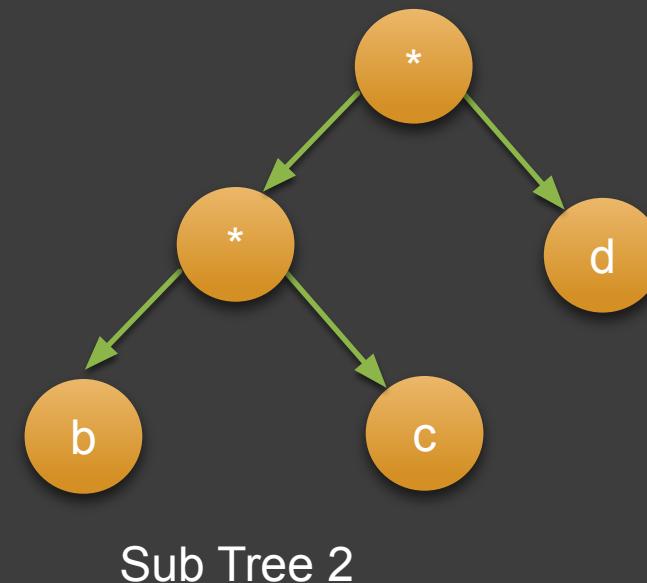


$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Sequence to be processed:

 $a, b, c, *, d, *, +$

Stack
a
$\$$



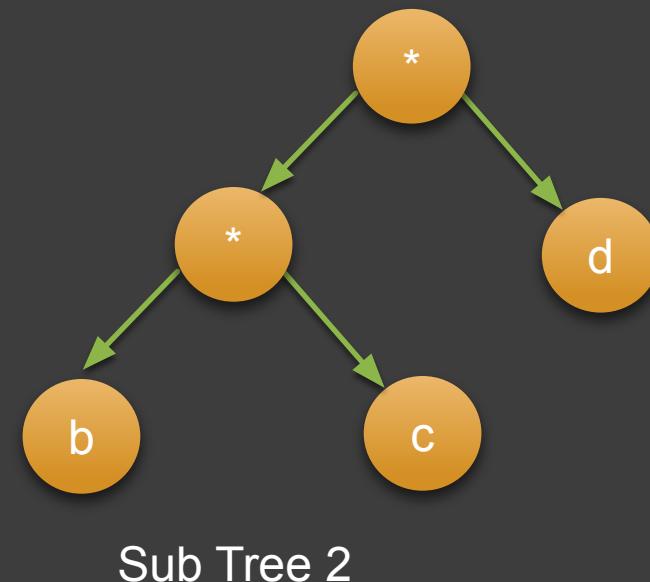
$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$

Input: a + b * c * d

Sequence to be processed:

a , b , c , * , d , * , +

Stack
Sub Tree 2
a
\$

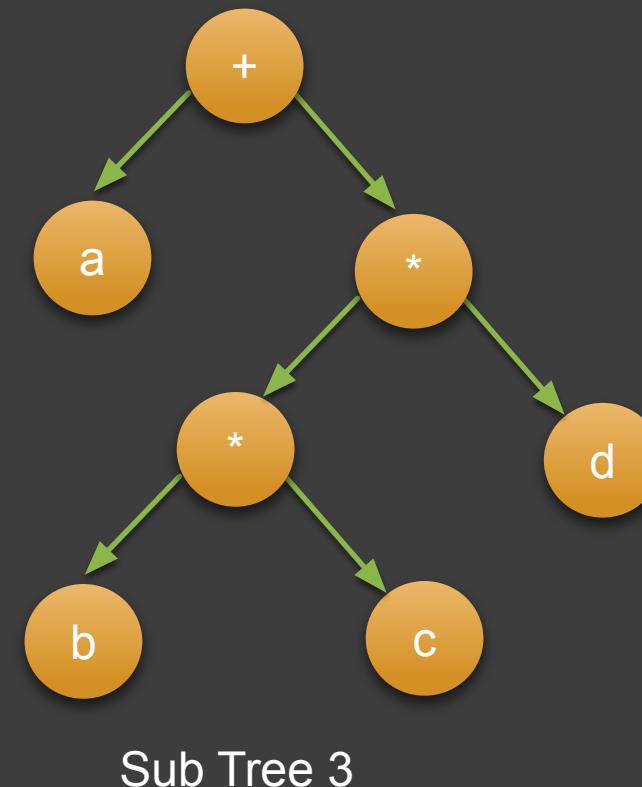


$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Sequence to be processed:

 $a, b, c, *, d, *, +$

Stack
\$

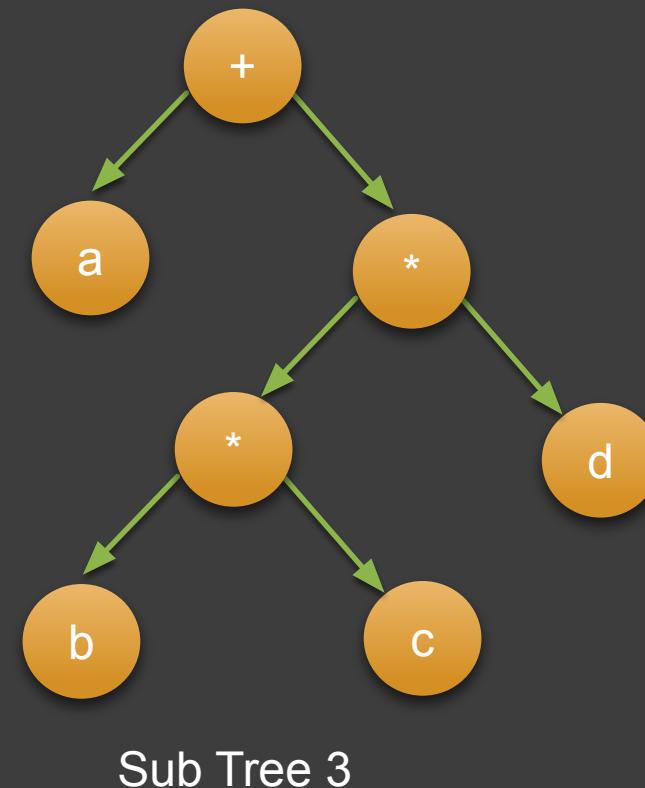


$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Sequence to be processed:

 $a, b, c, *, d, *, +$

Stack
Sub Tree 3
\$

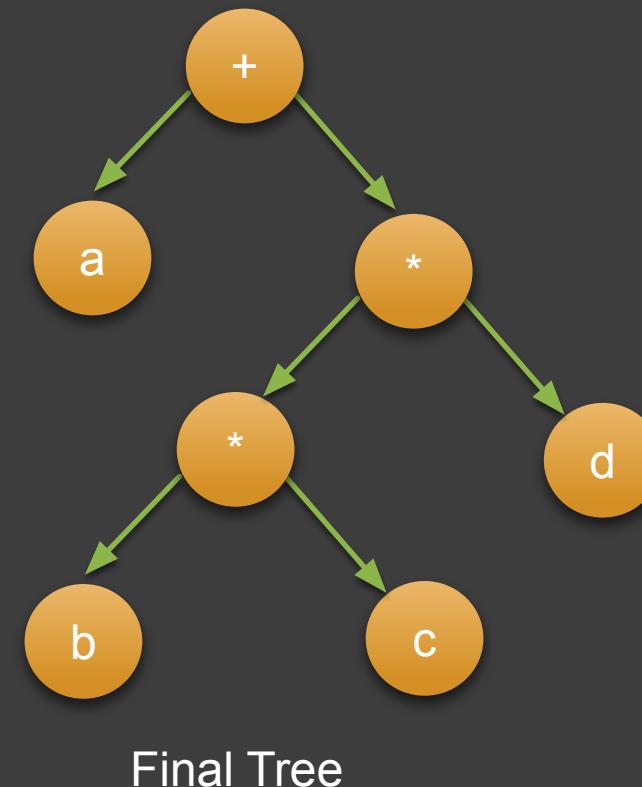


$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Sequence to be processed:

 $a, b, c, *, d, *, +$

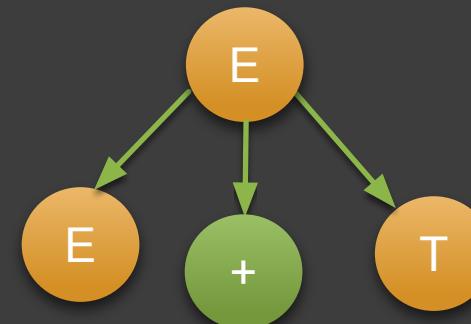
Stack
\$



$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Comment
Pop a (7)
Pop b (6)
Pop c (5)
Pop * (4)
Pop d (3)
Pop * (2)
Pop + (1) : $E \rightarrow E + T$
Accept

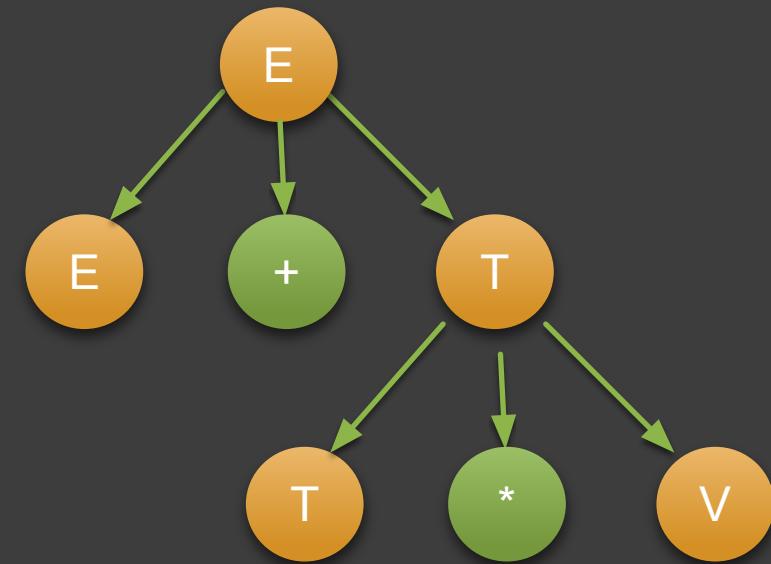
Construction of Entire Tree:



$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Comment
Pop a (7)
Pop b (6)
Pop c (5)
Pop * (4)
Pop d (3)
Pop * (2) : $T \rightarrow T^* V$
Pop + (1) : $E \rightarrow E + T$
Accept

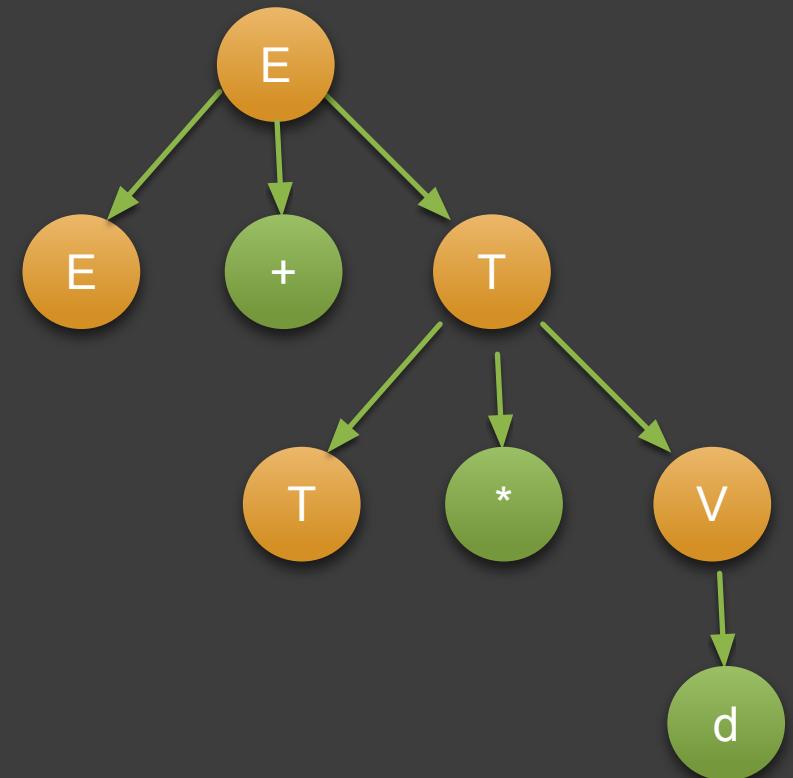
Construction of Entire Tree:



$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Comment
Pop a (7)
Pop b (6)
Pop c (5)
Pop * (4)
Pop d (3) : $V \rightarrow d$
Pop * (2) : $T \rightarrow T^* V$
Pop + (1) : $E \rightarrow E + T$
Accept

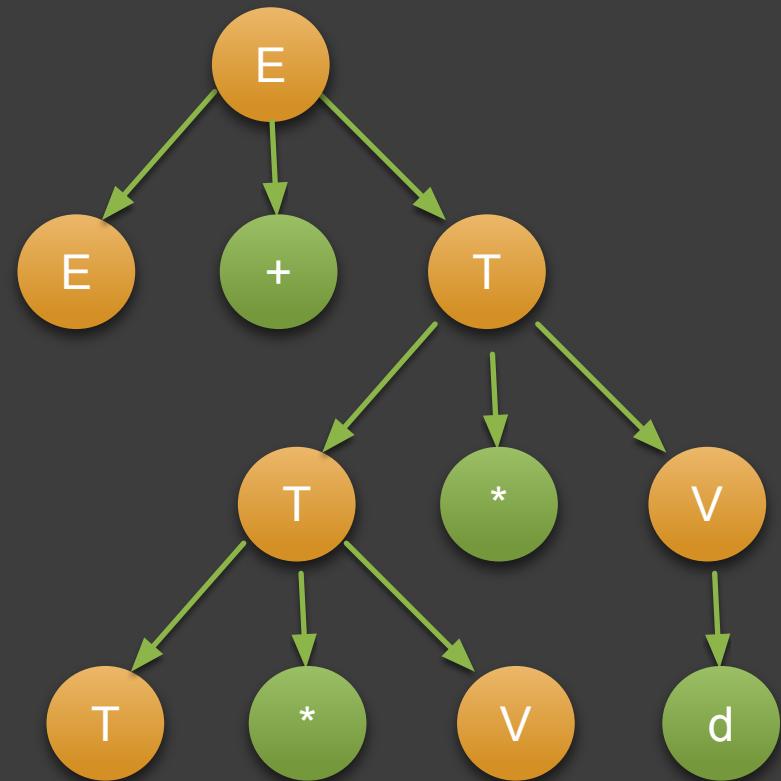
Construction of Entire Tree:



$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Comment
Pop a (7)
Pop b (6)
Pop c (5)
Pop * (4) : $T \rightarrow T^* V$
Pop d (3) : $V \rightarrow d$
Pop * (2) : $T \rightarrow T^* V$
Pop + (1) : $E \rightarrow E + T$
Accept

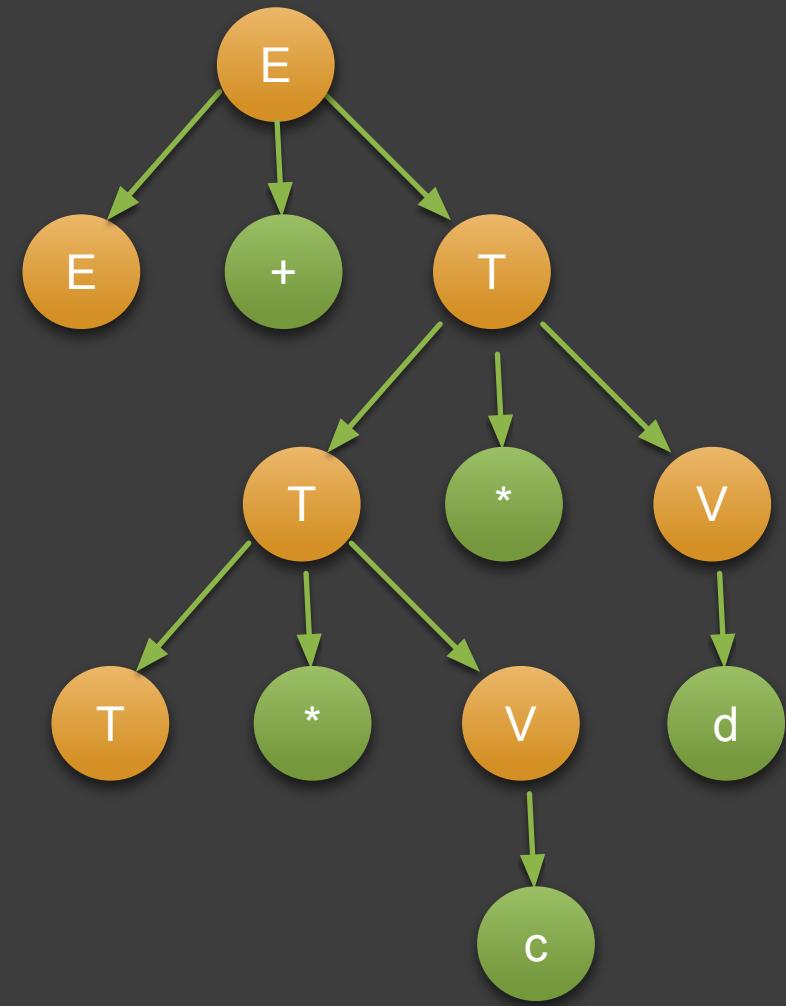
Construction of Entire Tree:



$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Comment
Pop a (7)
Pop b (6)
Pop c (5) : $V \rightarrow c$
Pop * (4) : $T \rightarrow T^* V$
Pop d (3) : $V \rightarrow d$
Pop * (2) : $T \rightarrow T^* V$
Pop + (1) : $E \rightarrow E + T$
Accept

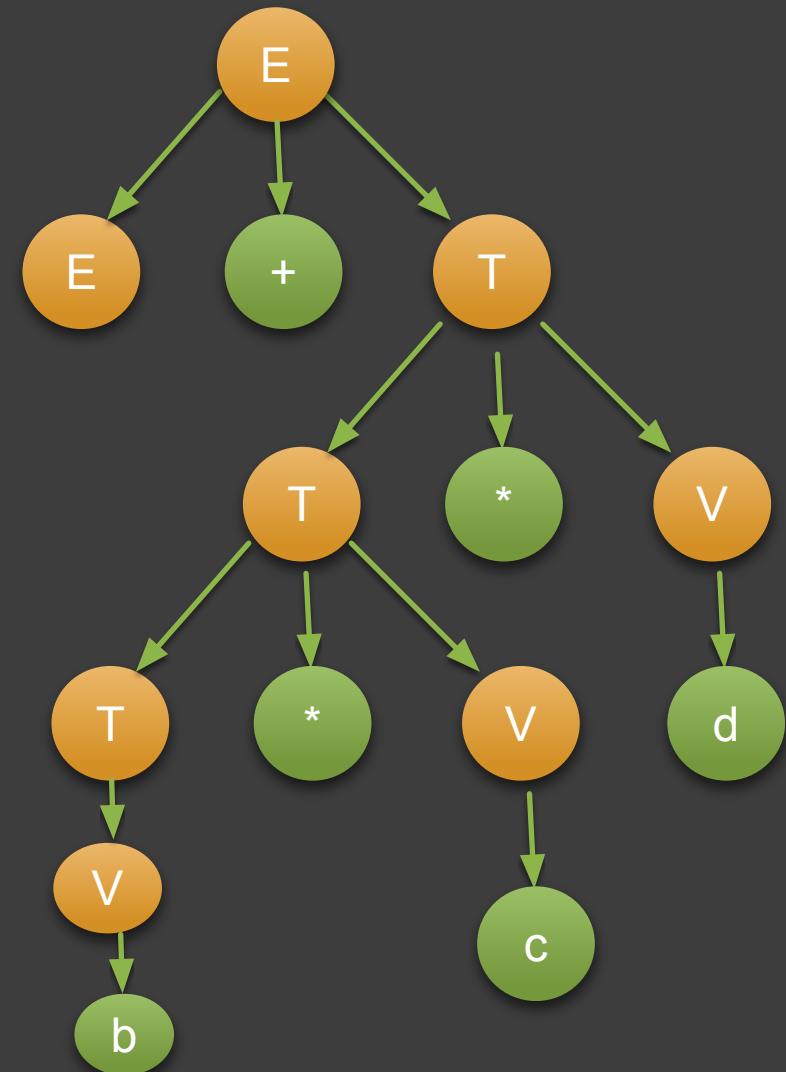
Construction of Entire Tree:



$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Comment
Pop a (7)
Pop b (6) : $T \rightarrow V, V \rightarrow b$
Pop c (5) : $V \rightarrow c$
Pop * (4) : $T \rightarrow T^* V$
Pop d (3) : $V \rightarrow d$
Pop * (2) : $T \rightarrow T^* V$
Pop + (1) : $E \rightarrow E + T$
Accept

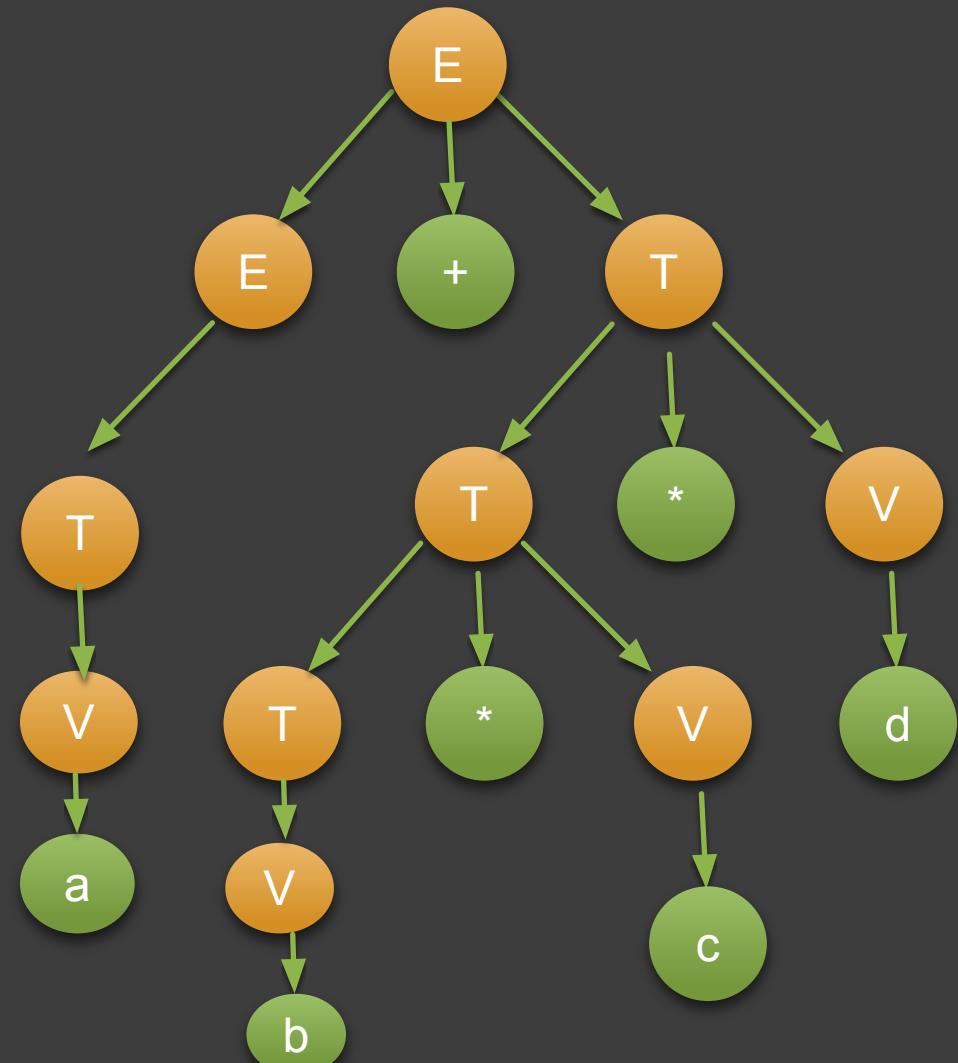
Construction of Entire Tree:

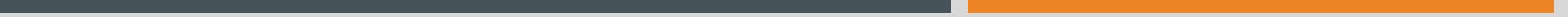


$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Comment
Pop a (7) : $E \rightarrow T, T \rightarrow V, V \rightarrow a$
Pop b (6) : $T \rightarrow V, V \rightarrow b$
Pop c (5) : $V \rightarrow c$
Pop * (4) : $T \rightarrow T^* V$
Pop d (3) : $V \rightarrow d$
Pop * (2) : $T \rightarrow T^* V$
Pop + (1) : $E \rightarrow E + T$
Accept

Construction of Entire Tree:





SYNTAX ANALYSIS

SLR PARSER

Introduction To LR Parser

- Most prevalent type of Bottom-Up Parser
- Known as LR (k) parser
 - L stands for Left to Right scanning of input
 - R stands for Rightmost Derivation in reverse
 - k used for number of input symbols of look ahead for making parsing decisions
- k = 0 or k = 1 is used for practical interest.
- When k is omitted then it is considered as 1
- Examples: SLR, Canonical LR and LALR

Why LR Parser??

- Table driven similar to Non recursive LL parser
- They can be constructed to recognize virtually all programming language construct for which context free grammar can be written
- Parsing method is the most General non-backtracking Shift Reduce
- Parsing Method can be implemented efficiently
- Can detect syntactic error at earliest from left to right scan
- The class of grammar that can be parsed using LR methods is the proper SUPERSET of the class of the grammar that can be parsed using LL Method

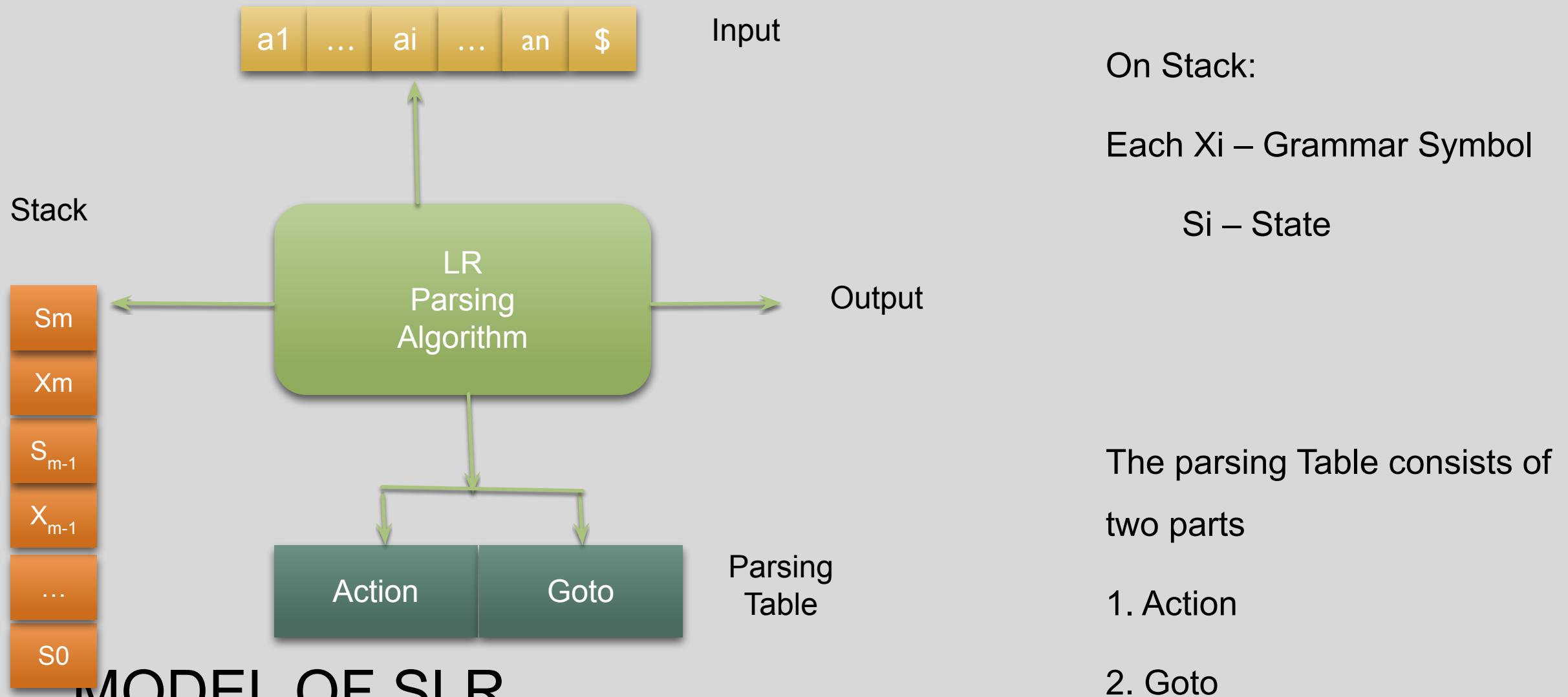
Why LR Parser??

Drawback:

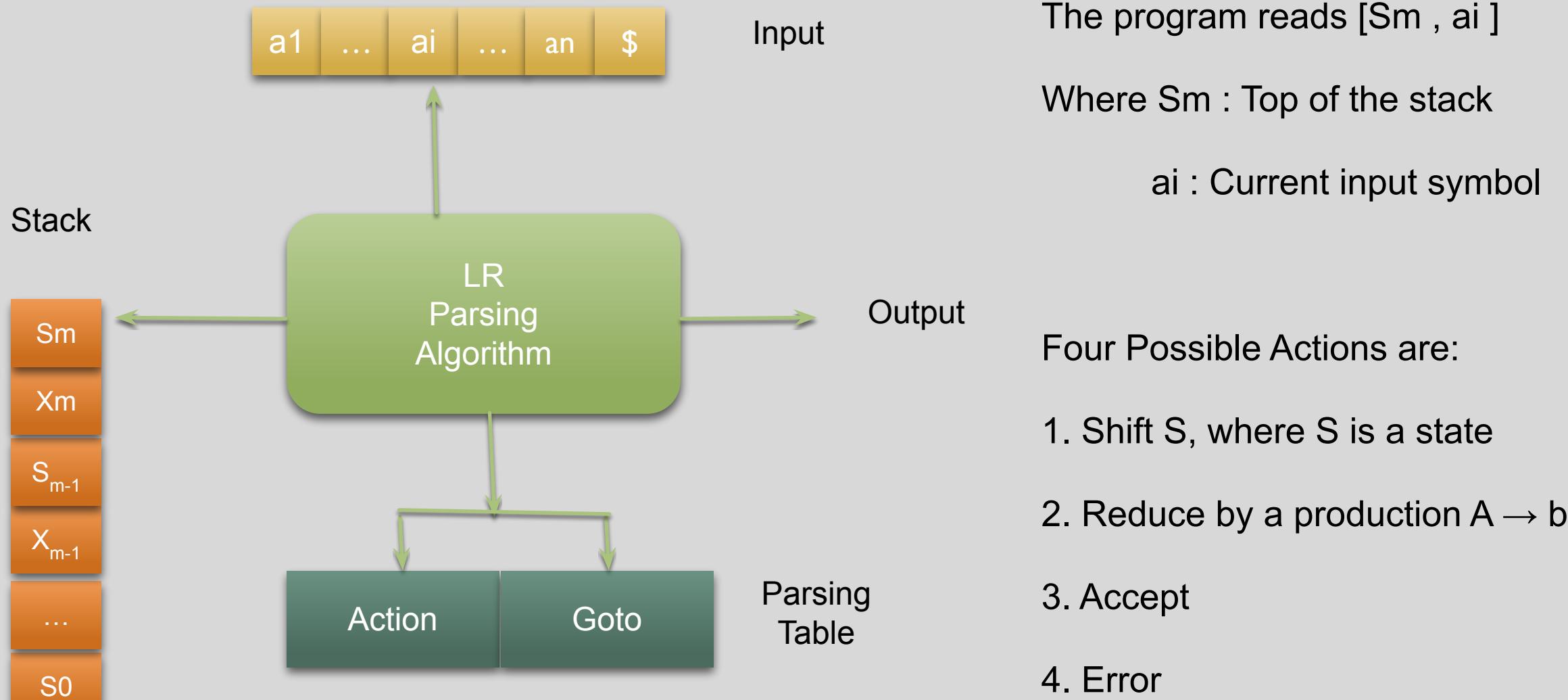
- Too much work to construct LR Parser by hand for typical programming-language grammar

Solution: LR Parser generator is needed.

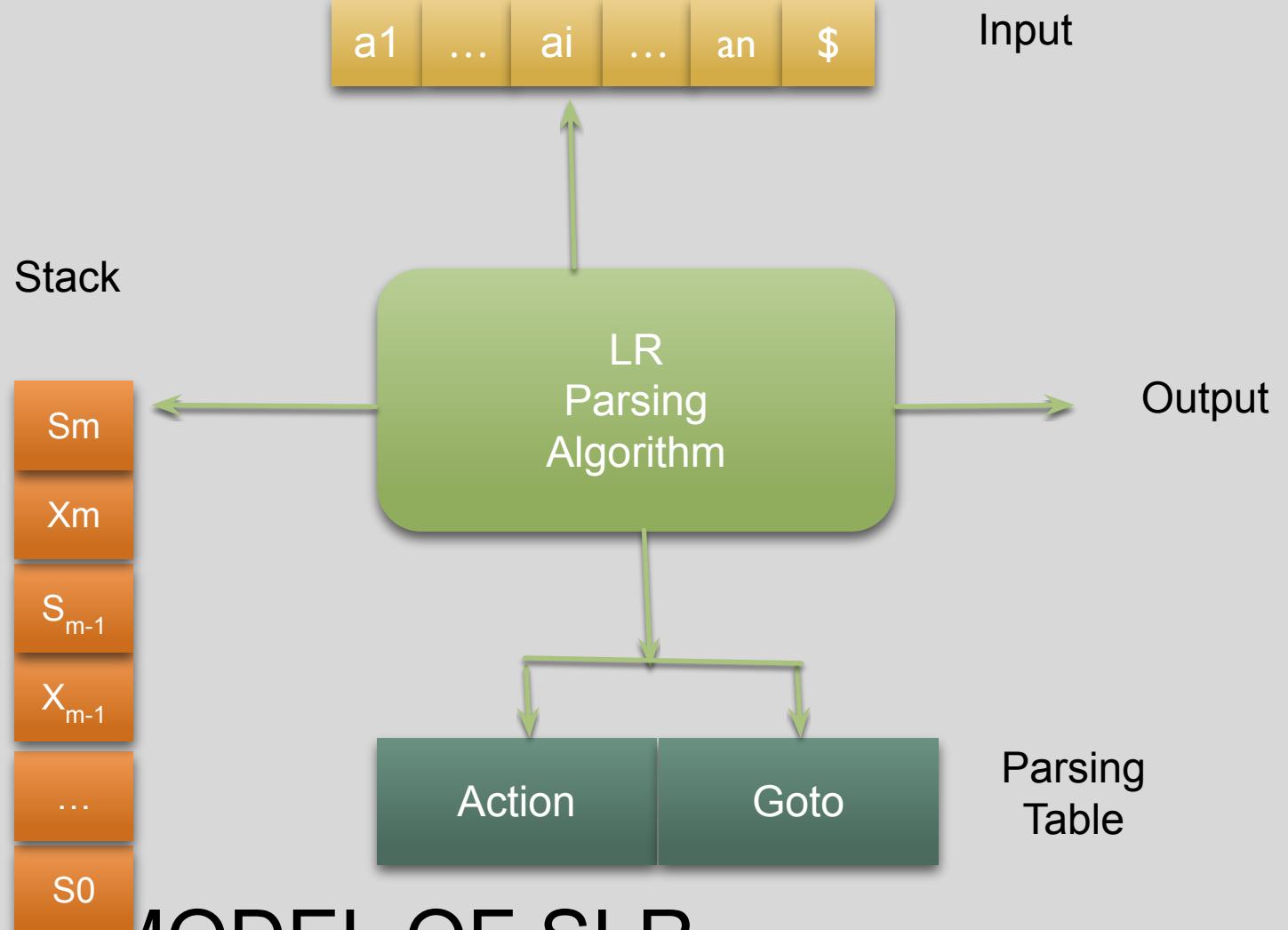
Example: YACC tool



MODEL OF SLR PARSER



MODEL OF SLR PARSER



The function Goto takes a state and grammar symbol as arguments and produce states

MODEL OF SLR PARSER

Construction Of SLR Parsing Table

Central Idea

To construct a DFA from the given grammar to recognize viable prefixes

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

Construction of LR(0) Items

Example:

If $A \rightarrow XYZ$ then there are four possible LR(0) items as:

$A \rightarrow .XYZ$

$A \rightarrow X .YZ$

$A \rightarrow XY .Z$

$A \rightarrow XYZ .$

Augmented Grammar

Given: If a grammar G is with start symbol S

then G' is augmented Grammar for G

Two elements are added in G to get G'

1. New start symbol G'
2. New production $S' \rightarrow S$

Acceptance of string is announced only when parser is about to reduce $S' \rightarrow S$

Closure Operation

If I is a set of items for a grammar G then

The closure (I) is the set of items constructed from I by the two rules as:

1. Initially every item in I is added in closure (I)
2. If $A \rightarrow \alpha . B \beta$ is in closure (I) and $B \rightarrow r$ then add the item $B \rightarrow . r$ to closure(I) if it is not already there.

Apply the rule until no more rules can be added

Closure Operation

Example:

Given Grammar as

$$E' \rightarrow E$$

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

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

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

If I is the set of one item $\{ [E' \rightarrow . E] \}$ then closure of I contains

$$E' \rightarrow . E$$

$$E \rightarrow . E + T$$

$$E \rightarrow . T$$

$$T \rightarrow . T^* F$$

$$T \rightarrow . F$$

$$F \rightarrow . (E)$$

$$F \rightarrow . id$$

GOTO Operation

If GOTO (I, X) where I is the set of items and X is a grammar symbol.

If I contains [A → α . X β] then

GOTO (I, X) - Closure of the set of all items [A → α X . β]

Example:

If I is set of two items as { [E' → E .], [E → E . + T] }

Then GOTO (I, +) consists of

E → E + . T

T → . T * F

T → . F

F → . (E)

F → . id

Construction of LR (0) Automation

- Creation of Transition Diagram
- Each state is obtained by shifting of “dot”
- Construction of Augmented Grammar is required

State 1: I0

Grammar

Consider Production with start symbol

$$E' \rightarrow . E$$

$$E' \rightarrow E$$

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

Add to I0

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

$$E \rightarrow . E + T$$

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

$$E \rightarrow . T$$

$$T \rightarrow . T^* F$$

$$T \rightarrow . F$$

$$F \rightarrow . (E)$$

$$F \rightarrow . id$$

State I0:

$$E' \rightarrow . E$$

$$E \rightarrow . E + T$$

$$E \rightarrow . T$$

$$T \rightarrow . T^* F$$

$$T \rightarrow . F$$

$$F \rightarrow . (E)$$

$$F \rightarrow . id$$

Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

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

$F \rightarrow (E) \mid id$

State I0:

$E' \rightarrow . E$

$E \rightarrow . E + T$

$E \rightarrow . T$

$T \rightarrow . T^* F$

$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . id$

GOTO (I0 , E) :

$E' \rightarrow E .$

$E \rightarrow E . + T$ [New State: I1]

State I1:

$E' \rightarrow E .$

$E \rightarrow E . + T$

GOTO (I0 , T) :

$E \rightarrow T .$

$T \rightarrow T . ^* F$ [New State: I2]

State I2:

$E \rightarrow T .$

$T \rightarrow T . ^* F$

Grammar

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

GOTO (I0 , F) :

$T \rightarrow F .$ [New State: I3]

State I3:

$T \rightarrow F.$

GOTO (I0 , ()) :

$F \rightarrow (. E)$

$E \rightarrow . E + T$

$E \rightarrow . T$

$T \rightarrow . T^* F$

$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . id$

[New State: I4]

State I4:

$F \rightarrow (. E)$

$E \rightarrow . E + T$

$E \rightarrow . T$

$T \rightarrow . T^* F$

$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . id$

Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

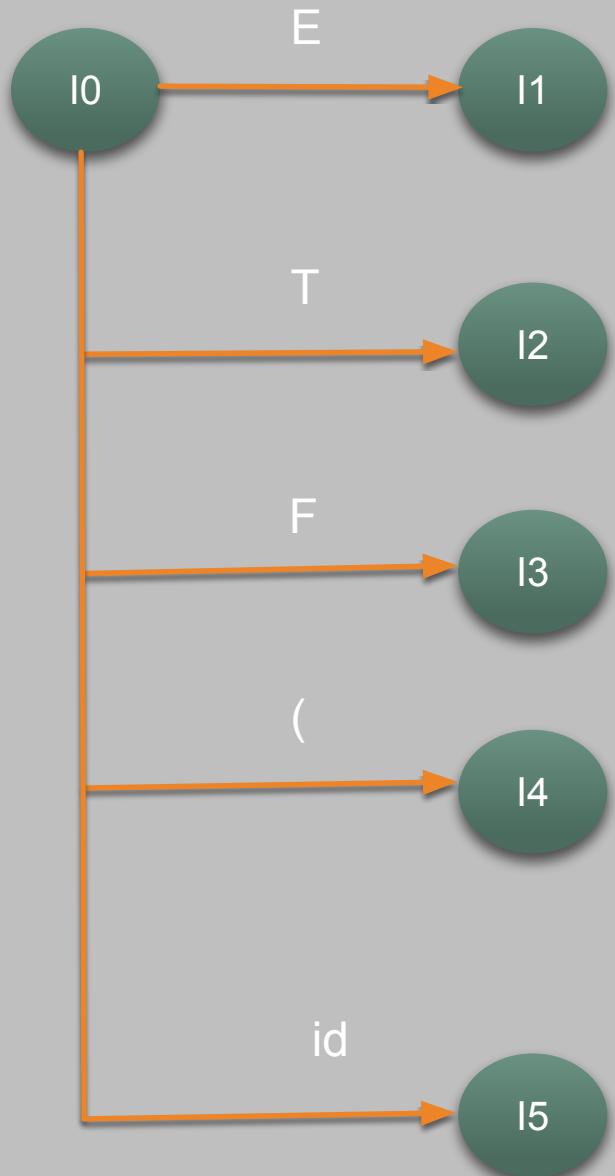
$F \rightarrow (E) \mid id$

GOTO (I0 , id) :

$F \rightarrow id . \quad [New\ State: I5]$

State I5:

$F \rightarrow id.$



Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

State I1:

$E' \rightarrow E .$

$E \rightarrow E . + T$

GOTO (I1 , +) :

$E \rightarrow E + . T$

$T \rightarrow . T * F$

$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . id$

[New State: I6]

State I6:

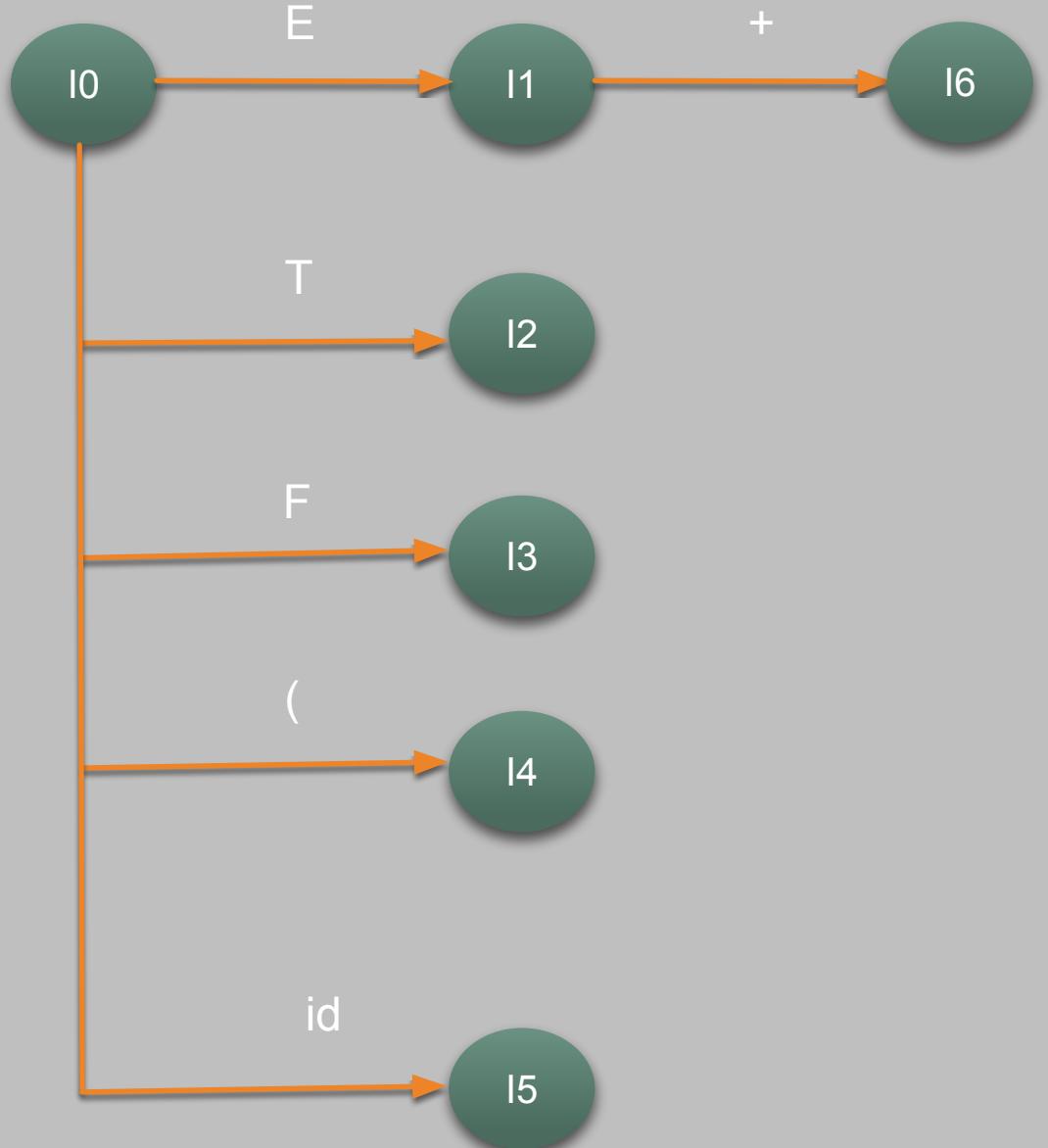
$E \rightarrow E + . T$

$T \rightarrow . T * F$

$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . id$



Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

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

$F \rightarrow (E) \mid id$

State I2:

$E \rightarrow T .$

$T \rightarrow T .^* F$

GOTO (I2 , *) :

$T \rightarrow T^* . F$

$F \rightarrow . (E)$

$F \rightarrow . id$

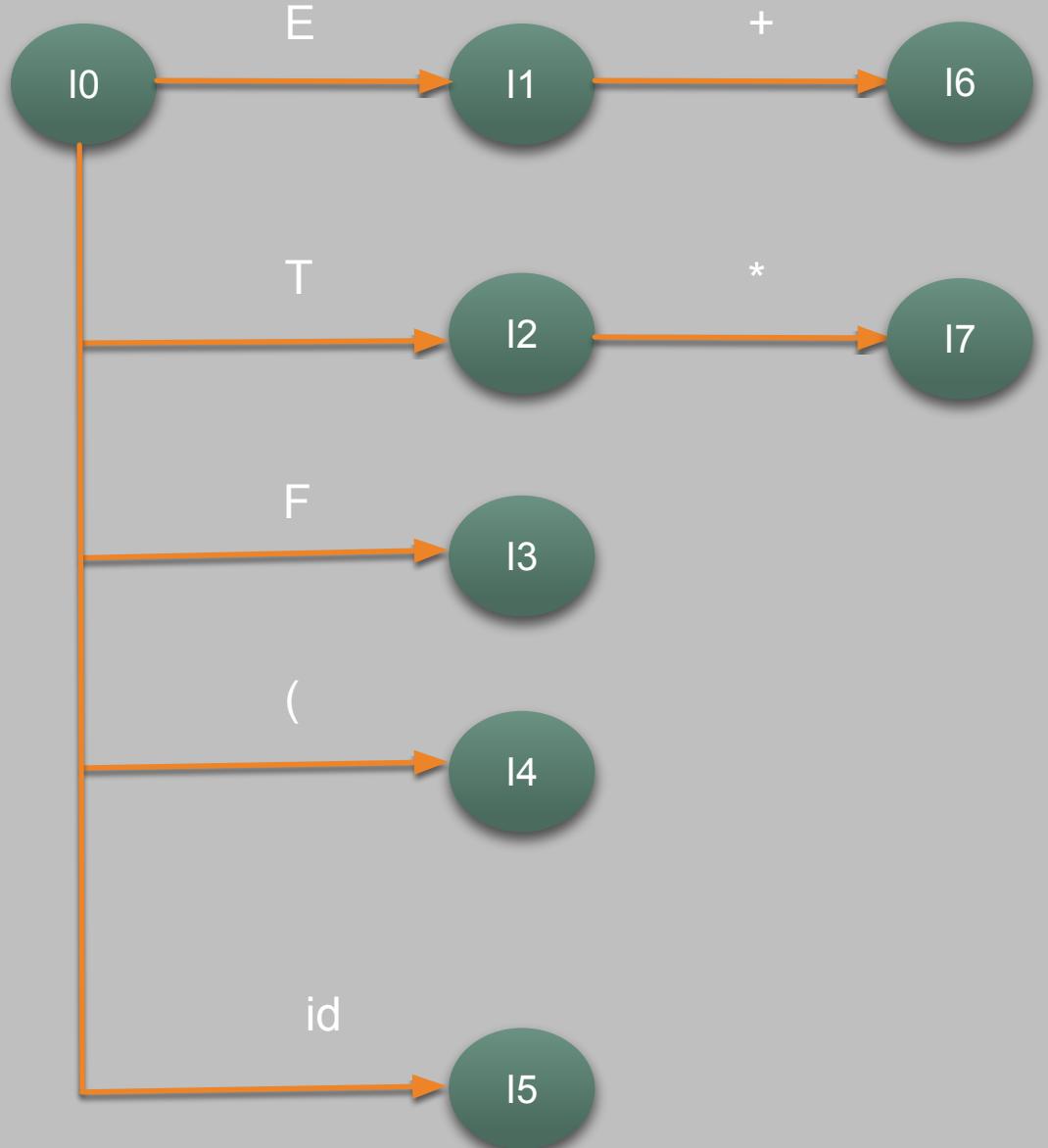
[New State: I7]

State I7:

$T \rightarrow T^* . F$

$F \rightarrow . (E)$

$F \rightarrow . id$



Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

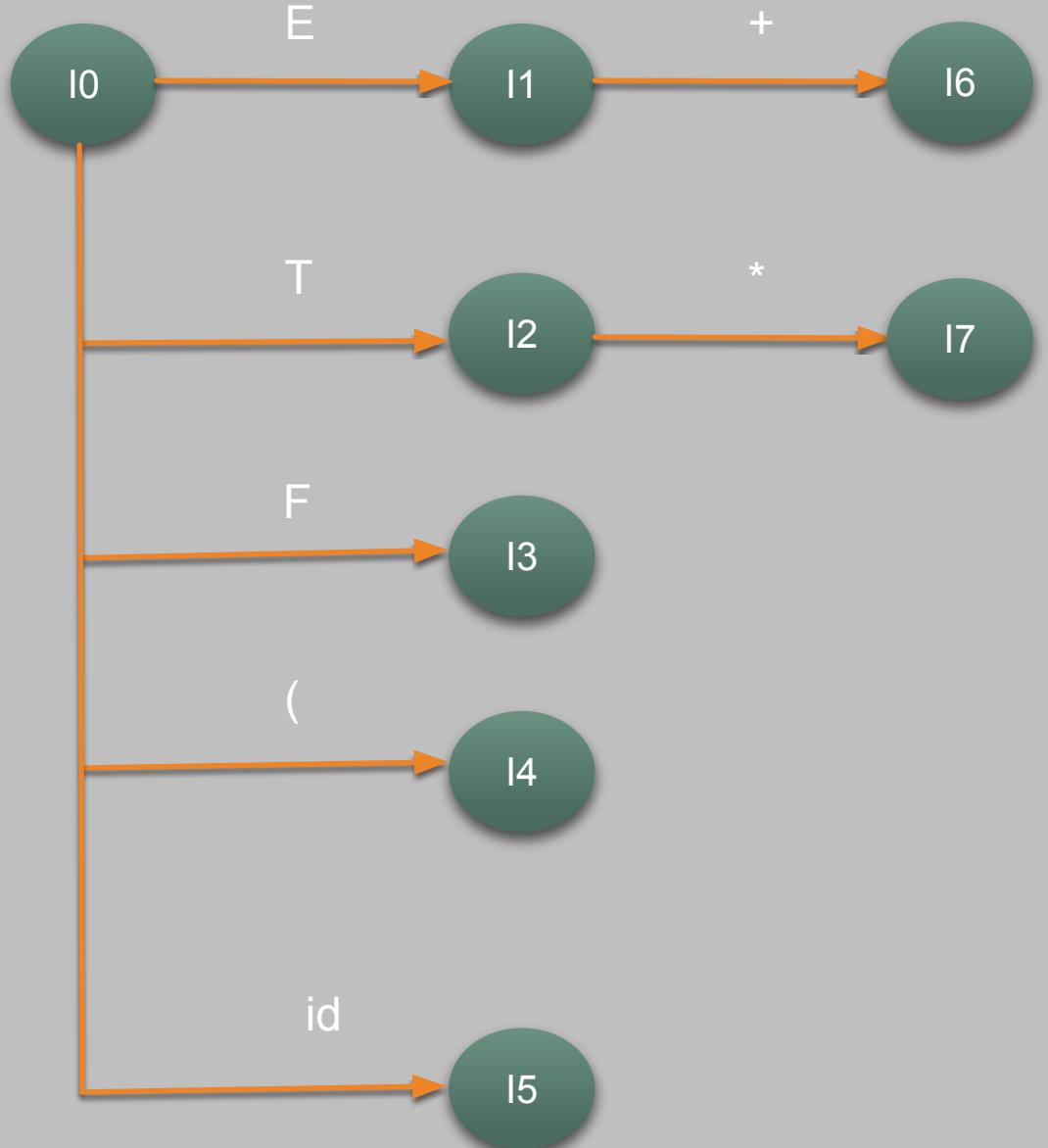
$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

No possible GOTO operation

State I3:

$T \rightarrow F .$



Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

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

$F \rightarrow (E) \mid id$

State I4:

$F \rightarrow (. E)$

$E \rightarrow . E + T$

$E \rightarrow . T$

$T \rightarrow . T^* F$

$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . id$

GOTO (I4 , E) :

$F \rightarrow (E .)$

$E \rightarrow E . + T$ [New State: I8]

GOTO (I4 , T) :

$E \rightarrow T .$

$T \rightarrow T . ^* F$ [Existing State: I2]

GOTO (I4 , F) :

$T \rightarrow F .$ [Existing State: I3]

State I8:

$F \rightarrow (E .)$

$E \rightarrow E . + T$

State I2:

$E \rightarrow T .$

$T \rightarrow T . ^* F$

State I3:

$T \rightarrow F .$

Grammar

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

State I4:

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

GOTO (I4 , () :

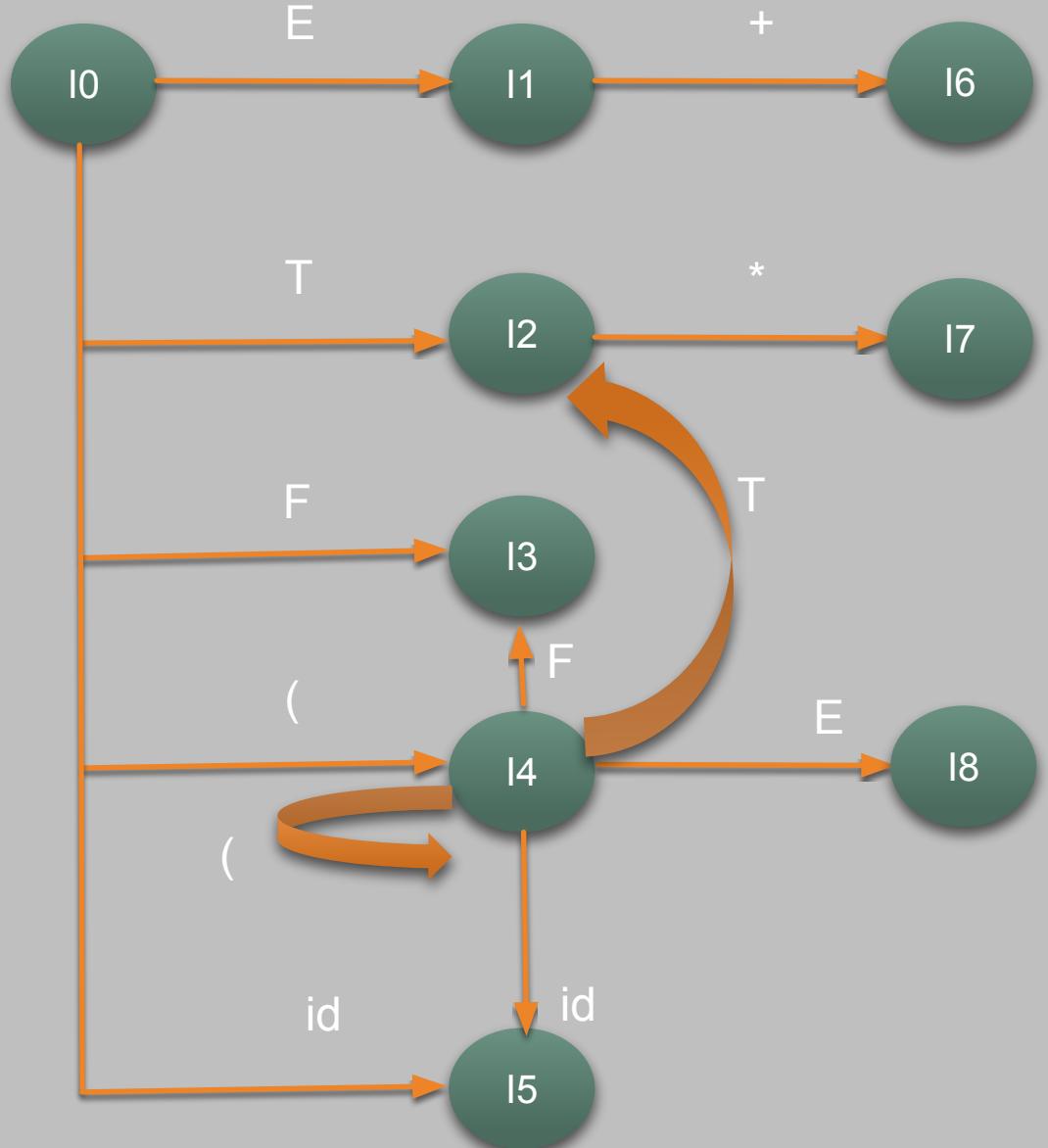
$$F \rightarrow (. E)$$
$$E \rightarrow . E + T$$
$$E \rightarrow . T$$
$$T \rightarrow . T^* F$$
$$T \rightarrow . F$$
$$F \rightarrow . (E)$$
$$F \rightarrow . Id \quad [Existing State: I4]$$

GOTO (I4 , id) :

$$F \rightarrow id . \quad [Existing State: I5]$$

State I5:

$$F \rightarrow id .$$



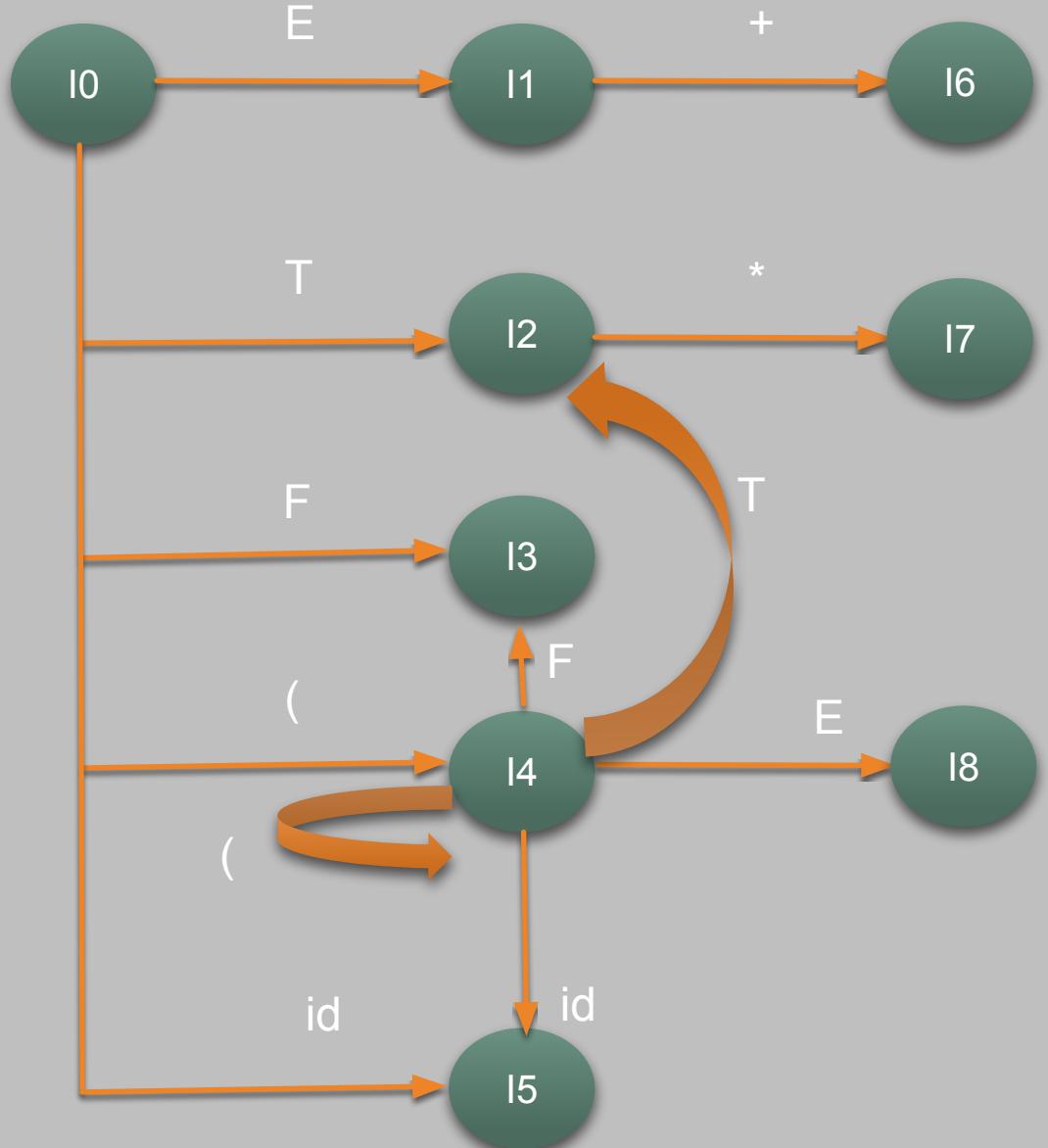
Grammar

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

No possible GOTO operation

State I5:

$F \rightarrow id .$



Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

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

$F \rightarrow (E) \mid id$

State I6:

$E \rightarrow E + . T$

$T \rightarrow . T^* F$

$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . id$

GOTO (I6 , T) :

$E \rightarrow E + T .$

$T \rightarrow T . * F \quad [New\ State: I9]$

State I9:

$E \rightarrow E + T .$

$T \rightarrow T . * F$

GOTO (I6 , F) :

$T \rightarrow F . \quad [Existing\ State: I3]$

State I3:

$T \rightarrow F .$

Grammar

$$E' \rightarrow E$$

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

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

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

State I6:

$$E \rightarrow E + . T$$

$$T \rightarrow . T^* F$$

$$T \rightarrow . F$$

$$F \rightarrow . (E)$$

$$F \rightarrow . id$$

GOTO (I6 , ()) :

$$F \rightarrow (. E)$$

$$E \rightarrow . E + T$$

$$E \rightarrow . T$$

$$T \rightarrow . T^* F$$

$$T \rightarrow . F$$

$$F \rightarrow . (E)$$

$$F \rightarrow . Id$$

[Existing State: I4]

State I4:

$$F \rightarrow (. E)$$

$$E \rightarrow . E + T$$

$$E \rightarrow . T$$

$$T \rightarrow . T^* F$$

$$T \rightarrow . F$$

$$F \rightarrow . (E)$$

$$F \rightarrow . id$$

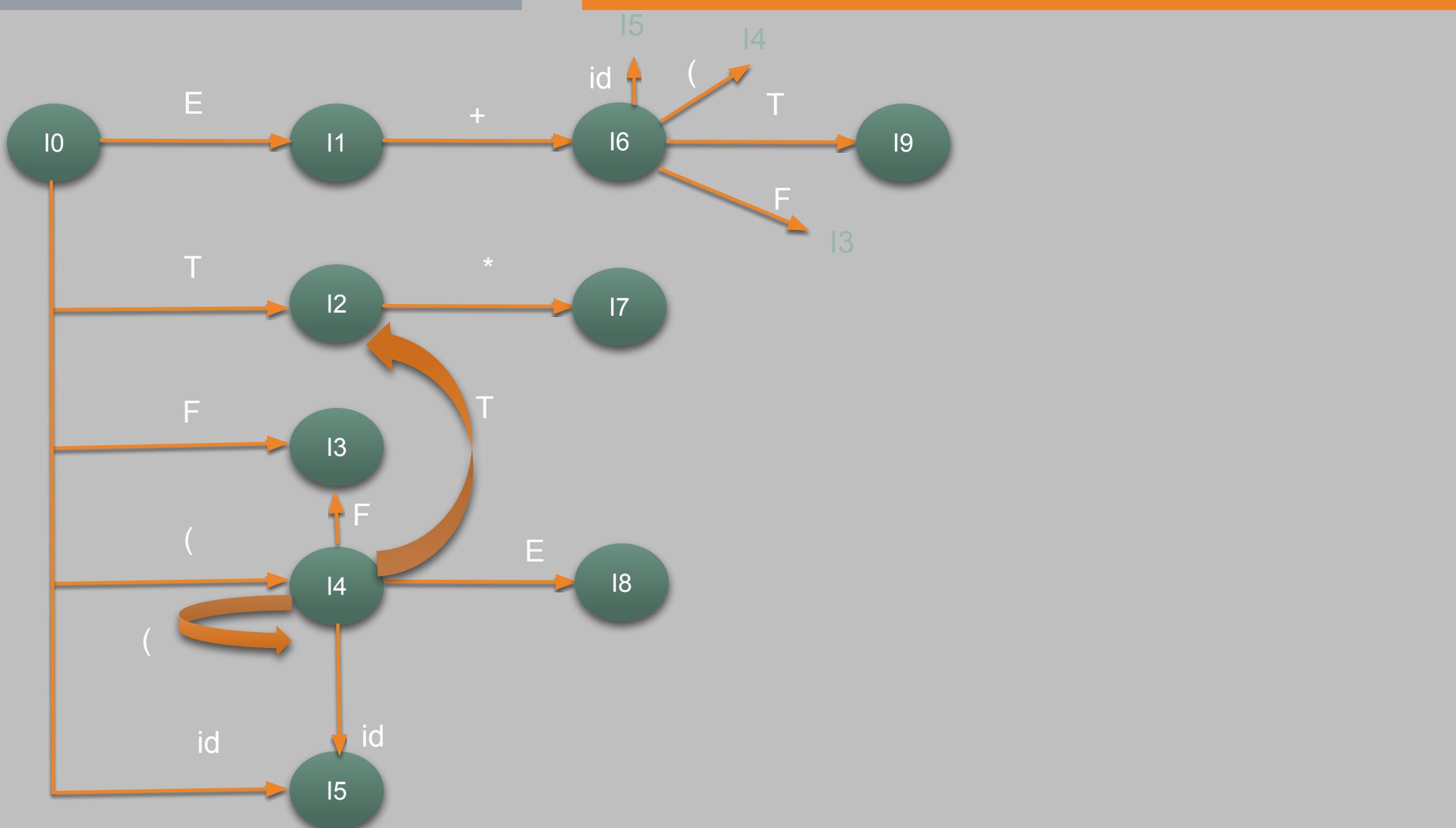
GOTO (I6 , id) :

$$F \rightarrow id .$$

[Existing State: I5]

State I5:

$$F \rightarrow id .$$



Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

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

$F \rightarrow (E) \mid id$

State I7:

$T \rightarrow T^* . F$

$F \rightarrow . (E)$

$F \rightarrow . id$

GOTO (I7 , F) :

$T \rightarrow T^* F .$ [New State: I10]

State I10:

$T \rightarrow T^* F .$

GOTO (I7 , ()) :

$F \rightarrow (. E)$

$E \rightarrow . E + T$

$E \rightarrow . T$

$T \rightarrow . T^* F$

$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . Id$ [Existing State: I4]

State I4:

$F \rightarrow (. E)$

$E \rightarrow . E + T$

$E \rightarrow . T$

$T \rightarrow . T^* F$

$T \rightarrow . F$

$F \rightarrow . (E)$

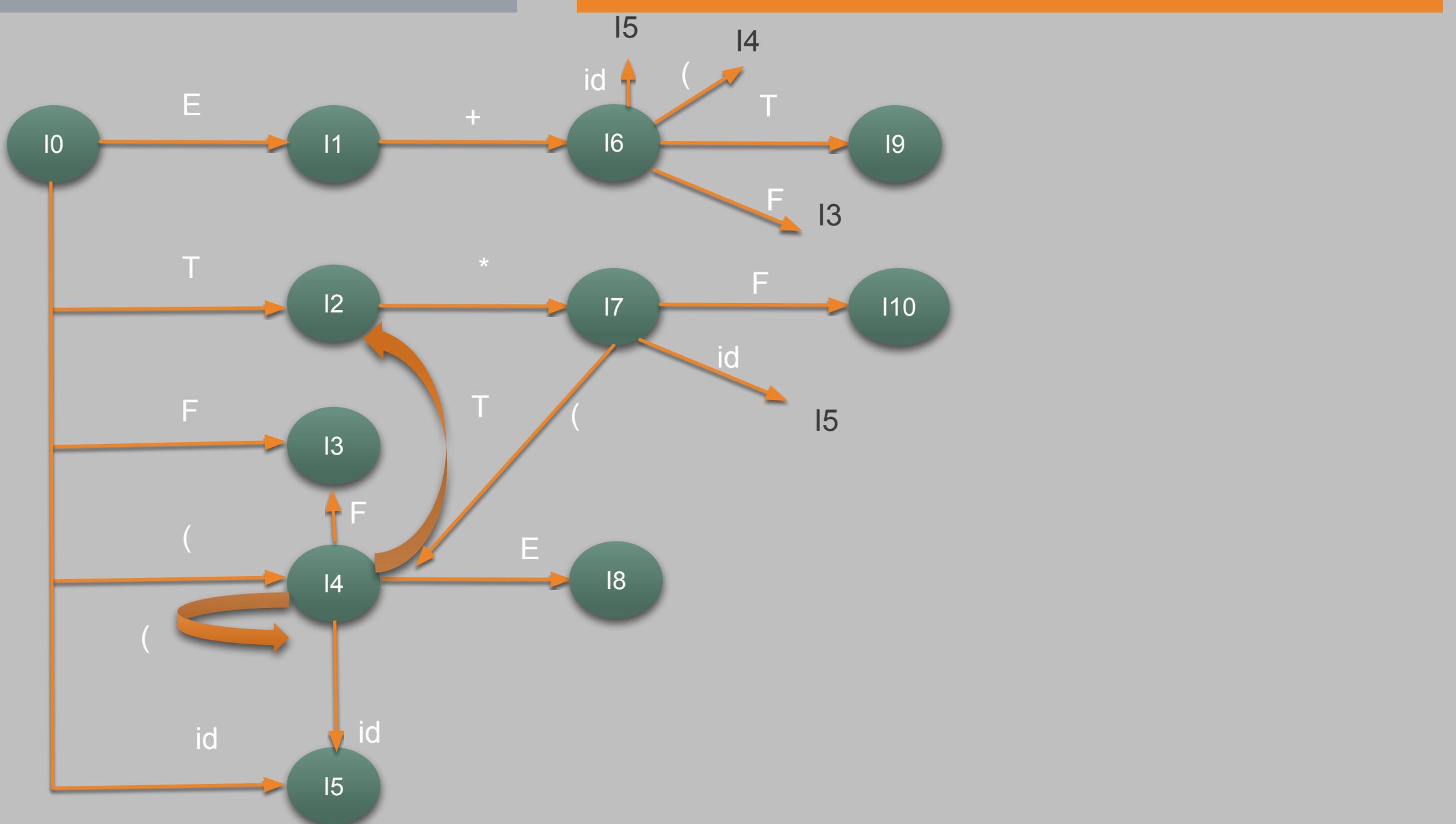
$F \rightarrow . id$

GOTO (I7 , id) :

$F \rightarrow id .$ [Existing State: I5]

State I5:

$F \rightarrow id .$



Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

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

$F \rightarrow (E) \mid id$

State I8:

$F \rightarrow (E .)$

$E \rightarrow E . + T$

GOTO (I8 ,) :

$F \rightarrow (E) .$

[New State: I11]

State I11:

$F \rightarrow (E) .$

GOTO (I8 , +) :

$E \rightarrow E + . T$

$T \rightarrow . T^* F$

$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . id$

[Existing State: I6]

State I6:

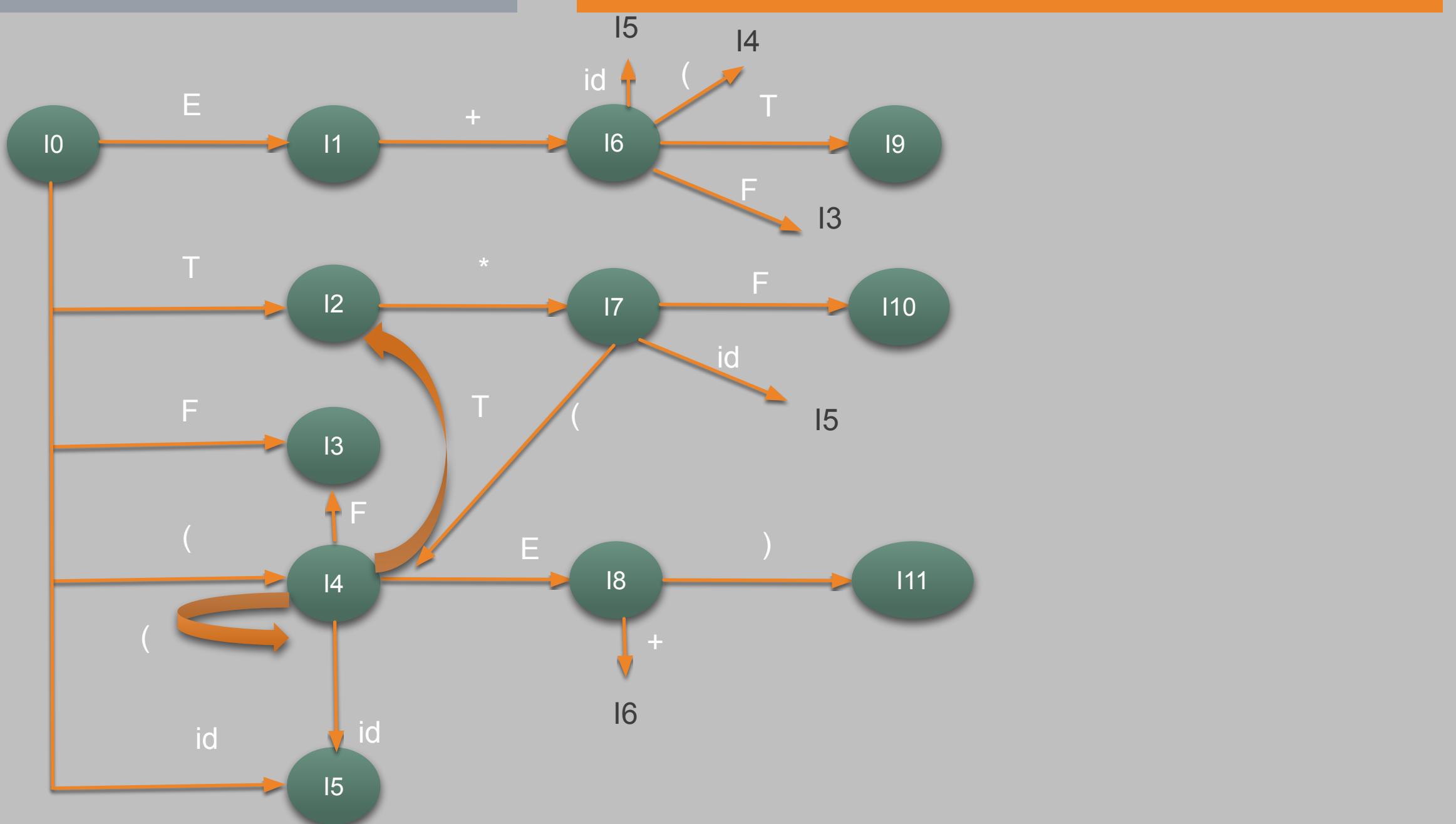
$E \rightarrow E + . T$

$T \rightarrow . T^* F$

$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . id$



Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

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

$F \rightarrow (E) \mid id$

State I9:

$F \rightarrow E + T .$

$T \rightarrow T . * F$

GOTO (I9 , *) :

$T \rightarrow T^* . F$

$F \rightarrow . (E)$

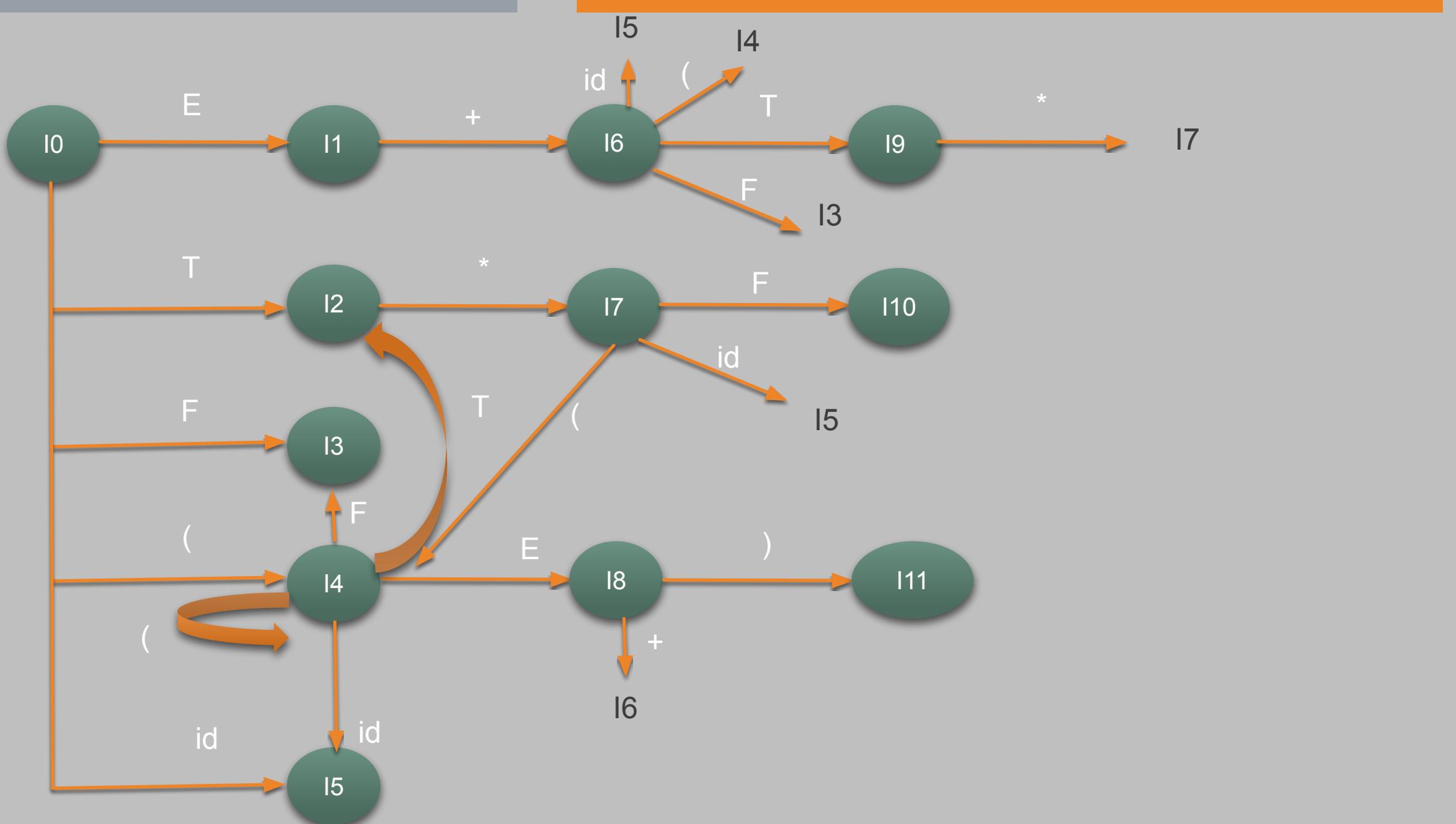
$F \rightarrow . id$ [Existing State: I7]

State I7:

$T \rightarrow T^* . F$

$F \rightarrow . (E)$

$F \rightarrow . id$



Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

No possible GOTO operation

State I10:

$T \rightarrow T * F .$

Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

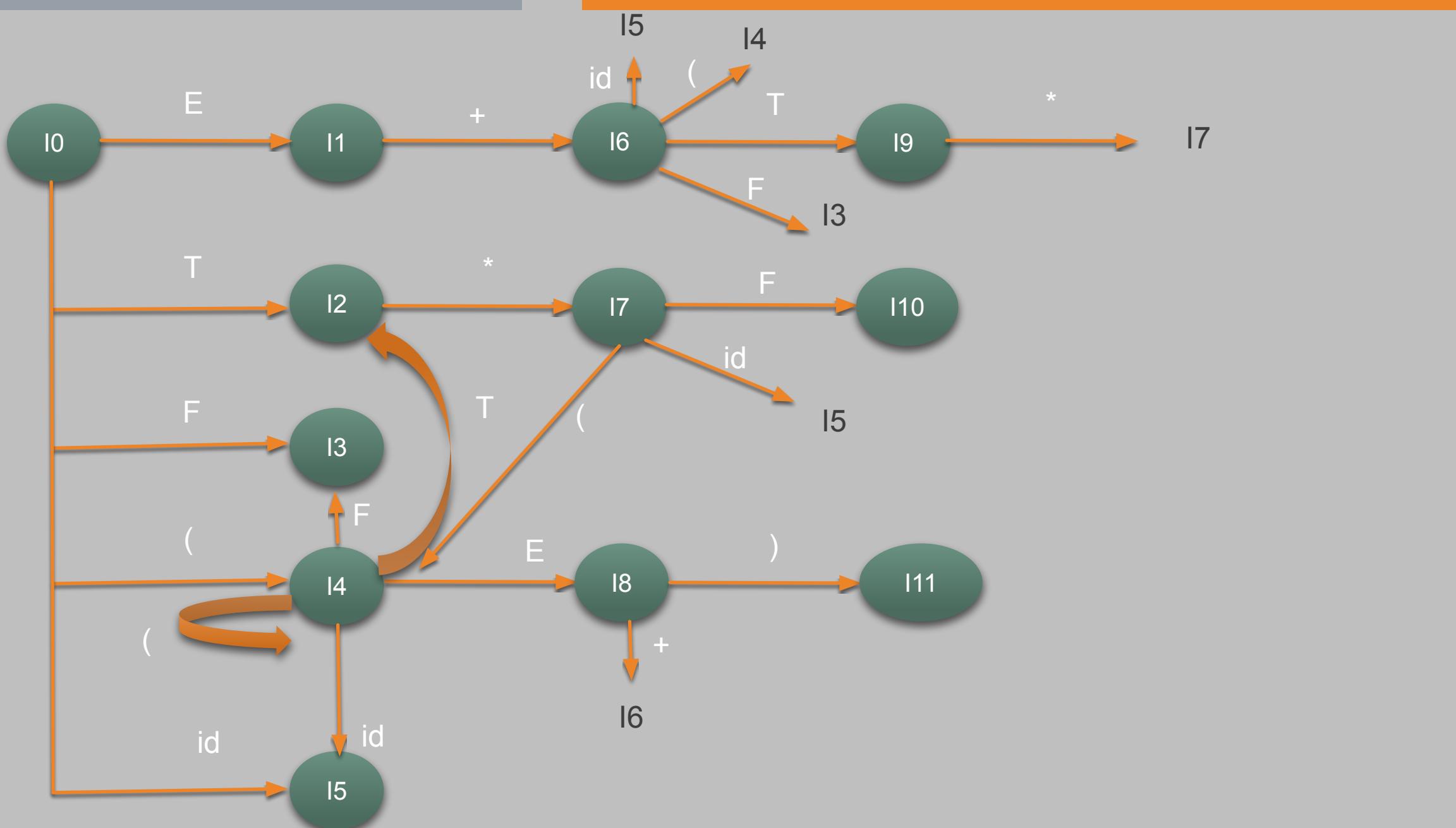
$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

No possible GOTO operation

State I11:

$F \rightarrow (E) .$



Construction of SLR Parsing Table

Input:

An augmented Grammar G'

Output:

The SLR Parsing Table with functions ACTION and GOTO for G'

Construction of SLR Parsing Table

Method:

1. Construct $C = \{ I_0, I_1, \dots, I_n \}$ the collection of sets of LR (0) items for G'
2. State i is constructed from I_i .

The parsing action for state i are determined as follows:

- a. If $[A \rightarrow \alpha . a \beta]$ is in I_i and $\text{GOTO} (I_i, a) = I_j$ then
Set Action $[i, a]$ to "**Shift j**"
Here a must be terminal
- b. If $[A \rightarrow \alpha .]$ then
Set Action $[i, a]$ to "**Reduce $A \rightarrow \alpha$** "
For all a in $\text{FOLLOW}(A)$
- c. If $[S' \rightarrow S .]$ is in I_i then
Set Action $[i, \$]$ to "**Accept**"

If any conflicting actions are generated by the above rules
Then grammar is not SLR (1)

Construction of SLR Parsing Table

Method:

3. The GOTO transitions for state i are constructed for all non-terminals A using the rule

If $\text{GOTO} [i , A] = l_j$ then
Set $\text{GOTO} [i , A] = j$

4. All entries not defined by rules (2) and (3) are made "ERROR"

5. The initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow . S]$

Construction of SLR Parsing Table

Given Grammar:

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

Construction of SLR Parsing Table

Given Grammar:

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

State 10:

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

Construction of SLR Parsing Table

Given Grammar:

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

State I1:

$$E' \rightarrow E.$$

$$E \rightarrow E_+ + T$$

16

Construction of SLR Parsing Table

Given Grammar:

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

State I2:

$E \rightarrow T .$
 $T \rightarrow T . * F$



State	Action						GOTO		
	id	+	*	()	\$	E	T	F
0	s5				s4				
1			s6				Accept		
2		r2	s7			r2	r2		
3									
4									
5									
6									
7									
8									
9									
10									
11									

Given Grammar:

$\text{FOLLOW}(E) = \{ +,), \$ \}$

$\text{FOLLOW}(T) = \{ *, +,), \$ \}$

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

Construction of SLR Parsing Table

Given Grammar:

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

State I3: $T \rightarrow F$.

Given Grammar:

$\text{FOLLOW}(E) = \{ +, , \}, \$ \}$

$$\text{FOLLOW}(T) = \{ *, +,), \$ \}$$

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

Construction of SLR Parsing Table

Given Grammar:

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

State 14:

F → (. E)

$$E \rightarrow E + T$$

E → T

T → T * E

T E

EN 100

Environ Biol Fish

Construction of SLR Parsing Table

Given Grammar:

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

State I5:
 $F \rightarrow id .$

State	Action						GOTO		
	id	+	*	()	\$	E	T	F
0	s5				s4				
1			s6				Accept		
2		r2	s7			r2	r2		
3		r4	r4			r4	r4		
4	s5				s4				
5		r6	r6			r6	r6		
6									
7									
8									
9									
10									
11									

Given Grammar:

$\text{FOLLOW}(E) = \{ +, \), \$ \}$

$\text{FOLLOW}(T) = \{ *, +, \), \$ \}$

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

Construction of SLR Parsing Table

Given Grammar:

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

State 16:

E → E + . T

$$T \rightarrow T^* F$$

T → F

F → (F)

E → id

Construction of SLR Parsing Table

Given Grammar:

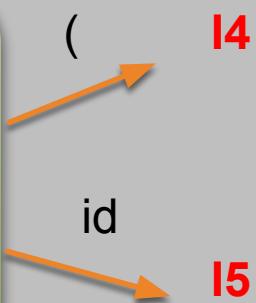
1. $E \rightarrow E + T$
 2. $E \rightarrow T$
 3. $T \rightarrow T * F$
 4. $T \rightarrow F$
 5. $F \rightarrow (E)$
 6. $F \rightarrow \text{id}$

State 17:

T → T *, F

F → . (E)

$F \rightarrow . id$



Construction of SLR Parsing Table

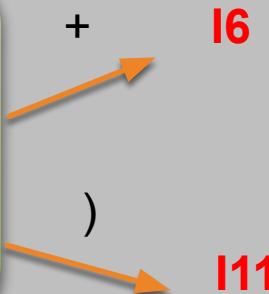
Given Grammar:

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

State 18:

F → (E.)

$$E \rightarrow E_{\gamma} + T$$



Construction of SLR Parsing Table

Given Grammar:

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

State 19:

$$E \rightarrow E + T.$$

T → T₁*F

17

Given Grammar:

$$\text{FOLLOW}(E) = \{ +,), \$ \}$$

$$\text{FOLLOW}(T) = \{ *, +,), \$ \}$$

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

Construction of SLR Parsing Table

Given Grammar:

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

State I10:

$$T \rightarrow T^* F.$$

Given Grammar:

$$\text{FOLLOW}(E) = \{ +,), \$ \}$$

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

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

Construction of SLR Parsing Table

Given Grammar:

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

State I11:
 $F \rightarrow (E) .$

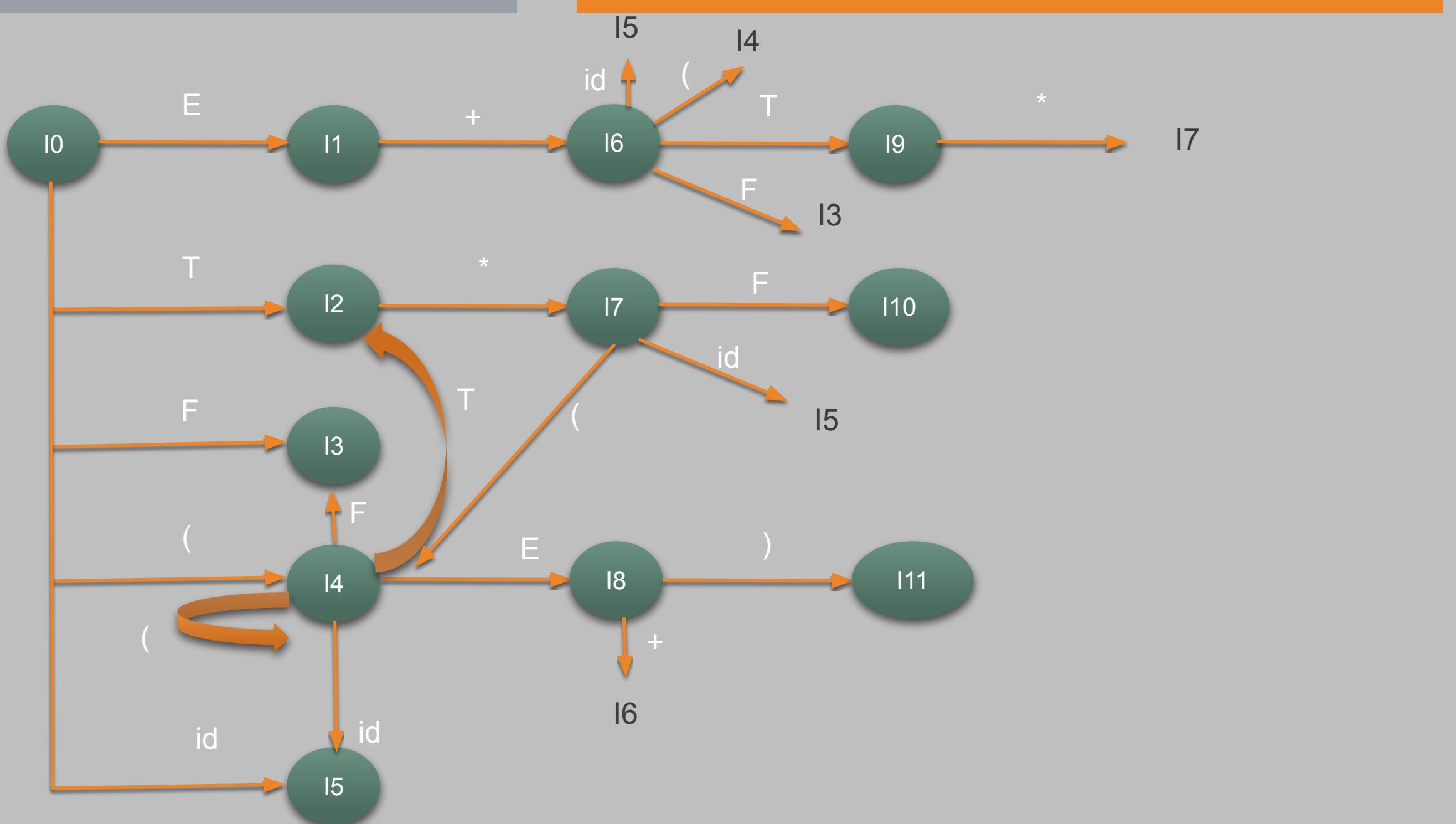
State	Action						GOTO		
	id	+	*	()	\$	E	T	F
0	s5				s4				
1		s6					Accept		
2		r2	s7			r2	r2		
3		r4	r4			r4	r4		
4	s5				s4				
5		r6	r6			r6	r6		
6	s5				s4				
7	s5				s4				
8		s6				s11			
9		r1	s7			r1	r1		
10		r3	r3			r3	r3		
11		r5	r5			r5	r5		

Given Grammar:

$$\text{FOLLOW}(E) = \{ +,), \$ \}$$

$$\text{FOLLOW}(T) = \{ *, +,), \$ \}$$

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



Construction of SLR Parsing Table

Given Grammar:

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

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				
5		r6	r6			r6	r6		
6	s5				s4				
7	s5				s4				
8		s6				s11			
9		r1	s7			r1	r1		
10		r3	r3			r3	r3		
11		r5	r5			r5	r5		

Construction of SLR Parsing Table

Given Grammar:

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

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					
7	s5			s4					
8		s6				s11			
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Construction of SLR Parsing Table

Given Grammar:

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

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
7	s5			s4					
8		s6				s11			
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Construction of SLR Parsing Table

Given Grammar:

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

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
7	s5			s4					10
8		s6				s11			
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Construction of SLR Parsing Table

Given Grammar:

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

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
7	s5			s4					10
8		s6				s11			
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

SLR Parsing Algorithm

Input:

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

Output:

If w is in $L(G)$, the reduction steps of bottom-up parse for w ; otherwise, there is an error

Method:

Initially, the parser has s_0 on the stack where s_0 is the initial state and $w\$$ is in the input buffer.

Execute the following Algorithm

SLR Parsing Algorithm

```
let a be the first symbol of w$;
while(1)                                /* repeat forever */
{
    let s be the state on top of the stack;
    if ( ACTION [ s , a ] = shift t )
    {
        push t onto the stack;
        let a be the next input symbol;
    }
    else if ( ACTION [ s , a ] = reduce A → β )
    {
        pop | β | symbols of the stack;
        let state t now be on top of the stack;
        push GOTO [ t , A] onto the stack;
        output the production A → β ;
    }
    else if ( ACTION [ s , a ] = accept )
        break; /* parsing is done */
    else call error-recovery routine;
}
```

Stack	Symbols	Input	Action	Output
0		id * id + id \$	Shift	
0 5	id	* id + id \$	Reduce by F → id Pop '5' and check GOTO (0,F)	F → id
0 3	F	* id + id \$	Reduce by T → F Pop '3' and check GOTO (0,T)	T → F
0 2	T	* id + id \$	Shift	
0 2 7	T *	id + id \$	Shift	
0 2 7 5	T * id	+ id \$	Reduce by F → id Pop '5' and check GOTO (7,F)	F → id
0 2 7 10	T * F	+ id \$	Reduce by T → T * F Pop '10','7','2' and check GOTO (0,T)	T → T * F
0 2	T	+ id \$	Reduce by E → T Pop '2' and check GOTO (0,E)	E → T
0 1	E	+ id \$	Shift	
0 1 6	E +	id \$	Shift	
0 1 6 5	E + id	\$	Reduce by F → id Pop '5' and check GOTO (6,F)	F → id
0 1 6 3	E + F	\$	Reduce by T → F Pop '3' and check GOTO (6,T)	T → F
0 1 6 9	E + T	\$	Reduce by E → E + T Pop '9', '6', '1' and check GOTO (0,E)	E → E + T
0 1	E	\$	Accept	

Module 5

Compilers: Analysis Phase



Semantic analysis

- .Syntax directed definitions

Semantic Analysis

- . Parsing only verifies that the program consists of tokens arranged in a syntactically valid combination.
- . Semantic analysis checks whether they form a sensible set of instructions in the programming language.
- . For a program to be semantically valid, all variables, functions, classes, etc. must be properly defined, expressions and variables must be used in proper ways

Semantic Analysis

- .A large part of semantic analysis consists of tracking variable/function/type declarations and type checking
- .**Type checking** is the process of verifying that each operation executed in a program respects the type system of the language.
- .This means that all operands in any expression are of appropriate types and number.

Semantic Analysis

- .A language is considered *strongly typed* if each and every type error is detected during compilation.
- .Type checking can be done compilation, during execution, or divided across both

Semantic Analysis

- . **Static type checking** is done at compile time.
- . The information the type checker needs is obtained via declarations and stored in a master symbol table.
- . After this information is collected, the types involved in each operation are checked.
- . It is very difficult for a language that only does static type checking to meet the full definition of strongly typed.

Semantic Analysis

- .**Dynamic type checking** is implemented by including type information for each data location at runtime.
- .Dynamic type checking clearly comes with a runtime performance penalty, but it can report errors that are not possible to detect at compile time

Semantic Analysis

. Translation techniques like **Syntax Directed Definition (SDD)** is applied to type checking and intermediate-code generation.

Syntax Directed Definitions(SDD)

- .We can associate information with a language construct by attaching attributes to the grammar symbols.
- .A syntax directed definition specifies the values of attributes by associating semantic rules with the grammar productions.
- .Attributes are associated with grammar symbols and rules are associated with productions
- .Attributes may be of many kinds: numbers, types, table references, strings, etc.

Syntax Directed Definitions(SDD)

- If X is a symbol and a is one of its attributes, then we write $X.a$ to denote the value of a at a particular parse-tree node labeled X .
- If we implement the nodes of the parse tree by records or objects, then the attributes of X can be implemented by data fields in the records that represent the nodes for X .

SDD for a simple desk calculator

Production	Semantic Rules
$L \rightarrow E \ n$	Print (E.val)
$L \rightarrow E_1 + T$	E.val = E1.val + T.val
$E \rightarrow T$	E.val = T.val
$T \rightarrow T_1 * F$	T.val = T1.val * F.val
$T \rightarrow F$	T.val = F.val
$F \rightarrow (E)$	F.val = E.val
$F \rightarrow \text{digit}$	F.val = digit.lexval

Syntax Directed Definitions(SDD)

- Two Types:
 - Synthesized Attributes
 - Inherited Attributes

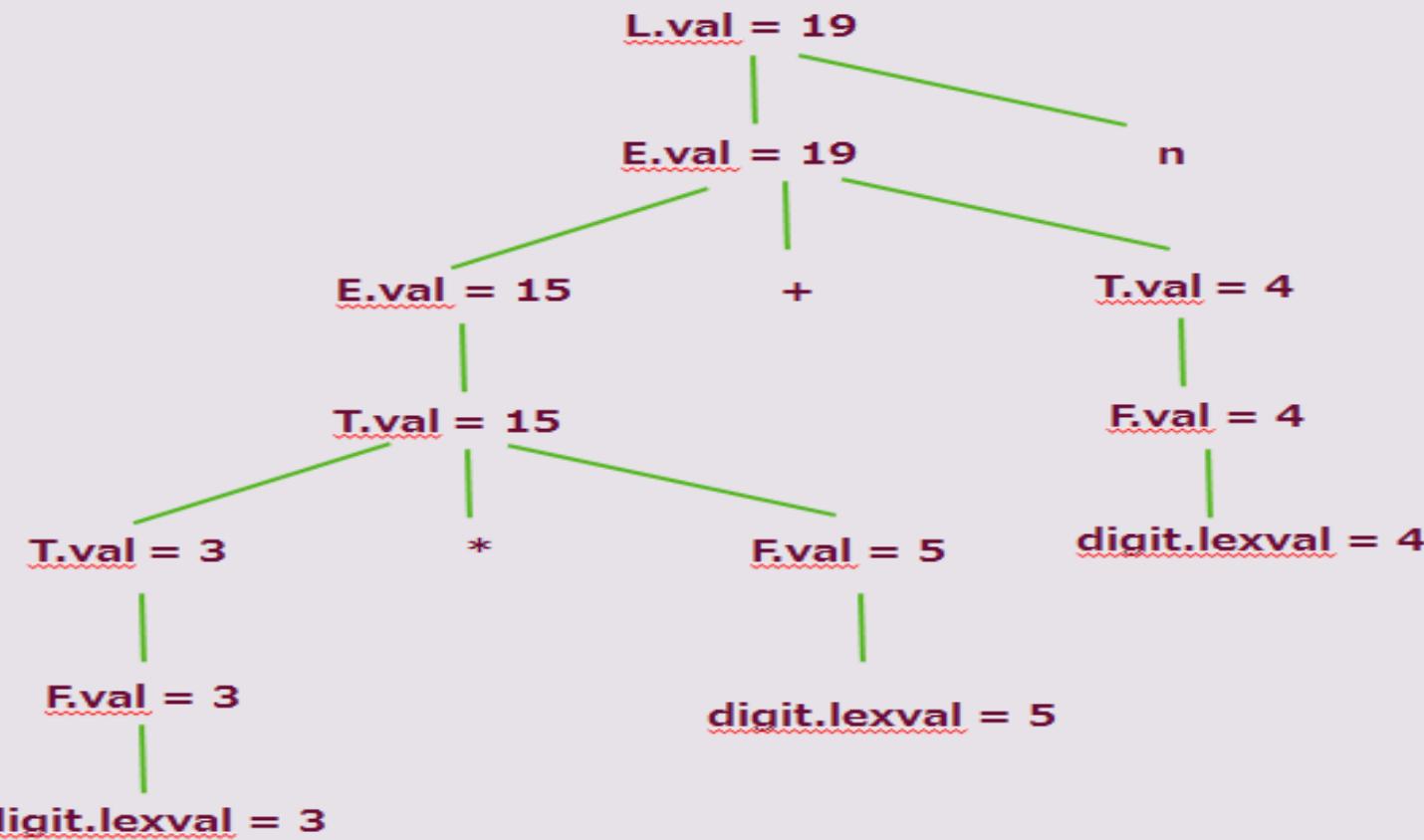
Synthesized Attributes

- .A synthesized attribute for a non terminal A at a parse tree node N is defined by a semantic rule associated with the production at N.
- .The production must have A as its head.
- .A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.
- .An syntax directed definition that uses Synthesized attributes exclusively is said to be an S-attributed definition.

Synthesized Attributes

- .A parse tree for an S-attributed definition can be annotated (interpreted) by evaluating semantic rules for attributes at each node
- .To implement S-attributed definition LR parser can be used i.e. bottom up parsing is used

Synthesized Attributes



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

Inherited Attributes

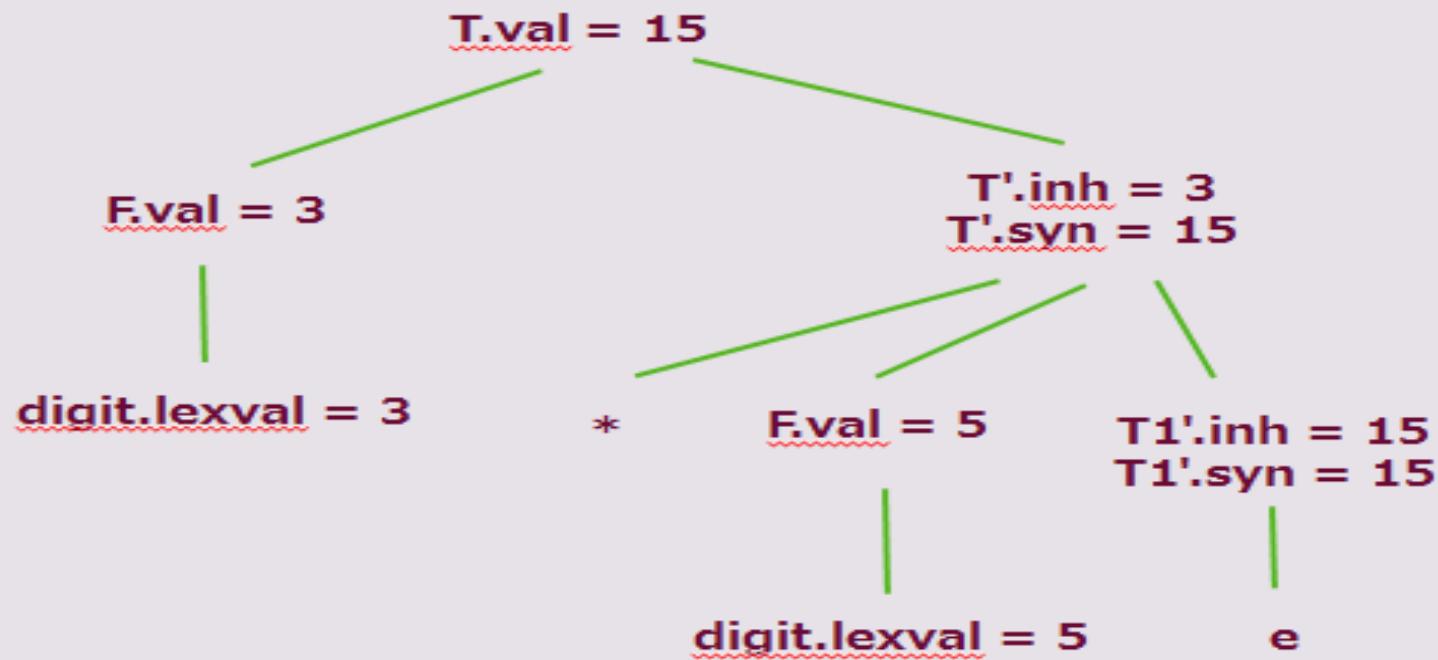
- .An inherited attribute for a non terminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N.
- .The production must have B as a symbol in its body.
- .An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself and N's siblings.
- .Inherited attributes are convenient for expressing dependence of a program language construct on the context in which appears

Inherited Attributes

Production	Semantic Rules
$T \rightarrow F T'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \rightarrow * F T1'$	$T1'.inh = T'.inh * F.val$ $T'.syn = T1'.syn$
$T' \rightarrow e$	$T'.syn = T'.inh$
$F \rightarrow digit$	$F.val = digit.lexval$

SDD with inherited attribute L.in

Inherited Attributes

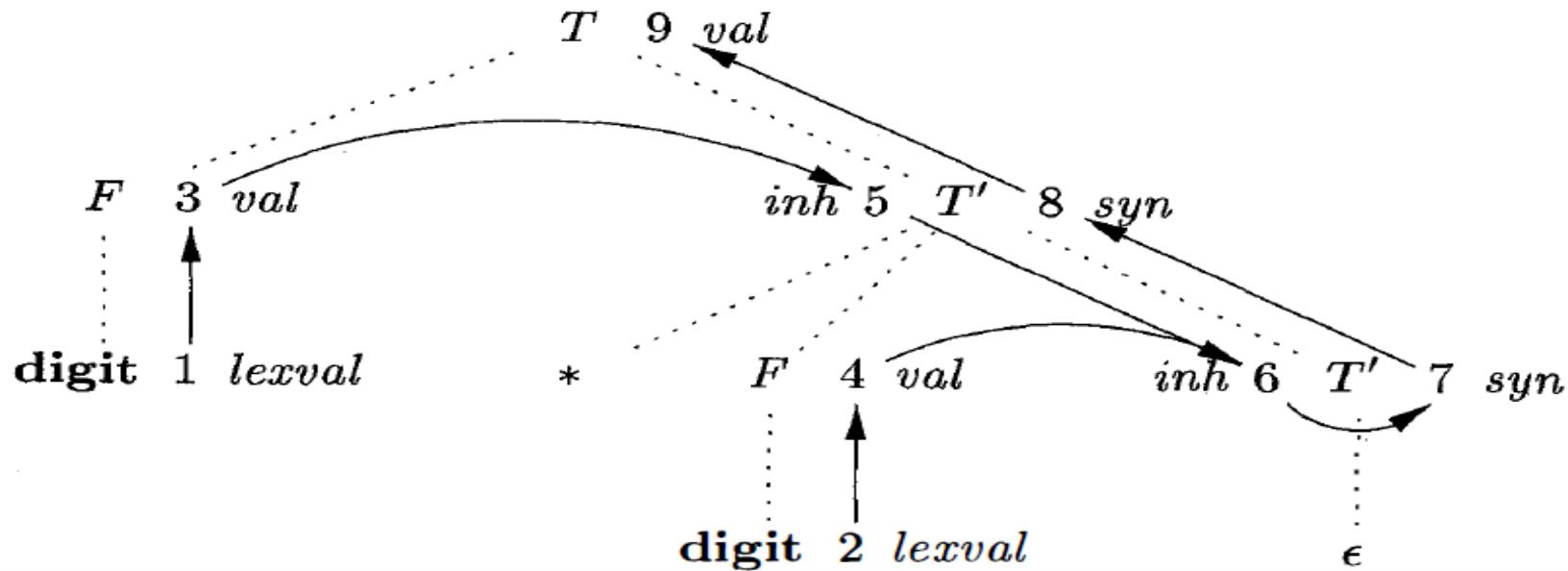


Annotated Parse Tree for $3 * 5$

Evaluation Order for SDDs

- .**Dependency graphs** are a useful tool for determining an evaluation order for the attribute instances in a given parse tree.
- .While an annotated parse tree shows the values of attributes, a dependency graph helps us determine how those values can be computed.

Dependency Graph



- .A dependency graph shows the flow of information among the attribute instances in a particular parse tree;
- .An edge from one attribute instance to another means that the value of the first is needed to compute the second. Edges express constraints implied by the semantic rules.

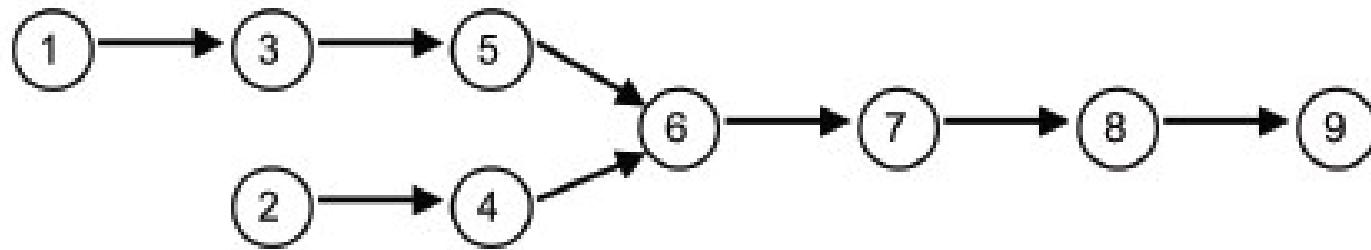
Topological Sort of Dependency Graph

- A dependency graph characterizes the possible order in which we can evaluate the attributes at various nodes of a parse tree.
- If there is an edge from node M to N, then attribute corresponding to M is first to be evaluated before evaluating N.
- Thus the only allowable orders of evaluation are N_1, N_2, \dots, N_k such that if there is an edge from N_i to N_j then $i < j$.

Topological Sort of Dependency Graph

- Such an ordering embeds a directed graph into a linear order, and is called a topological sort of the graph.
- If there is any cycle in the graph, then there are no topological sorts; that is, there is no way to evaluate the SDD on this parse tree.
- If there are no cycles, however, then there is always at least one topological sort.

Topological Sort of Dependency Graph



Base sequences are {1 3 5} and {2 4} with the suffix {6 7 8 9} being constant as the last nodes of the topological sorting need to remain fixed.

1 3 5 2 4 6 7 8 9, 1 3 2 5 4 6 7 8 9, 1 2 3 5 4 6 7 8 9,

1 3 2 4 5 6 7 8 9, 1 2 3 4 5 6 7 8 9, 1 2 4 3 5 6 7 8 9,

2 1 3 5 4 6 7 8 9, 2 1 3 4 5 6 7 8 9, 2 1 4 3 5 6 7 8 9,

2 4 1 3 5 6 7 8 9

S-attributed Definition

- Uses only synthesized attributes
- Semantic actions are placed at right end of the production
- Also called as postfix SDT
- Attributes are evaluated during bottom up parsing

L-attributed Definition

- Uses both inherited and synthesized attributes.
- Each inherited attribute is restricted to inherit either from parent or left sibling
- Semantic actions are placed anywhere on the RHS
- Attributes are evaluated by traversing parse tree depth first and left to right

COMPILER

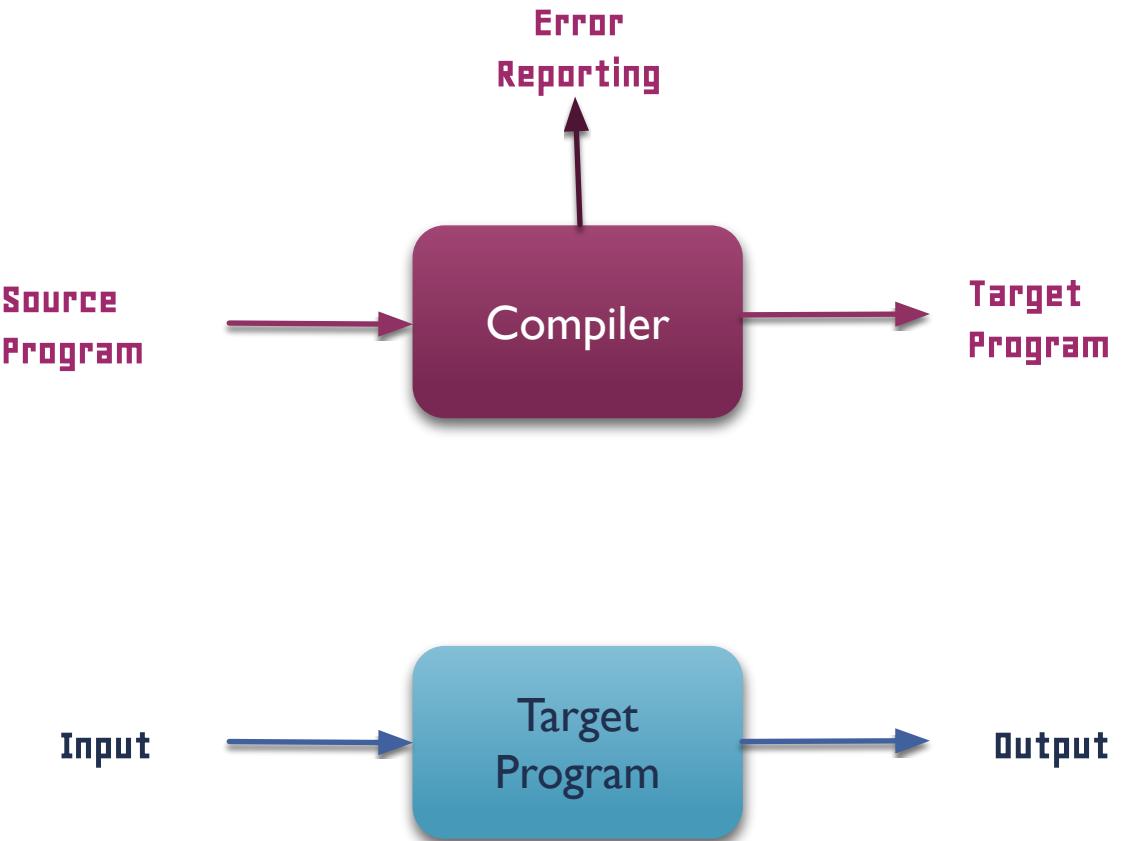
MODULE 5

CONTENT

- **Introduction to Compiler**
- **Language Processing System**
- **Phases of Compiler**
- **Example**

COMPILER

- Language Processor
- Translates program from one language [Source Language] to another language [Target Language]
- Reports error if any
- Target program can be executed directly if it is executable machine code



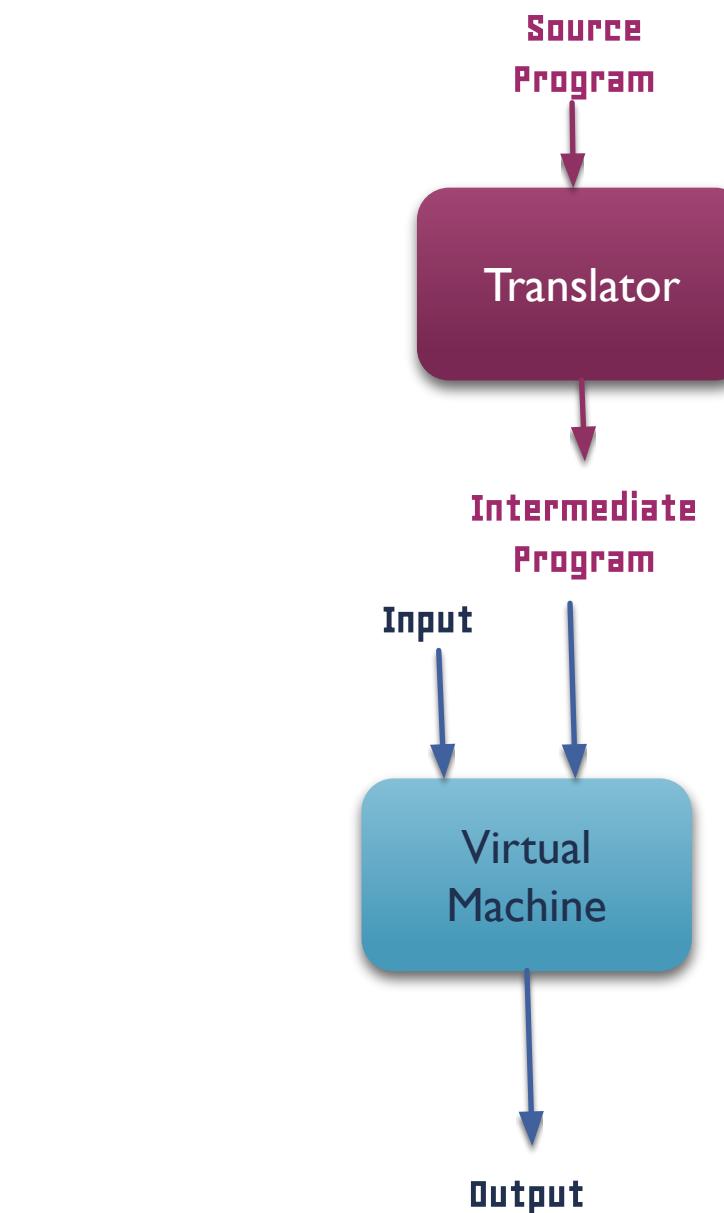
INTERPRETER

- Language Processor
- Translates code line by line and execute it immediately
- Slow execution compared to Compiler
- Better error diagnostic

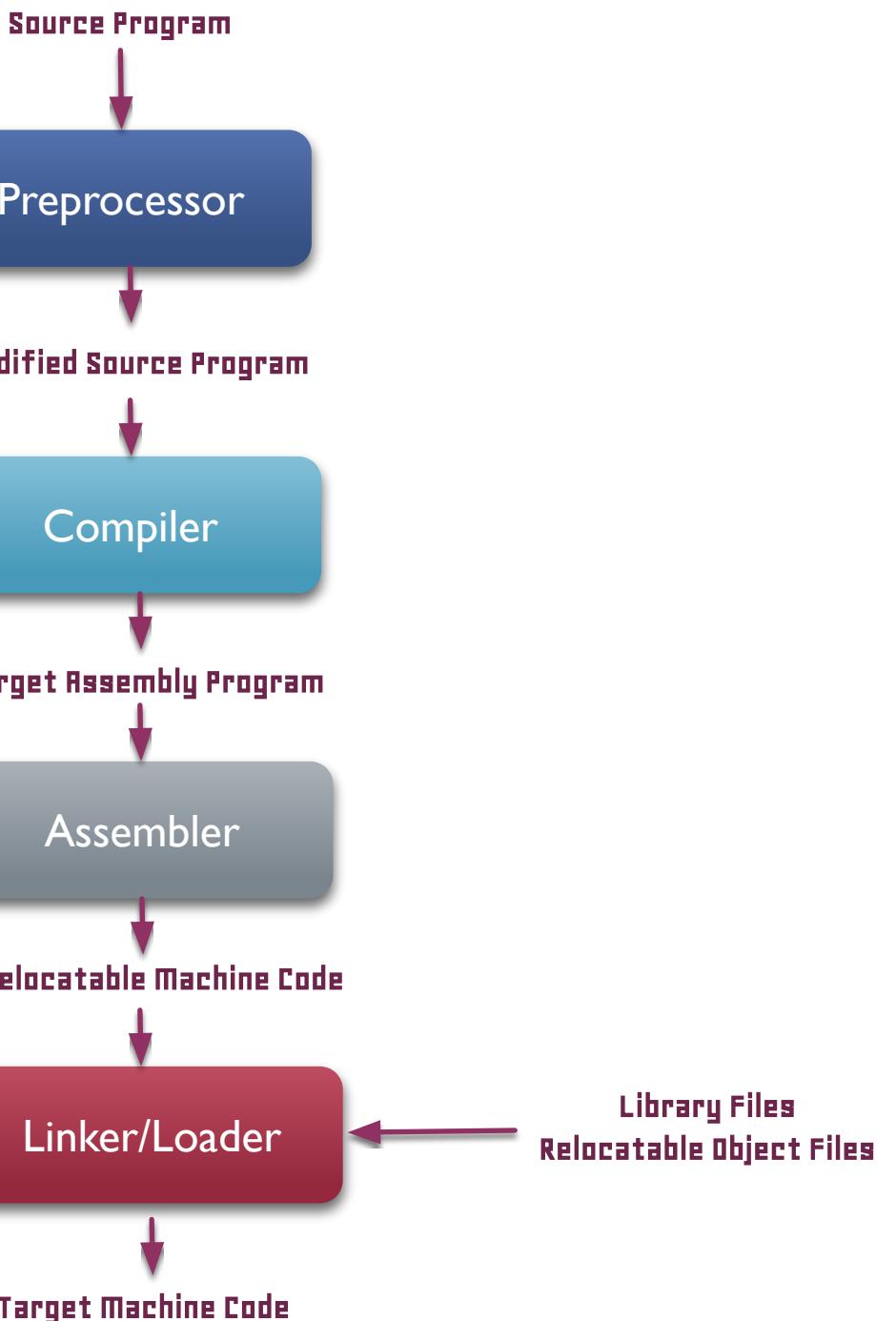


HYBRID COMPILER

- Language Processor
- Combines both compilation and interpretation
- E.g. Java Language Processor
- Java source program is compiled into Bytecode
- Bytecode can be interpreted on any machine
- Just In Time Compiler: Translate byte code into machine code immediately

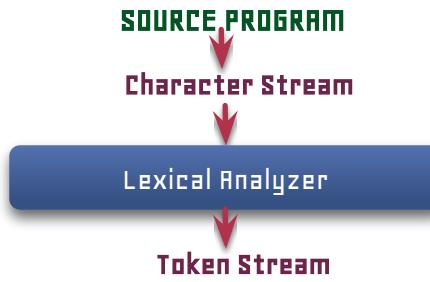


LANGUAGE PROCESSING SYSTEM



PHASES OF COMPILER

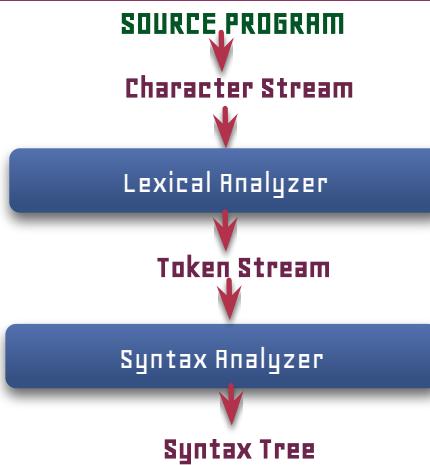
PHASES OF COMPILER



Lexical Analyzer

- Lexical analysis is also called linear analysis or scanning
- It processes character from left to right & group the character stream into significant unit called lexemes
- These lexemes are mapped into token
- It also does the additional job of removing extra whitespace, comments added by the user, etc. from the program

PHASES OF COMPILER



Syntax Analyzer

- The syntax analysis phase takes word/tokens from the lexical analyzer and checks the syntactic correctness of the input program
- Syntax analysis identifies and notifies the syntactic errors in the program once the program has been broken into the tokens
- It often referred as hierarchical analysis or simply parsing
- The hierarchical structure of the source program can be represented by a parse tree

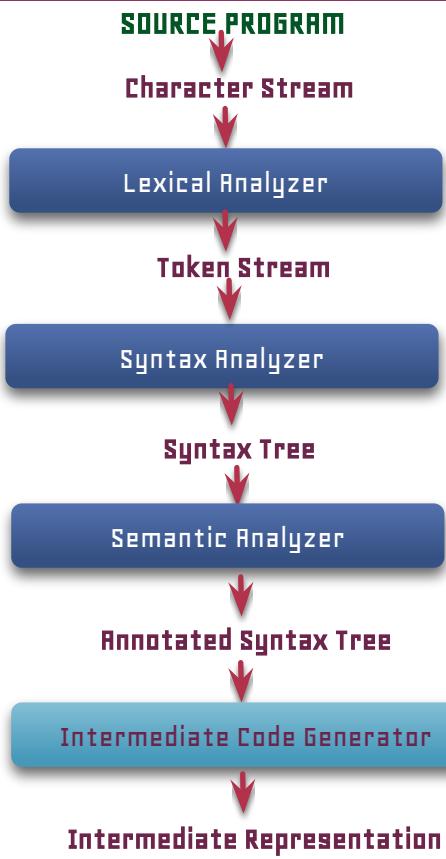
PHASES OF COMPILER



Semantic Analyzer

- Once the program is syntactically correct means grammatically correct the next task is to check semantic correctness
- This phase performs the checks on the meaning of the statement and makes the necessary modification in the parse tree representation
- The word semantic refer to meaning & the semantic analyzer checks the meaning of the program
- This phase checks the source program for semantic error and gathers "type information" about the program element

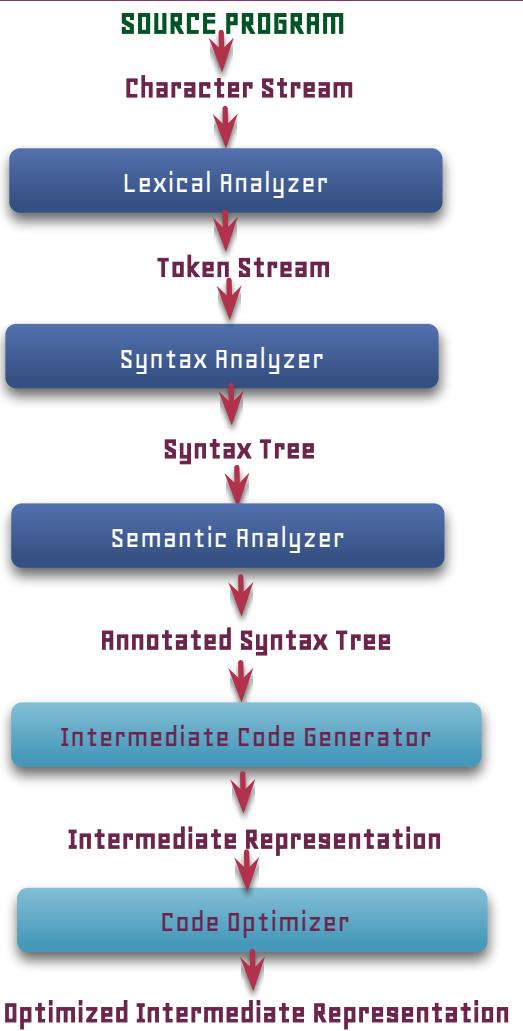
PHASES OF COMPILER



Intermediate Code Generator

- This phase generates an intermediate code which helps to simplify the complexity of the code
- The intermediate representation should have two important properties:-
 - It should be easy to produce
 - It should be easy to translate into the target program

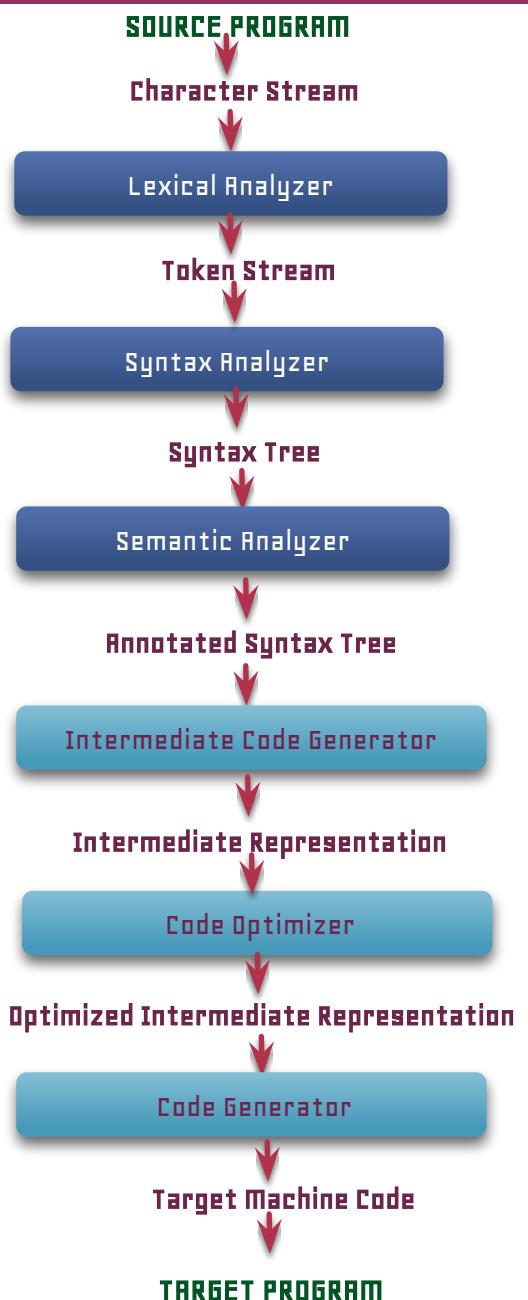
PHASES OF COMPILER



Code Optimizer

- The code optimization phase attempts to improve the intermediate code so that a faster running machine code could be generated
- Code optimization is performed to:
 - Minimize the time taken to execute a program
 - Minimize the amount of memory occupied

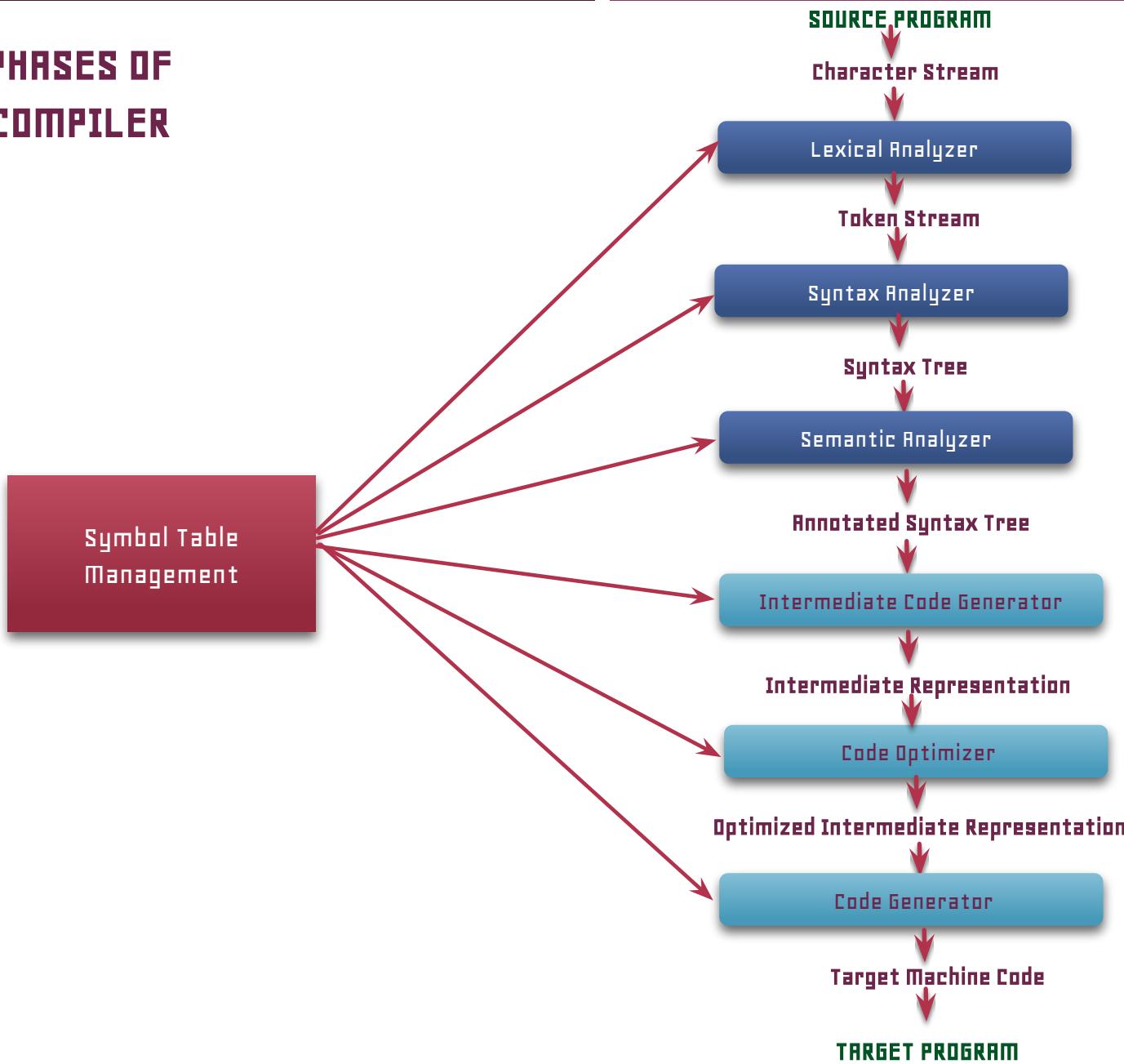
PHASES OF COMPILER



Code Generator

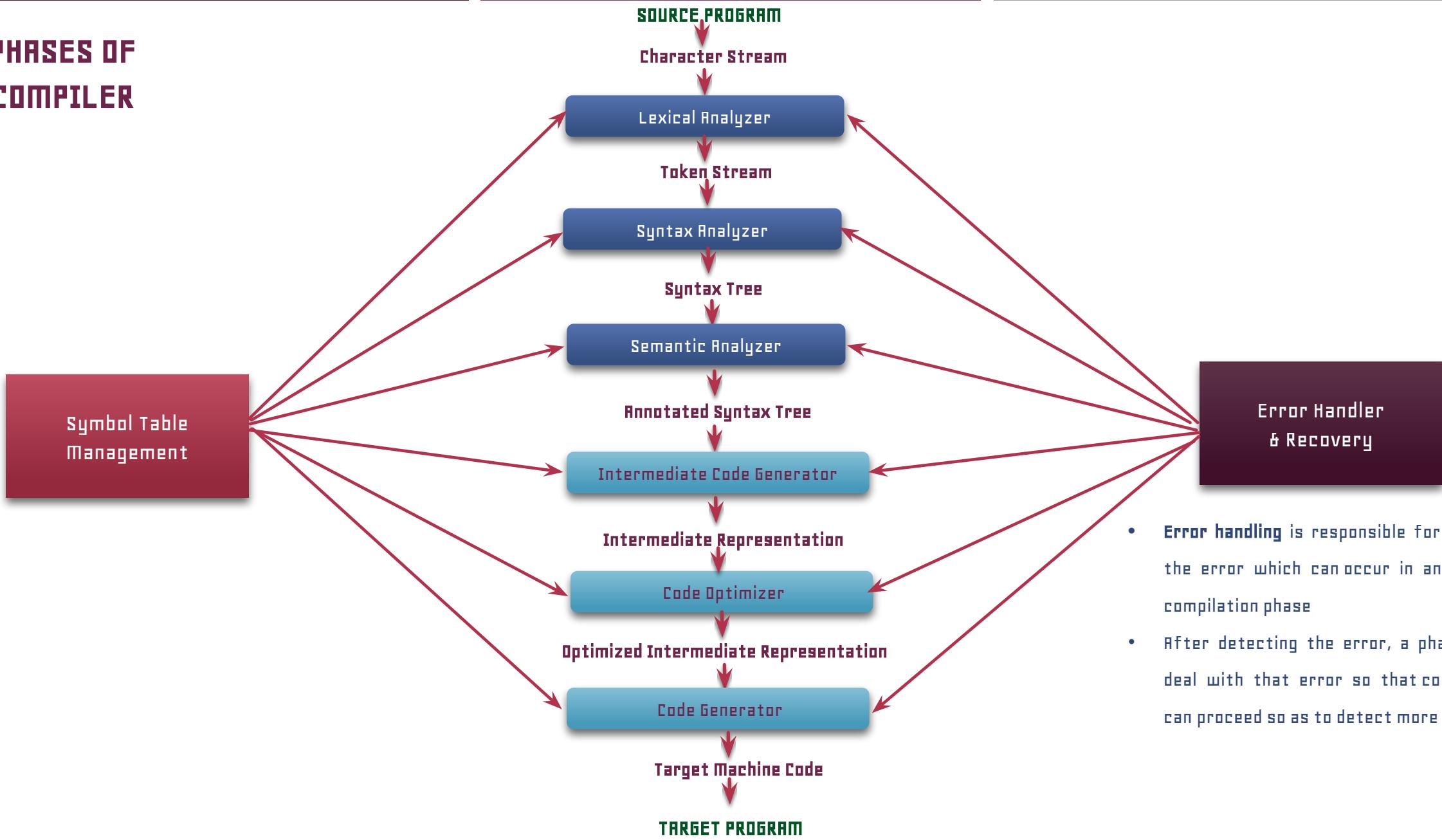
- This phase is responsible for the generation of target code [machine language code]
- Memory locations are selected for each variable
- Instructions are translated into a sequence of assembly instructions
- Variables and intermediate results are assigned to memory registers

PHASES OF COMPILER



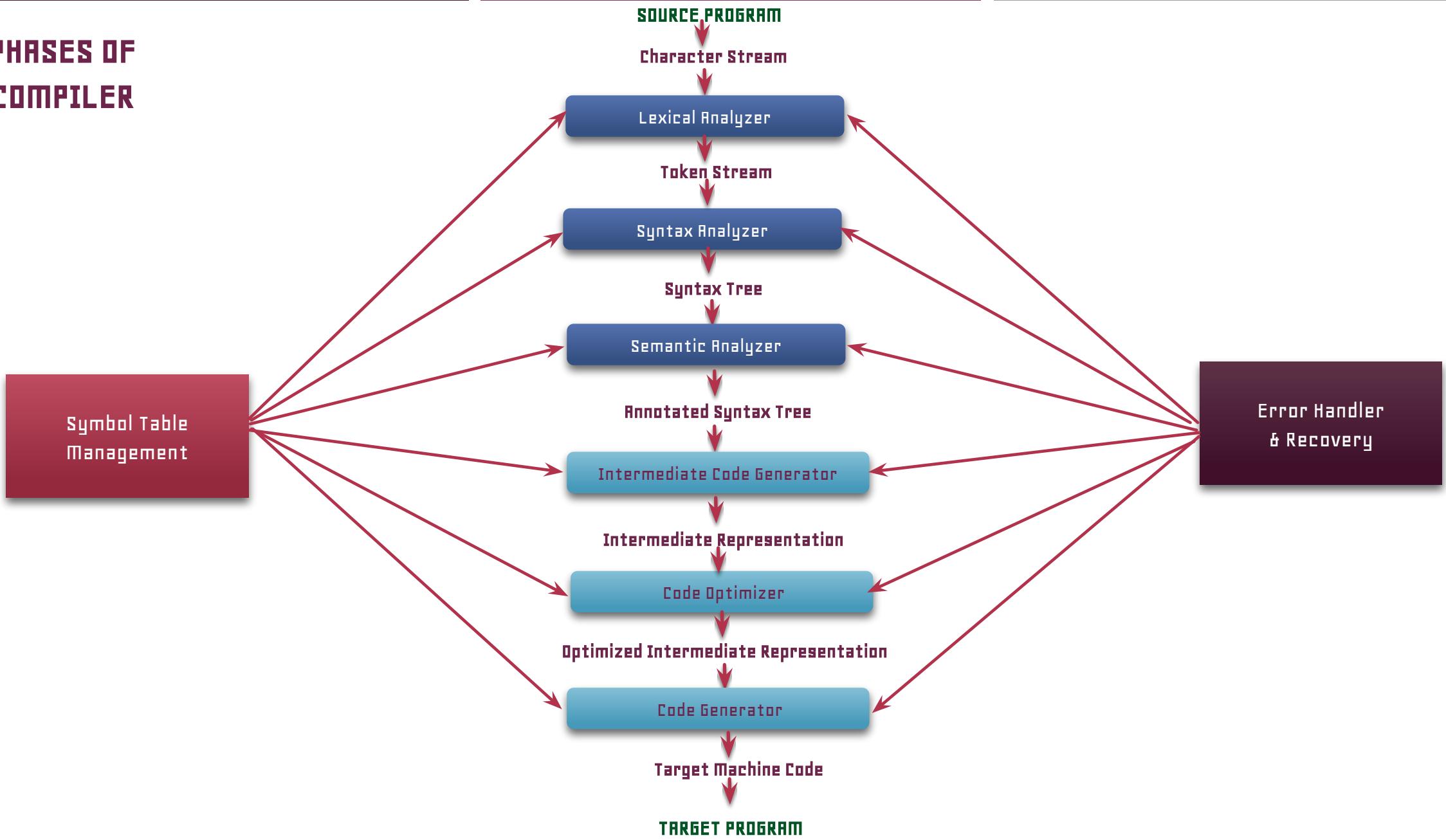
- **Symbol table** is a data structure holding information about all the **symbols** defined in the source program
- It is used as a reference table by all the phases of a compiler
- The typical information stored in the symbol table includes the name of the variables, their types, sizes relative offset within the program and so on
- The generation of this table is normally carried out by the lexical analyzer and syntax analyzer phases

PHASES OF COMPILER



- Error handling is responsible for handling the error which can occur in any of the compilation phase
- After detecting the error, a phase must deal with that error so that compilation can proceed so as to detect more errors

PHASES OF COMPILER

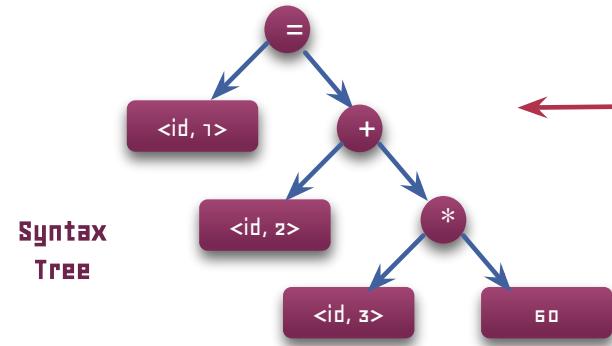


EXAMPLE

TRANSLATION OF ASSIGNMENT STATEMENT

Example

Position = initial + rate * 60



Position = initial + rate * 60

Lexical Analyzer

<id, 1> <id, 2> + <id, 3> * <id, 4>

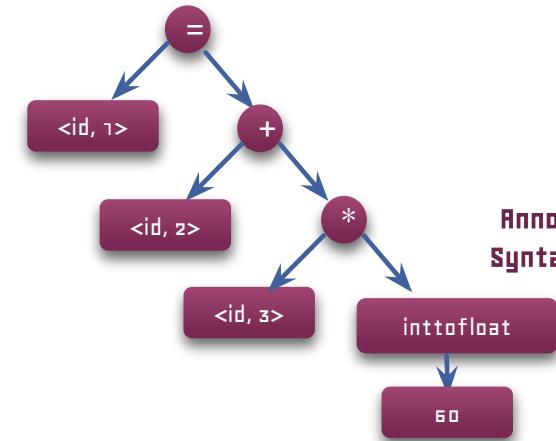
Syntax Analyzer

Token Stream

Semantic Analyzer

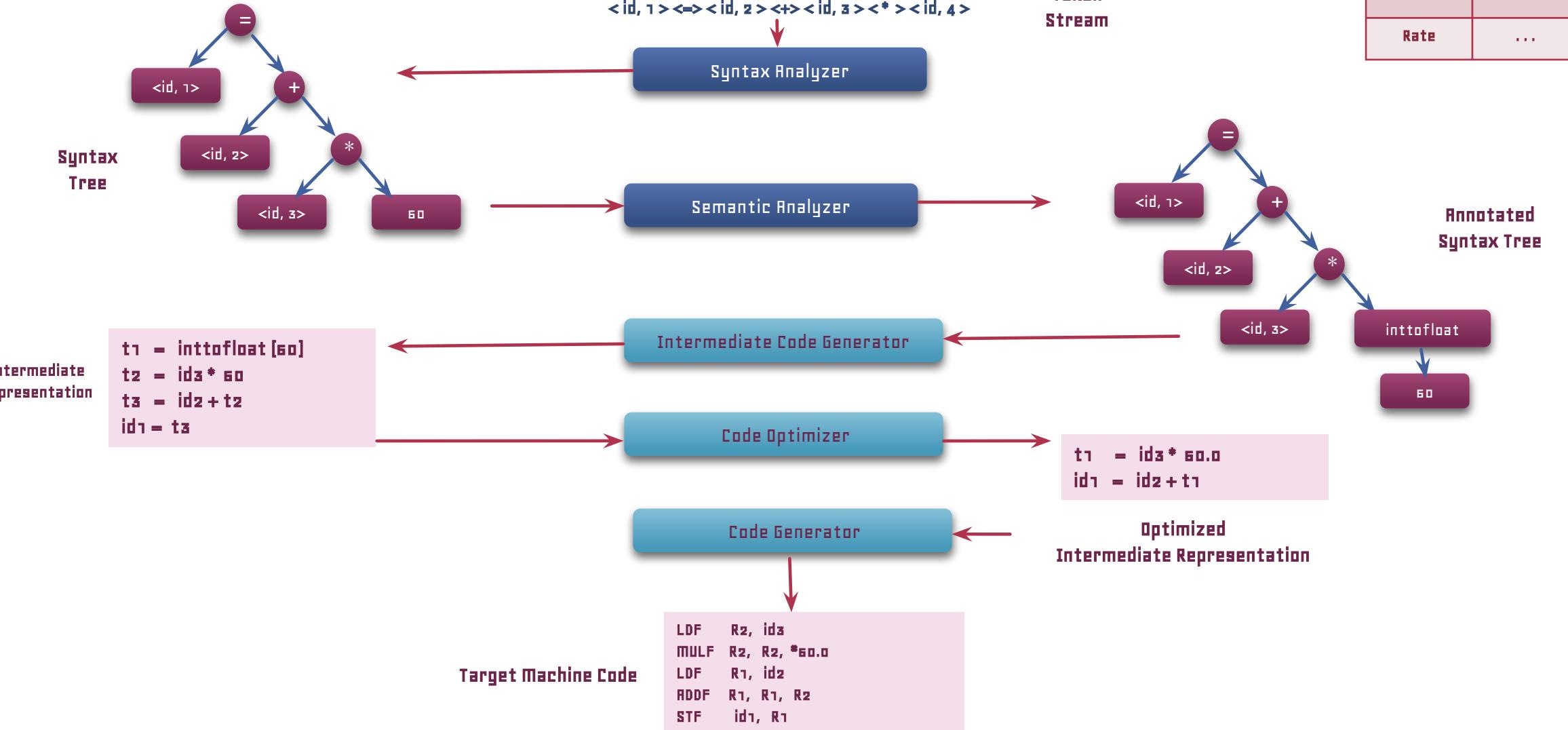
Symbol Table

Position	...
Initial	...
Rate	...



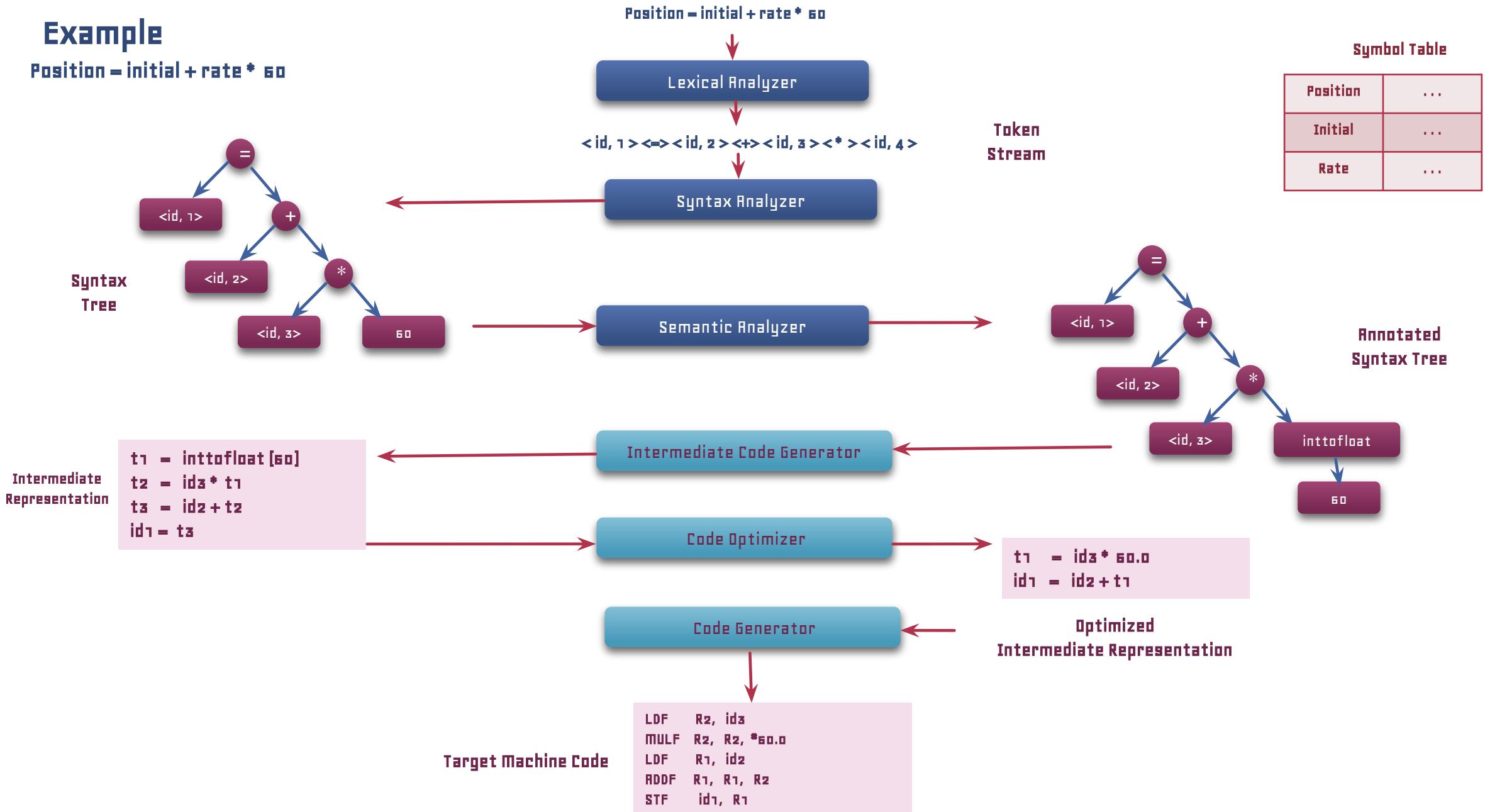
Example

$\text{Position} = \text{initial} + \text{rate} * 60$



Example

$\text{Position} = \text{initial} + \text{rate} * 60$



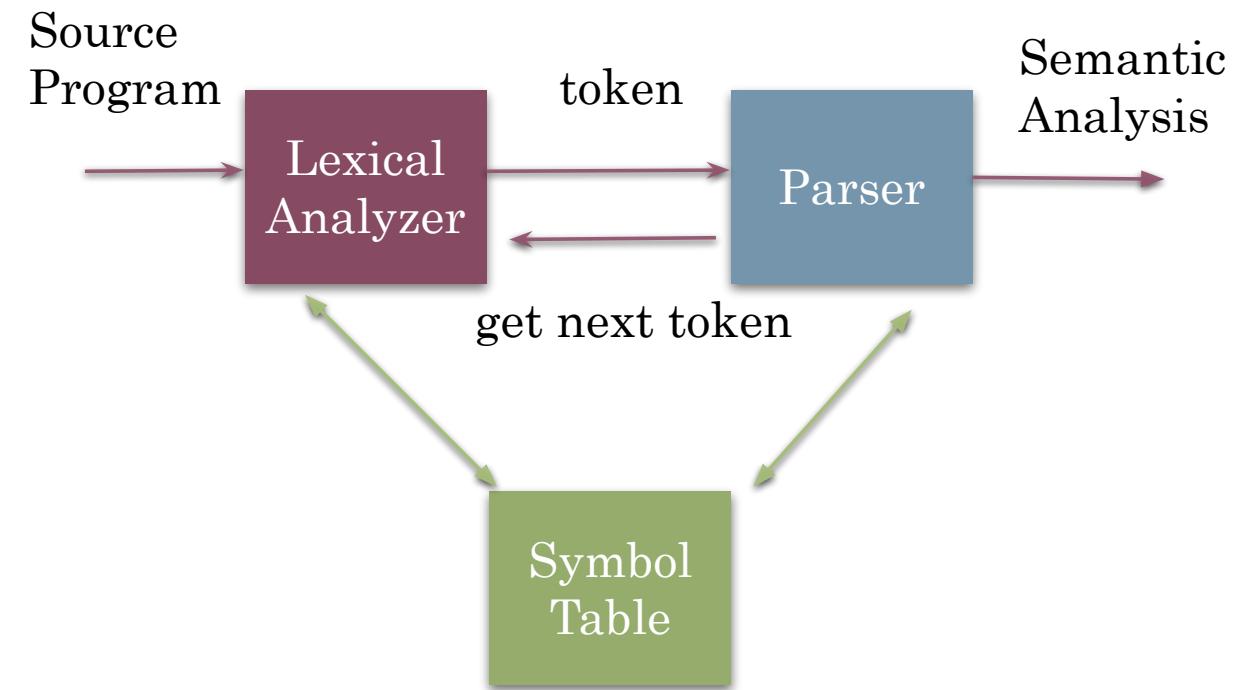
Lexical Analysis

Module 5

Content

- Role of Lexical Analyzer
- Input Buffering
- Specification and recognition of Tokens

Role of Lexical Analyzer



- ✓ Lexical Analyzer is the first phase
- ✓ Main Task: Accept input as character and produce output as sequence of tokens
- ✓ Removal of Comment and white spaces
- ✓ Correlating error messages from the compiler with the source program
- ✓ If source language supports some macro processor functions, then these preprocessor functions may also be implemented

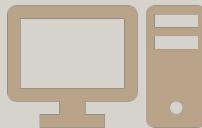
Issues in Lexical Analysis



Simpler design is the
most important
consideration



Compiler efficiency is
improved



Compiler portability
is enhanced

Tokens, Patterns and Lexemes

Token A string of characters which logically belong together

Tokens are treated as terminal symbols of the grammar specifying the source language

Pattern The set of strings described by the rule for which the same token is produced

The pattern is said to match each string in the set

Lexeme The sequence of characters matched by a pattern to form the corresponding token

Tokens, Patterns and Lexemes

Example:

const pi = 3.1416

Here 'pi' the Lexeme for the token identifier

Tokens, Patterns and Lexemes

Token	Informal Description	Sample Example
const	const	const
If	if	if
relation	< or <= or = or <> or >= or >	<, <=, =, <>, >, >=
id	Letter followed by letters and digits	Pi, count, D2
num	Any Numeric Constant	3.1416, 0, 6.02E23
literal	Any character between " and "	"core dumped"

Attributes of Token

- The lexical analyzer collects information about tokens into their associated attributes
- Attributes help in translation of tokens
- A token has a single attribute – A pointer to the symbol table entry in which the information about the token is kept
- The pointer becomes attribute for the token

Attributes of Token

$E = M * C ^\star 2$

Attributes for above statements

1. <id, pointer to symbol-table entry for E>
2. <assign_op,>
3. <id, pointer to symbol-table entry for M>
4. <mult_op,>
5. <id, pointer to symbol-table entry for C>
6. <exp_op,>
7. <num, integer value 2>

Lexical Errors

Recovery Strategies

1. Panic Mode Recovery- Delete successive characters from the remaining input until the lexical analyzer finds a well-formed token
2. Deleting an extraneous character
3. Inserting a missing character
4. Replacing an incorrect character by a correct character
5. Transposing two adjacent characters

Input Buffering

- The Lexical Analyzer scans the characters of the source program one at a time to discover tokens
- A lot of time is consumed scanning the characters
- So, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character
- 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,

Input Buffering

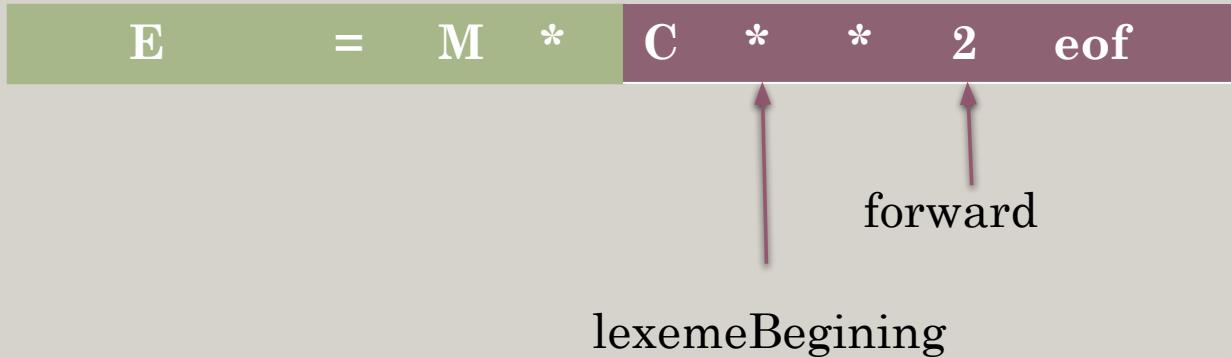
Buffering Techniques:

1. Buffer Pairs
2. Sentinels

Two pointers to the input are maintained:

- Pointer *lexemeBegin* marks the beginning of the current lexeme, whose extent we are attempting to determine.
- Pointer *forward* scans ahead until a pattern match is found.

Buffer Pairs



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

Buffer Pairs

Code to advance forward pointer

if forward at end of first half then begin

 reload second half;

 forward:=forward + 1

end

else if forward at end of second half then begin

 reload first half;

 move forward to beginning of first half

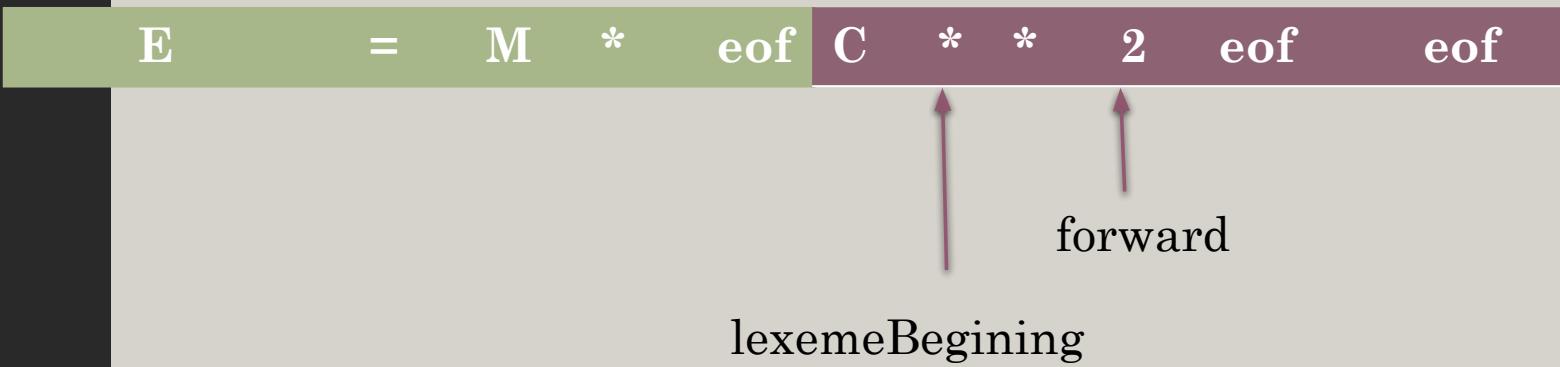
end

else

 forward:=forward + 1;

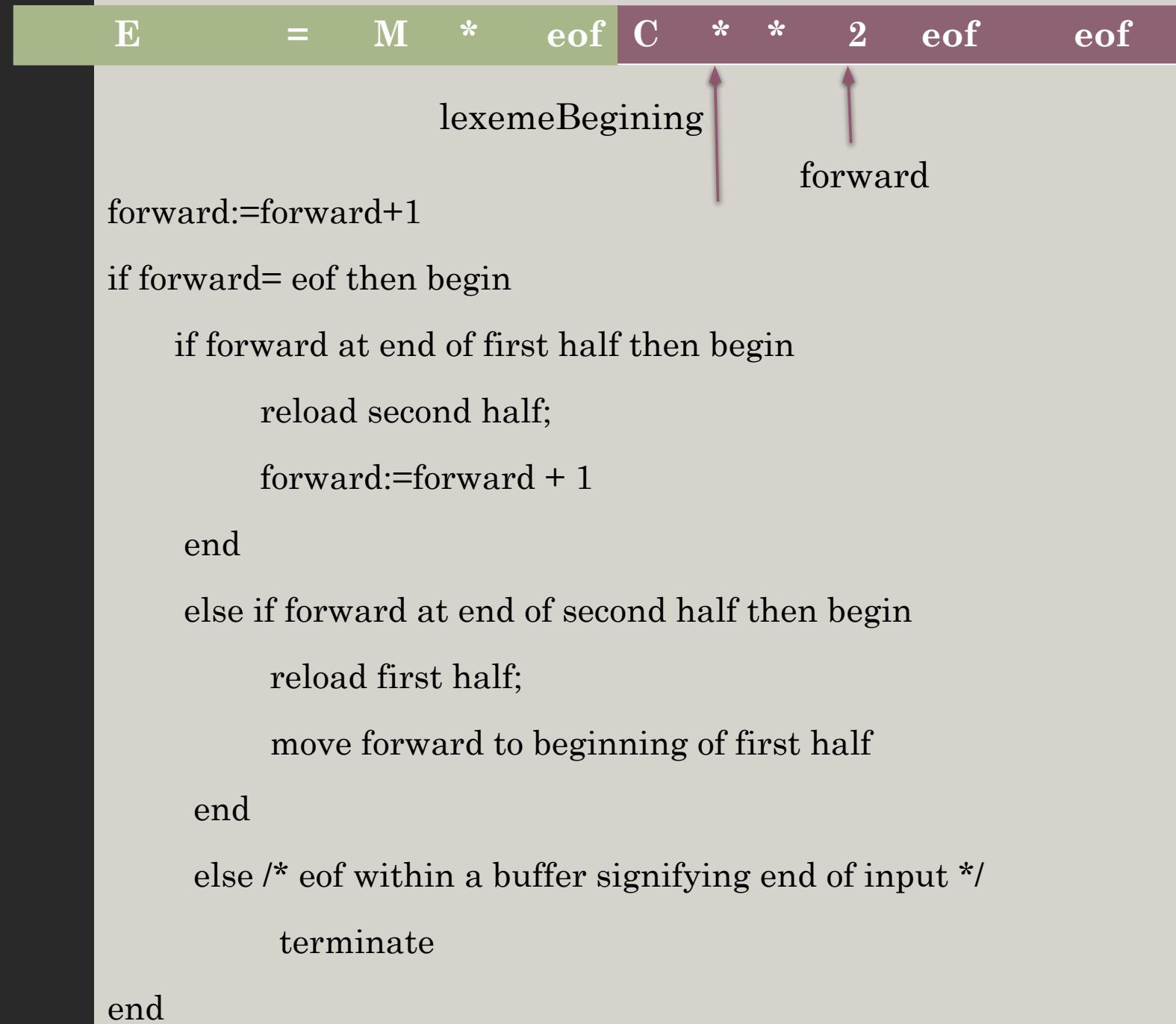


Sentinels



- for each character read, there are two tests:
 - one for the end of the buffer
 - one to determine what character is read
- We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end.
- The **sentinel** is a special character that cannot be part of the source program, and a natural choice is the character eof.

Sentinels



Lexical Analysis

Module 5

Content

- Specification of Tokens

Strings and Languages

Alphabet

- Any finite set of symbols.
- Letters, digits and punctuation
- $\{0,1\}$ – binary alphabet

String

- String over an alphabet is a finite sequence of symbols drawn from that alphabet.
- “Compiler” is a string of length eight.
- The empty string, denoted ϵ , is the string of length zero.

Strings and Languages

Language

- It is any countable set of strings over some fixed alphabet.
- Abstract languages are \emptyset , the empty set, or $\{\varepsilon\}$

Specification of Tokens

Operation	Definition
Union	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$

Specification of Tokens

Operation	Definition
Union	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation	$L \cdot M = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$

Specification of Tokens

Operation	Definition
Union	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation	$L \cdot M = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = L^0 \cup L^1 \cup L^2 \dots$

Specification of Tokens

Operation	Definition
Union	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation	$L \cdot M = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = L^0 \cup L^1 \cup L^2 \dots$
Positive Closure of L	$L^+ = L^1 \cup L^2 \cup L^3 \dots$

Specification of Tokens

Operations on Languages

Let L be the set {A, B,.....Z,a,b,.....z} and D be the set {0,1,....9}

L is the alphabet consisting of the set of upper and lower case letters

D is the alphabet consisting of the set of the ten decimal digits

Possible Operations:

- L U D is the set of letters and digits
- L . D is the set of string consisting of a letter followed by a digit
- L⁴ is the set of all four letter string
- L* is the set of all strings of letters including empty string
- L(L U D)* is the set of all strings of letters and digits beginning with letter
- D+ is the set of all strings of one or more digits.

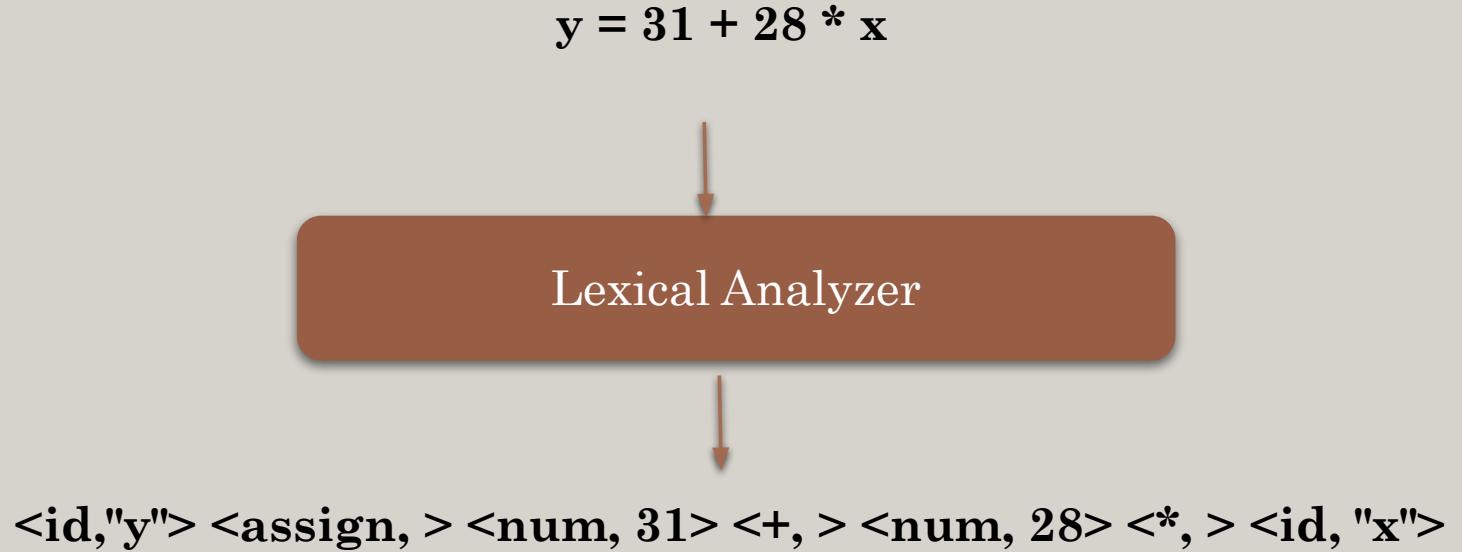
Regular Expression

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

- The regular expression $a | b$ denotes the set $\{a, b\}$
- The regular expression $(a | b)(a | b)$ denotes $\{aa, ab, ba, bb\}$.
The set of all strings of a's and b's of length two
- The regular expression a^* denotes set of all strings of zero or more a's. $r = \{ e, a, aa, aaa, aaaa, \dots \}$
- If two regular expressions represents same language then we can say that they are equivalent.

Tokens

- Token is usually represented by a pair token type and token value



Parser

Recognition of Tokens

A Grammar of branching statement

- stmt -> if expr **then** stmt |
 if expr **then** stmt **else** stmt |
 ε
- expr -> term **relop** term | term
- term -> **id** | **number**

Recognition of Tokens

Lexemes	Token Name	Attribute Value
Any ws	--	--
if	if	--
else	else	--
then	then	--
id	id	Pointer to table entry
num	num	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Pattern of Tokens

Token	Pattern
digit	[0–9]
digits	digit+
number	digits (.digits) ? (E [+ -] ? digits)?
letter	[A-Za-z]
id	letter (letter digit)*
if	if
then	then
else	else
relop	< <= > >= = <>

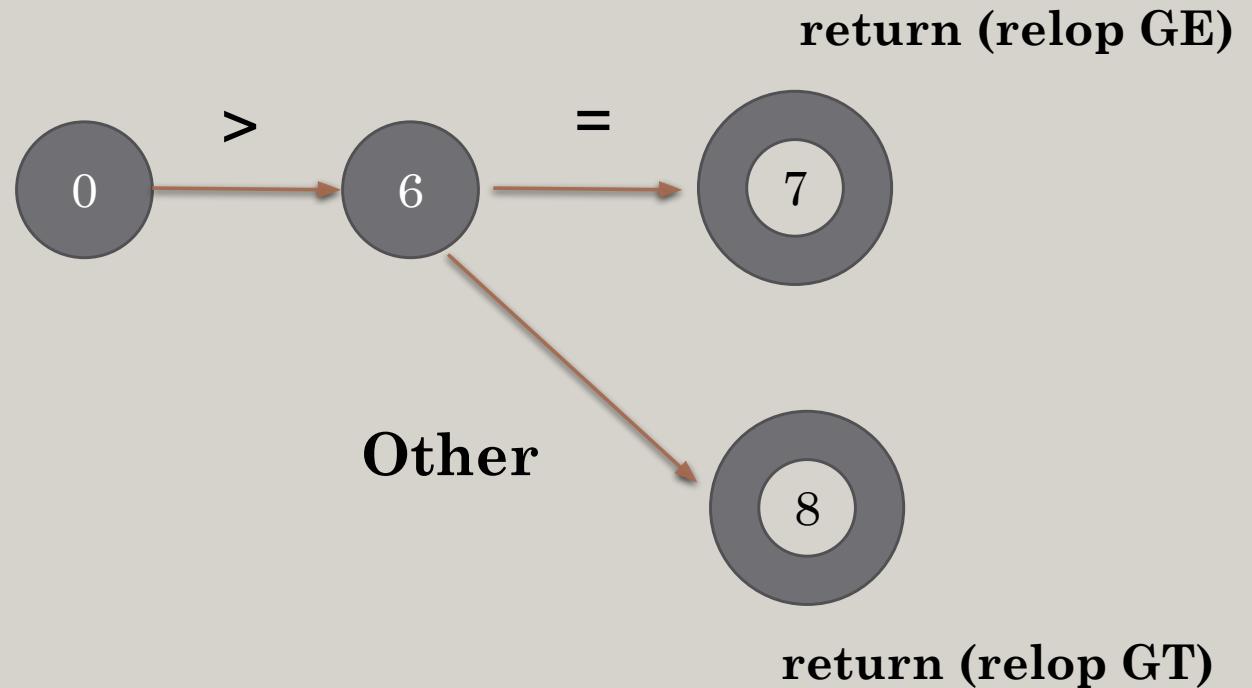
Recognition of Tokens

Transition Diagrams

- We convert patterns into stylized flowcharts, called "transition diagrams"
- Transition diagrams have a collection of nodes or circles, called states
- Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns
- Edges are directed from one state of the transition diagram to another
- Each edge is labeled by a symbol or set of symbols

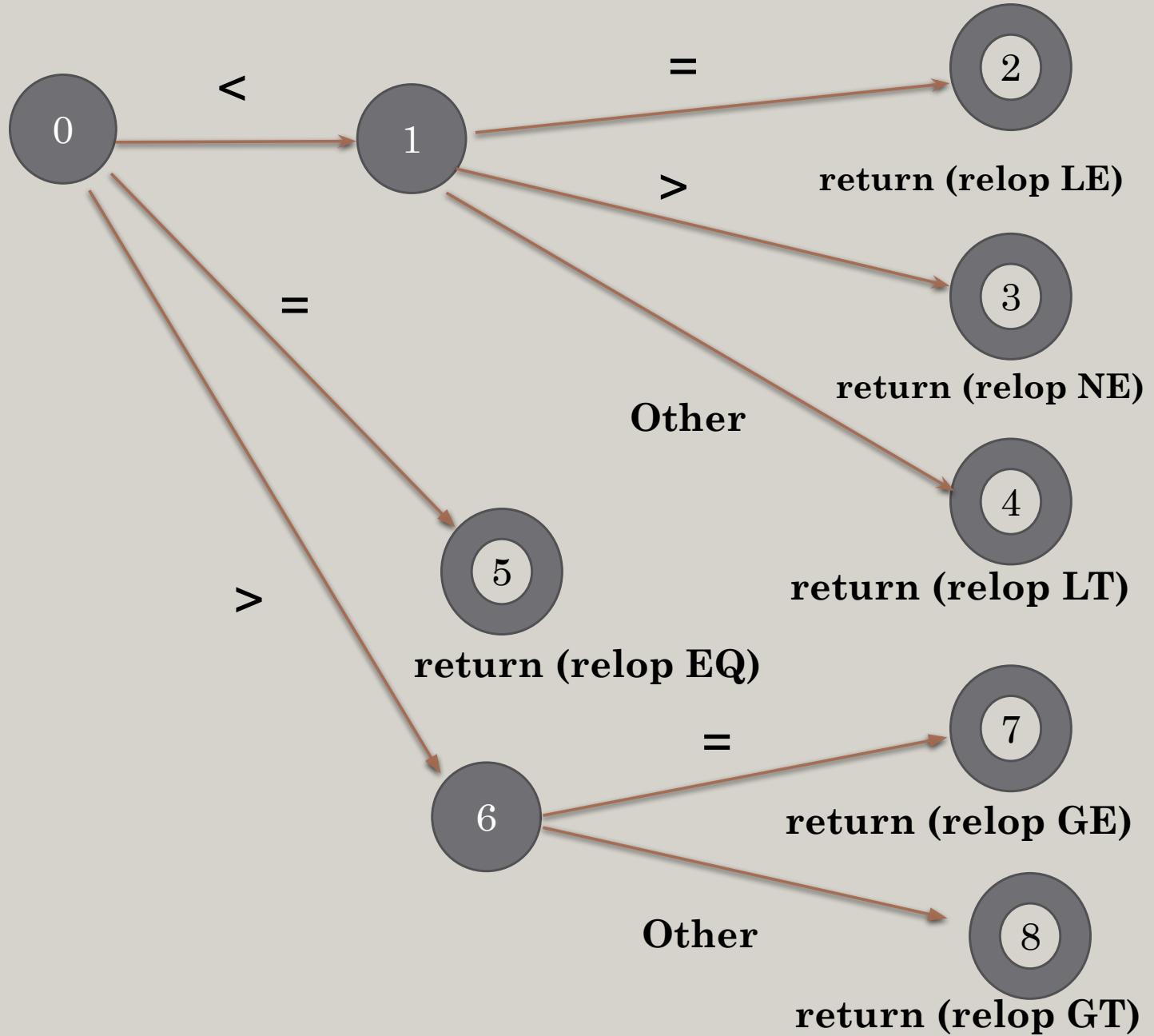
Recognition of Tokens

Transition Diagram for \geq



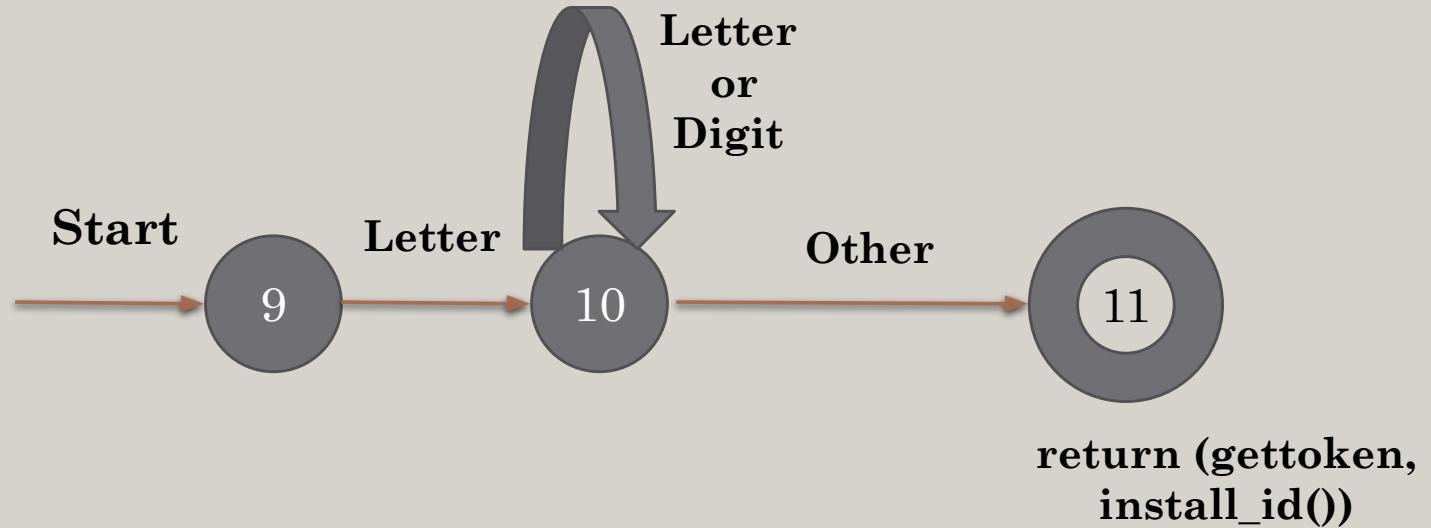
Recognition of Tokens

Transition Diagram for relop



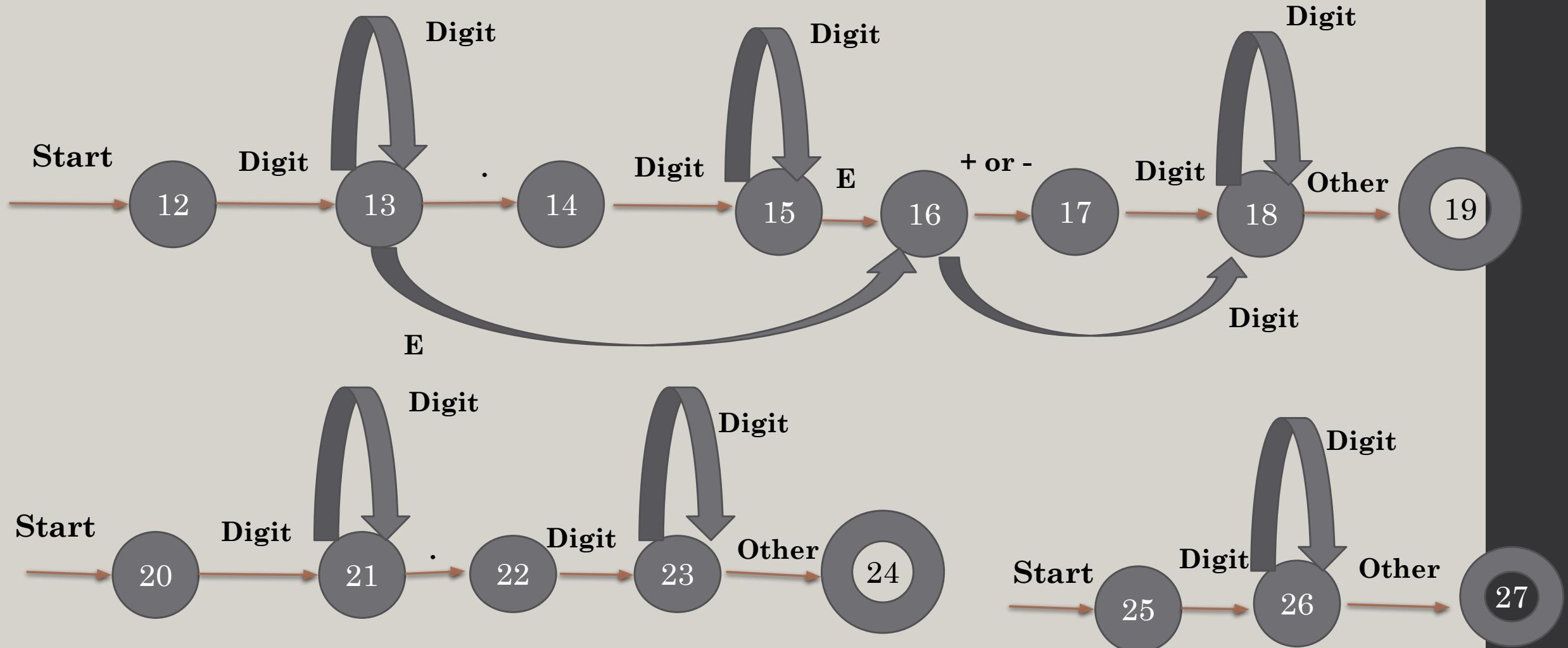
Recognition of Tokens

Transition Diagram for identifiers



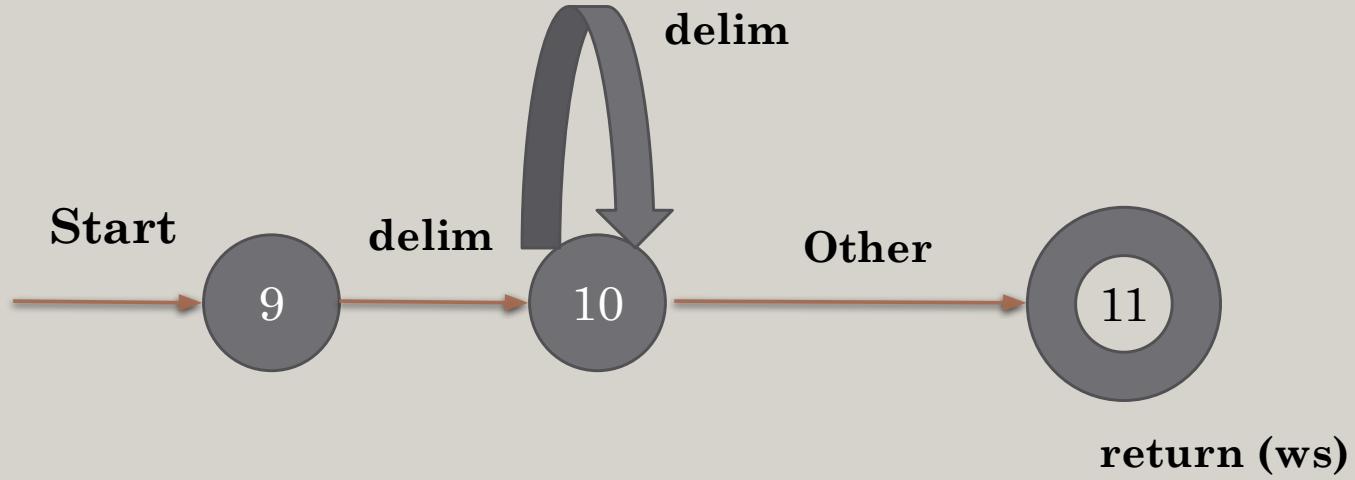
Recognition of Tokens

Transition Diagram of Unsigned Number



Recognition of Tokens

Transition Diagram for white spaces



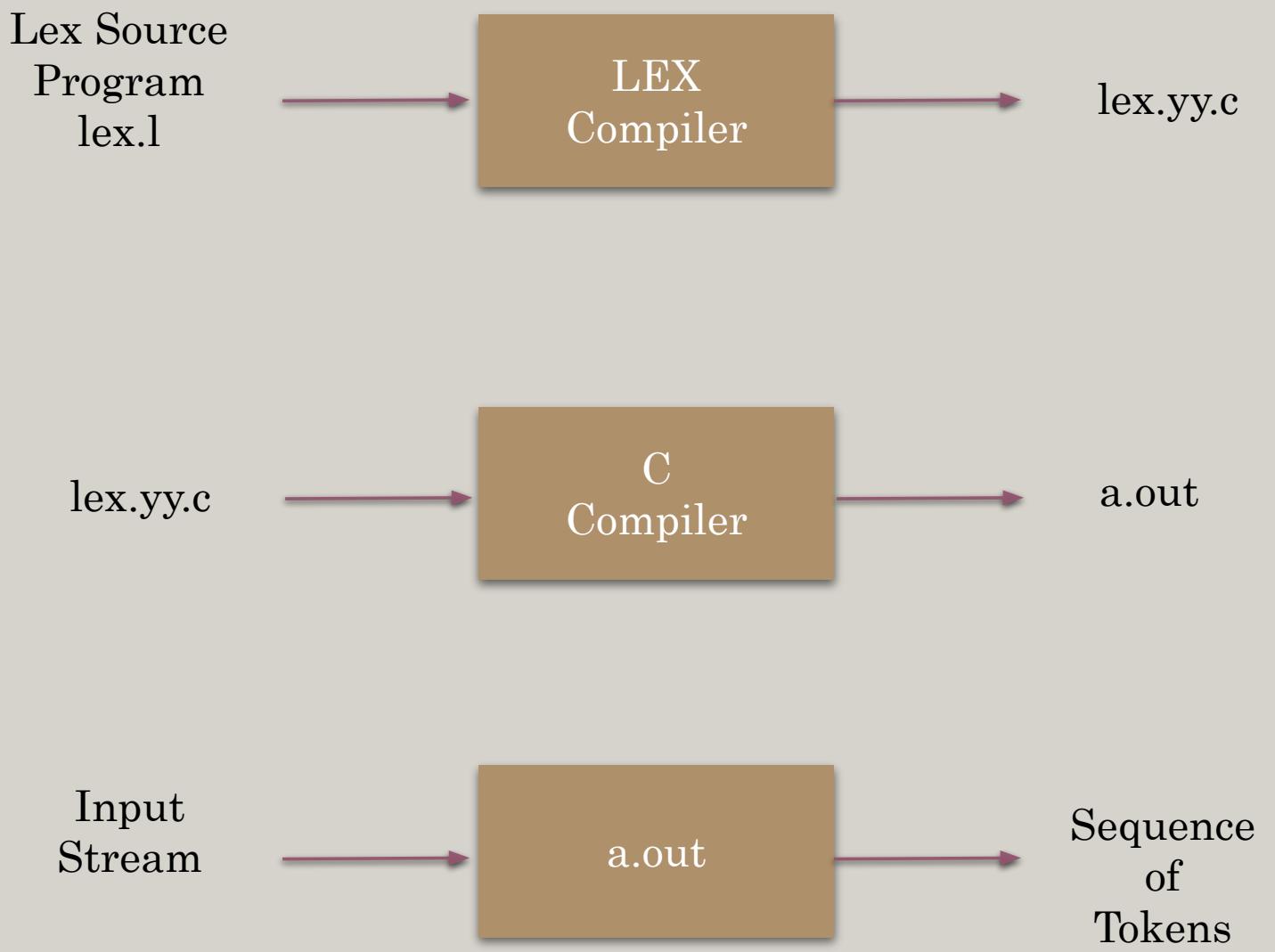
Lexical Analysis

Module 5

Content

- Lex Compiler
- Structure of LEX compiler
- Design of Lexical Analyzer Generator
- Data Structures used in Lexical Analysis

LEX Compiler



Structure of LEX program

Lex program consists of three parts:-

%{

Declaration

%}

%%

Rule Section

%%

Auxiliary Procedure

Structure of LEX program

Translation Rules are of the form:-

P1 {action1}

P2 {action2}

....

Pi {action3}

- where each Pi is a regular expression and
- each action i is a program fragment describing what action the lexical analyzer should take when pattern pi matches a lexeme

Structure of LEX program

Auxiliary Procedures

- This section holds whatever auxiliary functions are used in the actions
- These functions can be compiled separately and loaded with the lexical analyzer

Working of Lexical Analyzer with Parser

- When called by the parser, the lexical analyzer begins reading its remaining input, one character at a time
- It reads until it finds the longest prefix of the input that matches one of the patterns
- It then executes action i which returns the control to the parser
- But if it does not then the lexical analyzer proceeds to find additional lexemes, until one of the corresponding actions causes a return to the parser.
- The lexical analyzer returns a single quantity i.e the token to the parser
- To pass an attribute value with information about the lexeme, a global variable is set known as **yylval**

LEX program for Tokens

```
%{  
/* Definitions of manifest constants  
  
LT, LE, EQ, NE, GT, GE, IF, THEN, ELSE, ID, NUMBER, RELOP */  
  
%}  
  
/* Regular definitions */  
  
delim [\t\n]  
ws {delim}+  
letter [A-Za-z]  
digit [0-9]  
id {letter}({letter} | {digit})*  
number {digit}+(\. {digit} +)? (E [+ -] ? {digit}+)?
```

LEX program for Tokens

```
%%  
  
{ws}      { /* No action and No Return */ }  
  
if        { return (IF); }  
  
else      { return (ELSE); }  
  
then      { return (THEN); }  
  
{id}       { yyval = (int) installID(); return (ID); }  
  
{number}   { yyval = (int) installNum(); return (NUMBER); }  
  
“<”       { yyval = LT;  return(RELOP); }  
  
“<=”     { yyval = LE;  return(RELOP); }  
  
“=”       { yyval = EQ;  return(RELOP); }  
  
“<>”     { yyval = NE;  return(RELOP); }  
  
“>”       { yyval = GT;  return(RELOP); }  
  
“>=”     { yyval = GE;  return(RELOP); }  
  
%%
```

LEX program for Tokens

```
int installID() {  
  
/* Function to install the lexeme whose first character is pointed to by yytext, and whose length  
is yyleng, into the symbol table and return a pointer to Symbol Table  
  
*/}  
  
int installNum() {  
  
/* Similar to installID but puts numerical constants into a separate table */}
```

LEX Program

Steps for installation of LEX and YACC

- sudo apt-get update
- sudo apt-get install flex
- sudo apt-get install bison

Compilation steps:

- lex program.l
- gcc lex.yy.c -lfl
- ./a.out

LEX Program

Example 1

```
%{  
%}  
%%  
\n { printf("\n Hello Good Morning");  
%%  
void main()  
{  
    yylex();  
}
```

LEX Program

Example 2

```
%{  
char name[10];  
%}  
  
%%  
[n] {printf("\n Hi.....%s.....Good Morning\n",name); return 1;}  
%%  
void main()  
{  
    char opt;  
    do {  
        printf("\nWhat is your name?"); scanf("%s",name);  
        yylex();  
        printf("\nPress y to continue"); scanf("%c",&opt);  
    }  
    while(opt=='y');  
}
```

LEX Program

Example 3

```
%{  
int linecount = 0, charcount = 0;  
%}  
  
%%  
. { charcount++; }  
\n { linecount++; charcount++; }  
%%  
  
void main()  
{  
    yylex();  
    printf("# of lines = %d, # of chars = %d", linecount, charcount);  
}
```

LEX Program Example 4

```
%{  
    void display(int, char *);  
    int flag;  
}%  
  
%%  
[a|e|i|o|u] { flag =1; display(flag,yytext); }  
.         { flag =0; display(flag,yytext); }  
%%  
  
void main()  
{  
    printf("\nEnter the word:"); yylex();  
}
```

```
void display(int flag,char *t)  
{  
    if(flag==1)  
    {  
        printf("\nThe given character %s is vowel \n",t);  
    }  
    else  
    {  
        printf("\nThe given character %s is not vowel \n",t);  
    }  
}
```

Input Buffer

Lexeme

Lexeme begin

Forward

Automaton
Simulator

Transition
Table
Action

Lex
Program

Lex
Compiler

Design of Lexical Analyzer Generator

- A Lex program is turned into a transition table and actions which are used by a finite automaton simulator
- The program that serves as the lexical analyzer includes a fixed program that simulates an automaton
- The rest of the lexical analyzer consists of following component
 1. A transition table of the automaton
 2. Those functions that are passed directly through Lex to the output
 3. The action from the input program which is to be invoked at the appropriate time by Automaton Simulator

Input Buffer

Lexeme

Lexeme begin

Forward

Automaton
Simulator

Transition
Table
Action

Lex
Program

Lex
Compiler

Design of Lexical Analyzer Generator

- To construct the automaton take each regular expression pattern in the Lex program and convert it using Algorithm to an NFA
- We need a single automaton that will recognize lexemes matching any of the patterns in the program
- Hence combine all the NFA's into one by introducing a new start state with transitions to each of the start states of the NFA's N_i for pattern p_i

Data Structure used in Lexical Analyzer

- Symbol: Identifier used in the Source Program
- Examples: Names of variables, functions and Procedures
- Symbol Table: To maintain information about attributes of symbol
- Operations on Symbol Table:
 1. Add a symbol and its attributes
 2. Locate a symbol's entry
 3. Delete a symbol's entry
 4. Access a symbol's entry

Data Structure used in Lexical Analyzer

- Design Goal of Symbol Table
 - 1. The Table's organization should facilitate efficient search
 - 2. ii. Table should be compact (Less Memory)
- Time Space Trade off
- To improve search efficiency, allocate more memory to symbol table
- Organization of Entries:
 - 1. Linear Data Structure
 - 2. Non-Linear Data Structure

Data Structure used in Lexical Analyzer

Symbol Table Entry Format

- Number of Fields to accommodate attributes of one symbol
- Symbol Field: The symbol to be stored
- Key Field: Basis for the search in table
- Entry of Symbol is called record
- Each entry can be of type fixed length, variable length or hybrid

Data Structure used in Lexical Analyzer

Symbol Class	Attributes
Variable	Type, Length, number and bounds of dimensions
Procedure	Number of parameters, address of parameter list
Function	Number of parameters, address of parameter list, type and length of returned value
Label	Statement Number





SYNTAX ANALYSIS

LECTURE I

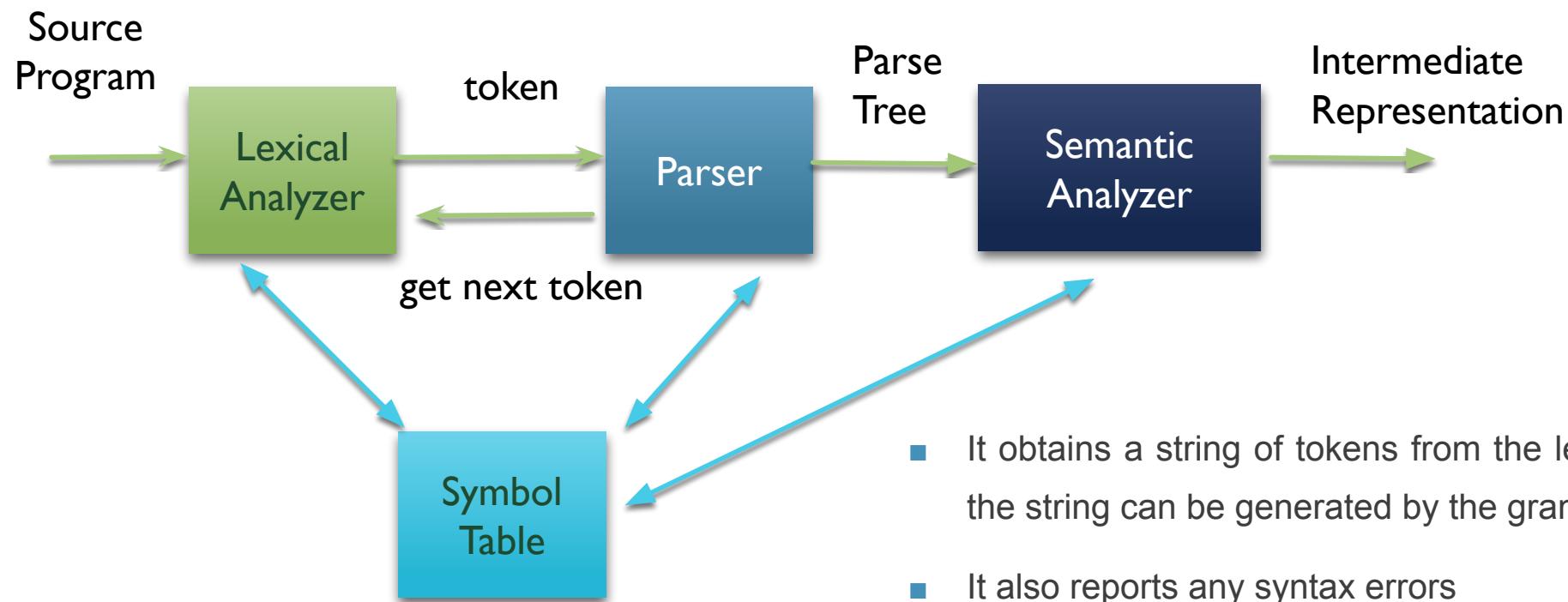
CONTENT

- Introduction
- Role of a parser
- Error Handling Techniques
- Context Free Grammar

INTRODUCTION

- Syntax analysis is the second phase of the compiler
- It is also called as parsing and it generates parse tree
- Parser: It is the program that takes tokens and grammar (context-free grammar -CFG) as input and validates the input token against the grammar

ROLE OF THE PARSER



- It obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language
- It also reports any syntax errors
- It also recovers from commonly occurring errors so that it can continue processing the remaining input

TYPE OF ERRORS

- **Lexical:** such as misspelling a keyword.
- **Syntactic:** such as an arithmetic expression with unbalanced parentheses.
- **Semantic,** such as an operator applied to an incompatible operand.
- **Logical:** such as an infinitely recursive call.

ERROR HANDLER IN PARSER

Goals of Error handler:

- Report the presence of errors clearly and accurately
- Recover from each error quickly enough to detect subsequent errors
- Add minimal overhead to the processing of correct programs

ERROR RECOVERY TECHNIQUES

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

Panic mode recovery

- Panic mode error recovery is based on the idea of discarding input symbols one at a time until one of the designated set of synchronized tokens is found
- The synchronizing tokens are usually delimiters, such as semicolon or end
- Advantage of simplicity and does not go into an infinite loop.
- Useful when multiple errors in the same statement are rare

Phrase Level Recovery

- On discovering error, a parser may perform a local fix to allow the parser to continue and simultaneously report error
- In phrase level recovery mode each empty entry in the parsing table is filled with a pointer to a specific error routine to take care of that error
- These error routine may be:
 1. Change , insert, or delete input symbol
 2. Issue an appropriate error message
 3. Pop items from the stack

Error Production

- Error production adds rules to the grammar that describes the erroneous syntax .
- This strategy can resolve many , but not all potential errors
- It includes production for common errors and we can augment the grammar for the production rules that generates the erroneous constructs
- A parser constructed from an augmented grammar by these error productions detects the anticipated error production is used during parsing
- Since it is almost impossible to know all the errors that can be made by the programmers , this method is not practical

Global Correction

- Replace incorrect input with input that is correct and require the fewer changes to create
- This requires expensive techniques that are costly in terms of time and space

The algorithm states that:

- For a given grammar G give an incorrect input string X
- Now find a parse tree for a related string y with the help of an algorithm such that the number of insertions , deletion and changes of token require to change x into y is as small as possible

Context Free Grammar

A context free grammar has four tuple $\langle V, T, P, S \rangle$ where,

- V: set of non terminal symbols for writing the grammar
- T: Set of terminal symbol T, used as token for the language
- P: Set of production rule
- S: A special non terminal which is start

Context Free Grammar

$$E \rightarrow E + E \mid E * E \mid (E) \mid - E \mid \text{id}$$

To generate a valid string: $- (\text{id} + \text{id} * \text{id})$

Steps:

Rightmost Derivation

Here, rightmost Non-Terminal is replaced in every step.

$$E \rightarrow (E)$$
$$E \rightarrow (E+E)$$
$$E \rightarrow (E+E^*E)$$
$$E \rightarrow (E+E^*\text{id})$$
$$E \rightarrow (E+\text{id}^*\text{id})$$
$$E \rightarrow (\text{id}+\text{id}^*\text{id})$$

Context Free Grammar

$$E \rightarrow E + E \mid E * E \mid (E) \mid - E \mid \text{id}$$

To generate a valid string: - (id + id * id)

Steps:

Leftmost Derivation

Here, leftmost Non-Terminal is replaced in every step.

$$E \rightarrow (E)$$
$$E \rightarrow (E^*E)$$
$$E \rightarrow (E+E^*E)$$
$$E \rightarrow (\text{id}+E^*E)$$
$$E \rightarrow (\text{id}+\text{id}^*E)$$
$$E \rightarrow (\text{id}+\text{id}^*\text{id})$$



SYNTAX ANALYSIS

LECTURE 2

CONTENT

- Context Free Grammar
- Derivation and Parse Tree
- Ambiguous and Unambiguous Grammar
- Removing Unambiguity
 - Removing Left Recursion
 - Left Factoring

Context Free Grammar

$$E \rightarrow E + E \mid E * E \mid (E) \mid - E \mid \text{id}$$

To generate a valid string: - (id + id * id)

Steps:

Rightmost Derivation

Here, rightmost Non-Terminal is replaced in every step.

$$E \rightarrow (E)$$
$$E \rightarrow (E+E)$$
$$E \rightarrow (E+E^*E)$$
$$E \rightarrow (E+E^*\text{id})$$
$$E \rightarrow (E+\text{id}^*\text{id})$$
$$E \rightarrow (\text{id}+\text{id}^*\text{id})$$

Context Free Grammar

$$E \rightarrow E + E \mid E * E \mid (E) \mid - E \mid \text{id}$$

To generate a valid string: - (id + id * id)

Steps:

Leftmost Derivation

Here, leftmost Non-Terminal is replaced in every step.

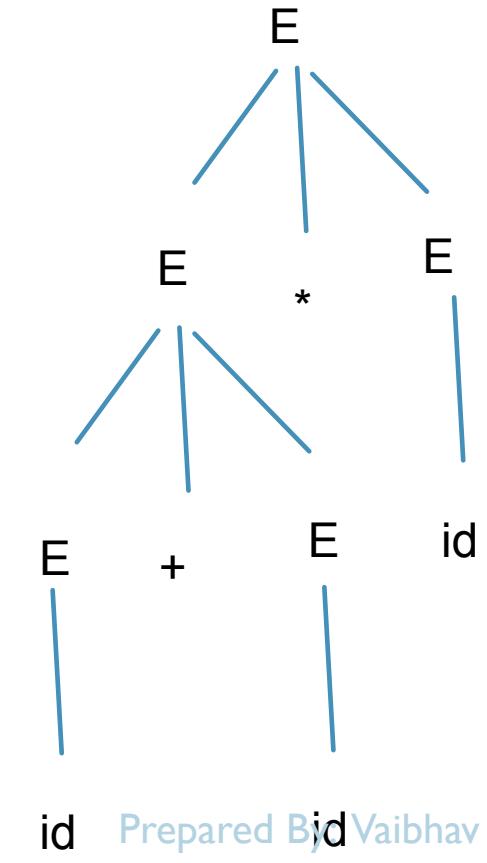
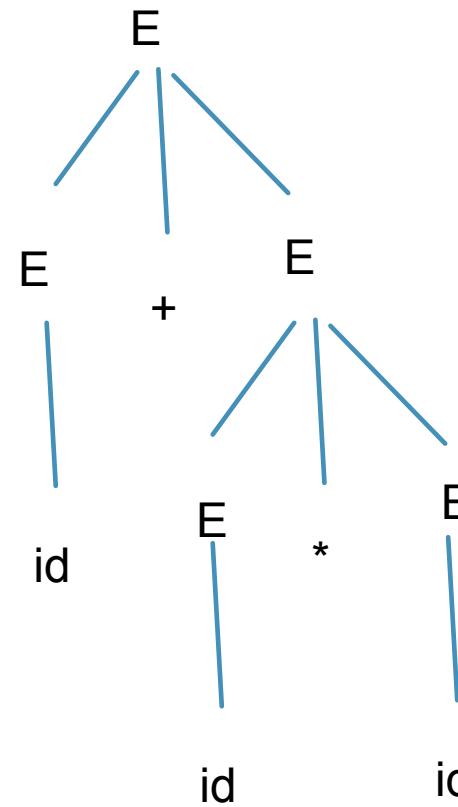
$$E \rightarrow (E)$$
$$E \rightarrow (E^*E)$$
$$E \rightarrow (E+E^*E)$$
$$E \rightarrow (\text{id}+E^*E)$$
$$E \rightarrow (\text{id}+\text{id}^*E)$$
$$E \rightarrow (\text{id}+\text{id}^*\text{id})$$

Context Free Grammar

$$E \rightarrow E + E \mid E * E \mid (E) \mid - E \mid \text{id}$$

To generate a valid string: $- (\text{id} + \text{id} * \text{id})$

Parse Trees



Ambiguous Grammar

- Ambiguous Grammar

For a given grammar, we can generate at least one string which can be presented using more than one parse tree then such grammar is called ambiguous grammar

- Unambiguous Grammar

For a given grammar, all possible strings which can be generated using it have only one representation of Parse Tree, such grammar is called Unambiguous Grammar

Removing Unambiguity

- Two Techniques:
 1. Removing Left Recursion
 2. Left Factoring

Removing Left Recursion

- A grammar is left recursive if it has a nonterminal such that there is a derivation $A \rightarrow A\alpha$ for some string α .
- Top-down parsing methods cannot handle left recursive grammars.
- Left recursion can be eliminated as follows:

Given Left Recursion: $A \rightarrow A\alpha | \beta$

Then,

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' | \epsilon$$

Examples on Removing Left Recursion

Given Grammar:

$$E \rightarrow E + T | T$$

$$T \rightarrow T^* F | F$$

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

Solution:

First eliminate the left recursion for E as $E \rightarrow E + T | T$

$$E \rightarrow TE'$$

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

Then eliminate for T as $T \rightarrow T^* F | F$

$$T \rightarrow FT'$$

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

Examples on Removing Left Recursion

Given Grammar:

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

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

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

Solution:

$$E \rightarrow T E'$$

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

$$T \rightarrow F T'$$

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

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

Examples on Removing Left Recursion

Example 2:

$$A \rightarrow ABd \mid Aa \mid a$$

$$B \rightarrow Be \mid b$$

Solution-

The grammar after eliminating left recursion is-

$$A \rightarrow aA'$$

$$A' \rightarrow BdA' \mid aA' \mid \epsilon$$

$$B \rightarrow bB'$$

$$B' \rightarrow eB' \mid \epsilon$$

Examples on Removing Left Recursion

Example 3:

$S \rightarrow A$

$A \rightarrow Ad \mid Ae \mid aB \mid ac$

$B \rightarrow bBc \mid f$

Solution:

The grammar after eliminating left recursion is-

$S \rightarrow A$

$A \rightarrow aBA' \mid acA'$

$A' \rightarrow dA' \mid eA' \mid \epsilon$

$B \rightarrow bBc \mid f$

Left Factoring

- It is a grammar transformation that is suitable for producing grammar for predictive parser
- Basic Idea: When it is not clear which of two alternative productions to use to expand non Terminal

If $A \rightarrow \alpha \beta_1 | \alpha \beta_2$

The left factor of above grammar is

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 | \beta_2$

Example on Left Factoring

$S \rightarrow a \mid ab \mid abc \mid abcd$

Solution.-

Step-01:

$S \rightarrow aS'$

$S' \rightarrow b \mid bc \mid bcd \mid \epsilon$

Again, this is a grammar with common prefixes

Step-02:

$S \rightarrow aS'$

$S' \rightarrow bA' \mid \epsilon$

$A' \rightarrow c \mid cd \mid \epsilon$

Again, this is a grammar with common

Example on Left Factoring

$S \rightarrow a \mid ab \mid abc \mid abcd$

Solution.-

Step-03:

$S \rightarrow aS'$

$S' \rightarrow bA' \mid \epsilon$

$A' \rightarrow cB' \mid \epsilon$

$B' \rightarrow d \mid \epsilon$

Example on Left Factoring

Example 2:

$$S \rightarrow aAd \mid aB$$

$$A \rightarrow a \mid ab$$

$$B \rightarrow ccd \mid ddc$$

Solution-

The left factored grammar is-

$$S \rightarrow aS'$$

$$S' \rightarrow Ad \mid B$$

$$A \rightarrow aA'$$

$$A' \rightarrow b \mid \epsilon$$

$$B \rightarrow ccd \mid ddc$$

Example on Left Factoring

Example 3:

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

Solution:

Step-01:

$$S \rightarrow aS' \mid b$$

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

Again, this is a grammar with common prefixes

Step-02:

$$S \rightarrow aS' \mid b$$

$$S' \rightarrow SA' \mid bb \quad A' \rightarrow SbS \mid aSb$$



SYNTAX ANALYSIS

LECTURE 3 - 4

CONTENT

- Top-Down Parsing
- Recursive Descent Parsing
- Concepts of FIRST and FOLLOW
- Examples

TOP DOWN PARSING

- Can be viewed as a problem of constructing a parse tree for the input string
- It starts from the root and create nodes of the parse tree in pre order (Depth First)
- Equivalently can be viewed as a leftmost derivation

TOP DOWN PARSING

Example:

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E)|id$

Leftmost Derivation of string id + id * id

$E \rightarrow TE'$

$\rightarrow FT'E'$

$\rightarrow idT'E'$

$\rightarrow id\ e\ E'$

$\rightarrow id\ +\ TE'$

$\rightarrow id\ +\ FT'E'$

$\rightarrow id\ +\ id\ T'E'$

$\rightarrow id\ +\ id\ *\ F\ T'\ E'$

$\rightarrow id\ +\ id\ *\ id\ T'\ E'$

$\rightarrow id\ +\ id\ *\ id\ e\ E'$

$\rightarrow id\ +\ id\ *\ id\ e\ e$

TOP DOWN PARSING

- At each of the step the key problem is that Determining the production to be applied for a nonterminal say A
- Once an A production is chosen the rest of the parsing process consists of matching the terminal symbols in the production body with the input string

RECURSIVE DESCENT PARSING

```
void A( )  
{  
    Choose an A production A → X1 X2 ... Xk  
    For ( i to k)  
    {  
        if (Xi is a nonterminal)  
            call procedure Xi( );  
        else if (Xi equals the current input symbol a)  
            advance the input to the next symbol  
        else  
            an error has occurred  
    }  
}
```

RECURSIVE DESCENT PARSING

- A recursive descent parsing program consists of a set of procedures one for each nonterminal
- Execution begins with the procedure for the start symbol
- The execution halts and announces success if its procedure body scans entire input string.
- Note that this pseudo code is nondeterministic since it begins by choosing the A production to apply in a manner that is not specified
- Backtracking is required

RECURSIVE DESCENT PARSING

Consider the grammar

$$S \rightarrow c A d$$

$$A \rightarrow a b \mid a$$

Derivation of string $w = cad$

If you go with $S \rightarrow cAd$ and then $A \rightarrow ab$ it leads to wrong string.

$$S \rightarrow cabd$$

Backtracking is necessary

FIRST OF GRAMMAR

- If X is a terminal then FIRST (X) = X
- If X is a nonterminal and
 $X \rightarrow Y_1 Y_2 \dots Y_k$ is a production for some $k \geq 1$
 - a. if for some i, 'a' is in FIRST(Y_i) and
 - b. epsilon is in all of FIRST Y_j where $j = 1, 2, \dots, i-1$

Then add 'a' in the FIRST(X)
- Case 2:
If epsilon is in FIRST(Y_j) for all $j = 1, 2, \dots, k$
then add epsilon in FIRST (X)
- If $X \rightarrow e$ is a production then add epsilon in FIRST(X)

FIRST OF GRAMMAR

Example 1:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

Solution:

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

FIRST OF GRAMMAR

Example 2

- A \rightarrow BC
- B \rightarrow Ax | x
- C \rightarrow yC | y

Solution

- In A \rightarrow BC
 $\text{FIRST}(A) = \{\text{FIRST}(B) \cup \text{FIRST}(C)\}$ if $B \rightarrow \epsilon$ is true
- $\text{FIRST}(A) = \{\text{FIRST}(B)\}$ if $B \rightarrow \epsilon$ is false
- $\text{FIRST}(A) = \{x\}$
- $\text{FIRST}(B) = \{x\}$
- $\text{FIRST}(C) = \{y\}$

FOLLOW OF GRAMMAR

Rule 1:

Place \$ in FOLLOW (S) where S is the start symbol and \$ is the input right endmarker

Rule 2:

If there is a production $A \rightarrow \alpha B \beta$ then everything in FIRST(β) except ϵ is in FOLLOW (B)

Rule 3:

If there is a production $A \rightarrow \alpha B$ or
a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains ϵ
then everything in FOLLOW (A) is in FOLLOW (B)

APPLY ABOVE RULES UNTIL THERE IS NO UPDATION IN FOLLOW LIST

FOLLOW OF GRAMMAR

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

Example 1:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

Solution:

1. FOLLOW (E) = { \$,) }

Since E is start symbol and Production Rule F → (E)

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

By Rule 3 of FOLLOW

FOLLOW OF GRAMMAR

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

Example 1:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

Solution:

$$3. FOLLOW(T) = \{ FIRST(E') - \varepsilon \} \cup \{ FOLLOW(E') \}$$

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

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

$$4. FOLLOW(T') = FOLLOW(T)$$

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

FOLLOW OF GRAMMAR

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

Example 1:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

Solution:

$$\begin{aligned}5. \text{ FOLLOW}(F) &= \{ \text{FIRST}(T') - \varepsilon \} \cup \{ \text{FOLLOW}(T') \} \\&= \{ * \} \cup \{ + , \$,) \} \\&= \{ * , + , \$,) \}\end{aligned}$$

FOLLOW OF GRAMMAR

	FIRST	FOLLOW
A	x	\$, x
B	x	y
C	y	\$, x

Example 2:

$$A \rightarrow BC$$

$$B \rightarrow Ax \mid x$$

$$C \rightarrow yC \mid y$$

Solution:

1. FOLLOW (A) = { \$ } U FIRST (x) ... As A is the start symbol

$$= \{ \$, x \}$$

2. FOLLOW (B) = FIRST (C)

$$= \{ y \}$$

3. FOLLOW (C) = FOLLOW (A)

$$= \{ \$, x \}$$

FOLLOW OF GRAMMAR

	FIRST	FOLLOW
S	d, g, h, b, a, ϵ	
A	d , g , h , ϵ	
B	g, ϵ	
C	h , ϵ	

Example 3:

$$S \rightarrow ACB \mid Cbb \mid Ba$$

$$A \rightarrow da \mid BC$$

$$B \rightarrow g \mid \epsilon$$

$$C \rightarrow h \mid \epsilon$$

Solution:

$$\begin{aligned}1. \text{ FIRST}(S) &= \text{FIRST}(A) \cup \text{FIRST}(C) \cup \text{FIRST}(B) \cup \text{FIRST}(b) \\&\quad \cup \text{FIRST}(a)\end{aligned}$$

$$\begin{aligned}&= \{ d, g, h \} \cup \{ h , \epsilon \} \cup \{ g , \epsilon \} \cup \{ b , a \} \\&= \{ d , g , h , b , a , \epsilon \}\end{aligned}$$

$$\begin{aligned}2. \text{ FIRST}(A) &= \text{FIRST}(d) \cup \text{FIRST}(B) \cup \text{FIRST}(C) \\&= \{ d , g , h , \epsilon \}\end{aligned}$$

$$3. \text{ FIRST}(B) = \{ g , \epsilon \}$$

$$4. \text{ FIRST}(C) = \{ h , \epsilon \}$$

FOLLOW OF GRAMMAR

	FIRST	FOLLOW
S	d, g, h, b, a, ϵ	\$
A	d , g , h , ϵ	h , g , \$
B	g, ϵ	a , h , g , \$
C	h , ϵ	b , h , g , \$

Example 3:

$$S \rightarrow ACB \mid Cbb \mid Ba$$

$$A \rightarrow da \mid BC$$

$$B \rightarrow g \mid \epsilon$$

$$C \rightarrow h \mid \epsilon$$

Solution:

1. FOLLOW (S) = { \$ } ... Since S is start symbol
2. FOLLOW (A) = { FIRST (C) - ϵ } U { FIRST (B) - ϵ } U FOLLOW (S)
= { h , g , \$ }
3. FOLLOW (B) = FIRST (a) U { FIRST (C) - ϵ } U FOLLOW (A) U FOLLOW (S)
= { a , h , g , \$ }
3. FOLLOW (C) = FIRST (b) U { FIRST(B) - ϵ } U FOLLOW (A)
= { b , g , h , \$ }

FOLLOW OF GRAMMAR

	FIRST	FOLLOW
S	a , b , d , ε	
A	a , b , d , ε	
B	b , d , ε	
D	d , ε	

Example 4:

$$S \rightarrow ABD$$

$$A \rightarrow a \mid BSB$$

$$B \rightarrow b \mid D$$

$$D \rightarrow d \mid \epsilon$$

Solution:

$$1. \text{FIRST}(S) = \text{FIRST}(A)$$

$$= \{ a \} \cup \text{FIRST}(B)$$

$$= \{ a \} \cup \{ b \} \cup \text{FIRST}(D)$$

$$= \{ a , b , d , \epsilon \}$$

$$2. \text{FIRST}(A) = \{ a \} \cup \text{FIRST}(B)$$

$$= \{ a \} \cup \{ b \} \cup \text{FIRST}(D)$$

$$= \{ a , b , d , \epsilon \}$$

$$3. \text{FIRST}(B) = \{ b \} \cup \text{FIRST}(D)$$

$$= \{ b , d , \epsilon \}$$

$$4. \text{FIRST}(D) = \{ d , \epsilon \}$$

FOLLOW OF GRAMMAR

	FIRST	FOLLOW
S	a , b , d , ϵ	b , d , \$
A	a , b , d , ϵ	b , d , \$
B	b , d , ϵ	a , b , d , \$
D	d , ϵ	a , b , d , \$

Example 4:

$$S \rightarrow ABD$$

$$A \rightarrow a \mid BSB$$

$$B \rightarrow b \mid D$$

$$D \rightarrow d \mid \epsilon$$

Solution:

1. FOLLOW (S) = { \$ } U { FIRST (B) - ϵ } U FOLLOW (A)
= { \$, b , d } U { FIRST (B) - ϵ } U { FIRST (D) - ϵ }
= { b , d , \$ }
2. FOLLOW (A) = { FIRST (B) - ϵ } U { FIRST (D) - ϵ } U FOLLOW (S)
= { b , d , \$ }
3. FOLLOW (B) = { FIRST (D) - ϵ } U FOLLOW (S) U { FIRST (S) - ϵ } U FOLLOW (A)
= { d , b , a , \$ }
3. FOLLOW (D) = FOLLOW (B) U FOLLOW (S)
= { a , b , d , \$ }

SYNTAX ANALYSIS

Lecture 5
PREDICTIVE PARSER

CONTENT

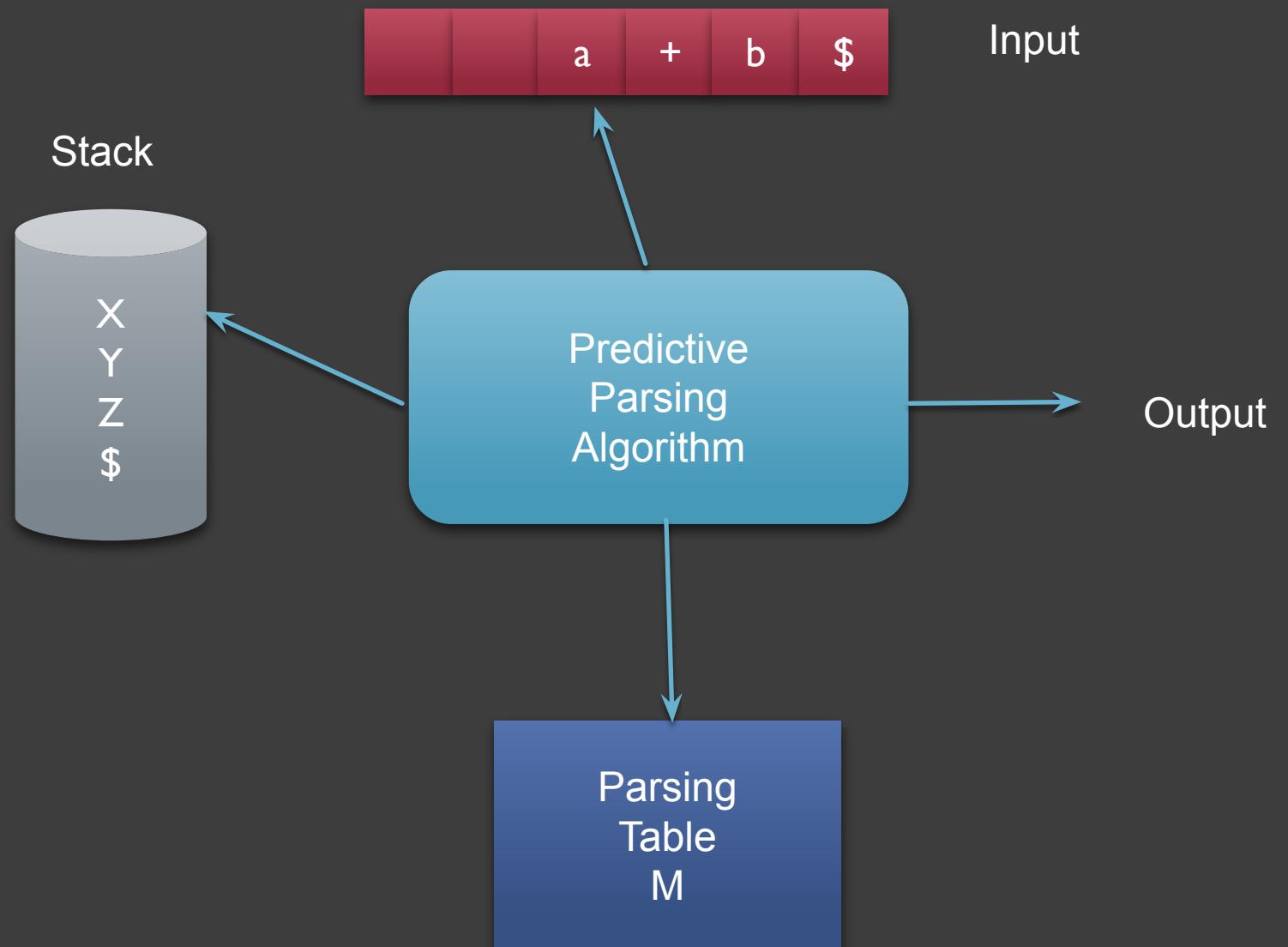
- LL (1) Parser
- Model of Non-Recursive Predictive Parser
- Construction of Predictive Parser Table
- Parsing a string

LL (I) GRAMMAR

LL (1) GRAMMAR

- Used to construct Predictive Parser
- Predictive Parser – Recursive Descent Parser with no need of Backtracking
- First 'L' - scanning input from Left to Right
- Second 'L' - Leftmost Derivation
- "1" - One input symbol of Look ahead at each step to make parsing action decisions
- Left Recursive and Ambiguous grammar is NOT LL(1)

Model Of Non-Recursive Predictive Parser



Construction Of Predictive Parsing Table

Input: Grammar G
Output: Parsing Table M

Table M has Non-Terminals as row and Terminals as columns

For Each Production $A \rightarrow \alpha$ of grammar do step 1 and 2

Step 1: For each terminal 'a' in FIRST (α)

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

Step 2:

Case 1:

If ϵ is in FIRST (α) then

for each terminal b in FOLLOW (A)

Add $A \rightarrow \alpha$ to M [A , b]

Case 2:

If ϵ is in FIRST (α) and \$ is in FOLLOW (A) then

for each terminal b in FOLLOW (A)

Add $A \rightarrow \alpha$ to M [A , b]

Step 3: Make each undefined entry of M be an error

Example:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

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

Consider Production $E \rightarrow TE'$

$$\text{FIRST} (TE') = \text{FIRST} (T) = \{ (, id \}$$

Add $E \rightarrow TE'$

to M [E , (] and M [E , id]

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

Example:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

Consider Production $E \rightarrow TE'$

$$\text{FIRST}(TE') = \text{FIRST}(T) = \{(, id \}$$

Add $E \rightarrow TE'$

to M [E , (] and M [E , id]

Example:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

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

Consider Production $E' \rightarrow +TE'$

$$\text{FIRST} (+TE') = \text{FIRST} (+) = \{ + \}$$

Add $E' \rightarrow +TE'$ to M [E' , +]

Example:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

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

Consider Production $E' \rightarrow +TE'$

$$\text{FIRST} (+TE') = \text{FIRST} (+) = \{ + \}$$

Add $E \rightarrow +TE'$ to M [E , +]

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

Example:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

Consider Production $E' \rightarrow \epsilon$

$$\text{FIRST}(\epsilon) = \{ \epsilon \}$$

$$\text{Here find FOLLOW}(E') = \{ \) , \$ \}$$

Add $E' \rightarrow \epsilon$ to M [E' ,)] and M [E' , \$]

Example:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

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

Consider Production $E' \rightarrow \epsilon$

$$\text{FIRST}(\epsilon) = \{ \epsilon \}$$

$$\text{Here find FOLLOW}(E') = \{ \), \$ \}$$

Add $E' \rightarrow \epsilon$ to M [E ,)] and M [E , \$]

Example:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

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

Consider Production $T \rightarrow FT'$

$$\text{FIRST}(FT') = \text{FIRST}(F) = \{(, id \}$$

Add $T \rightarrow FT'$ to M [T , (] and M [T , id]

Example:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

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

Consider Production $T \rightarrow FT'$

$$\text{FIRST}(FT') = \text{FIRST}(F) = \{(, id \}$$

Add $T \rightarrow FT'$ to M [T , (] and M [T , id]

Example:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

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

Consider Production $T' \rightarrow *FT'$

$$\text{FIRST}(*FT') = \text{FIRST}(*) = \{ *\}$$

Add $T' \rightarrow *FT'$ to M [T' , *]

Example:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

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

Consider Production $T' \rightarrow *FT'$

$$\text{FIRST}(*FT') = \text{FIRST}(*) = \{ *\}$$

Add $T' \rightarrow *FT'$ to M [T' , *]

Example:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

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

Consider Production $T' \rightarrow \epsilon$

$$\text{FIRST}(\epsilon) = \{ \epsilon \}$$

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

Add $E' \rightarrow \epsilon$ to M [T', +], M [T',)] and M [T', \$]

Example:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

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

Consider Production $T' \rightarrow \epsilon$

$$\text{FIRST}(\epsilon) = \{ \epsilon \}$$

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

Add $E' \rightarrow \epsilon$ to M [T', +], M [T',)] and M [T', \$]

Example:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

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

Consider Production $F \rightarrow (E)$

$$\text{FIRST}((E)) = \text{FIRST}(()) = \{ (\}$$

Add $F \rightarrow (E)$ to M [F, (]

Example:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

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

Consider Production $F \rightarrow (E)$

$$\text{FIRST}((E)) = \text{FIRST}(()) = \{ (\}$$

Add $F \rightarrow (E)$ to M [F, (]

Example:

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

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

Consider Production $F \rightarrow id$

$\text{FIRST}(\text{id}) = \{ \text{id} \}$

Add $F \rightarrow id$ to M [F, id]

Example:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

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

Consider Production $F \rightarrow id$

FIRST (id) = { id }

Add $F \rightarrow id$ to M [F, id]

Example 2:

$$S \rightarrow iEtSS' | a$$

$$S' \rightarrow eS | \epsilon$$

$$E \rightarrow b$$

	a	b	e	i	t	\$
S						
S'						
E						

Consider Production $S \rightarrow iEtSS'$

$$\text{FIRST} (iEtSS') = \text{FIRST} (i) = \{ i \}$$

Add $S \rightarrow iEtSS'$ to M [S , i]

Example 2:

$$S \rightarrow iEtSS' | a$$

$$S' \rightarrow eS | \epsilon$$

$$E \rightarrow b$$

	a	b	e	i	t	\$
S				$S \rightarrow iEtSS'$		
S'						
E						

Consider Production $S \rightarrow iEtSS'$

$$\text{FIRST} (iEtSS') = \text{FIRST} (i) = \{ i \}$$

Add $S \rightarrow iEtSS'$ to M [S , i]

Example 2:

$$S \rightarrow iEtSS' | a$$

$$S' \rightarrow eS | \epsilon$$

$$E \rightarrow b$$

	a	b	e	i	t	\$
S				$S \rightarrow iEtSS'$		
S'						
E						

Consider Production $S \rightarrow a$

$$\text{FIRST} (a) = \{ a \}$$

Add $S \rightarrow a$ to M [S , a]

Example 2:

$$S \rightarrow iEtSS' | a$$

$$S' \rightarrow eS | \epsilon$$

$$E \rightarrow b$$

	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'						
E						

Consider Production $S \rightarrow a$

$$\text{FIRST} (a) = \{ a \}$$

Add $S \rightarrow a$ to M [S , a]

Example 2:

$$S \rightarrow iEtSS' | a$$

$$S' \rightarrow eS | \epsilon$$

$$E \rightarrow b$$

	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'						
E						

Consider Production $S' \rightarrow eS$

FIRST (e) = { e }

Add $S' \rightarrow eS$ to M [S' , e]

Example 2:

$$S \rightarrow iEtSS' \mid a$$

$$S' \rightarrow eS \mid \epsilon$$

$$E \rightarrow b$$

	a	b	e	i	t	\$
S	$S \rightarrow a$				$S \rightarrow iEtSS'$	
S'				$S' \rightarrow eS$		
E						

Consider Production $S' \rightarrow eS$

FIRST (e) = { e }

Add $S' \rightarrow eS$ to M [S' , e]

Example 2:

$$S \rightarrow iEtSS' \mid a$$

$$S' \rightarrow eS \mid \epsilon$$

$$E \rightarrow b$$

	a	b	e	i	t	\$
S	$S \rightarrow a$				$S \rightarrow iEtSS'$	
S'				$S' \rightarrow eS$		
E						

Consider Production $S' \rightarrow \epsilon$

$$\text{FIRST}(\epsilon) = \{\epsilon\}$$

$$\text{Here find FOLLOW}(S') = \{e, \$\}$$

Add $S' \rightarrow \epsilon$ to M [S', e] and M [S', \$]

Example 2:

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \epsilon$

$E \rightarrow b$

	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow eS$ $S' \rightarrow \epsilon$			$S' \rightarrow \epsilon$
E						

Consider Production $S' \rightarrow \epsilon$

FIRST (ϵ) = { ϵ }

Here find FOLLOW (S') = { e , \$ }

Add $S' \rightarrow \epsilon$ to M [S', e] and M [S' , \$]

Example 2:

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \epsilon$

$E \rightarrow b$

	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow eS$ $S' \rightarrow \epsilon$			$S' \rightarrow \epsilon$
E						

Consider Production $E \rightarrow b$

FIRST (b) = { b }

Here find FOLLOW (S') = { e , \$ }

Add $E \rightarrow b$ to M [E , b]

Example 2:

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \epsilon$

$E \rightarrow b$

	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow eS$ $S' \rightarrow \epsilon$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

Consider Production $E \rightarrow b$

FIRST (b) = { b }

Here find FOLLOW (S') = { e , \$ }

Add $E \rightarrow b$ to M [E , b]

Construction Of Predictive Parsing Table

- For every LL grammar each parsing table entry uniquely identifies a production or signals an error
- For some grammars however M may have some entries that are multiply defined
- Such grammars are not LL(1) Grammar

Predictive Parsing Algorithm

```
Let a be the first symbol of w
Let X be the top of the Stack symbol
while ( X != $)
{
    if ( X == a)
        pop the stack and let 'a' be the next symbol of w
    else if ( X is a terminal )                                // X != a and X is terminal
        Error ( )
    else if ( M [ X , a ] is an error entry )
        Error ( )
    else if ( M [ X , a ] = Y1 Y2 ... Yk )
        Output the production X → Y1 Y2 ... Yk
        Pop the stack
        Push Yk Yk-1 ... Y1 onto the stack with Y1 on top
    Let X be the top stack symbol
}
```

Predictive Parsing Algorithm

Matched

Stack

E \$

Input

id + id * id \$

Action

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

Predictive Parsing Algorithm

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

Matched	Stack	Input	Action
	$E \$$	$id + id * id \$$	
	$TE' \$$	$id + id * id \$$	Output $E \rightarrow TE'$

Predictive Parsing Algorithm

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

Matched	Stack	Input	Action
	$E \$$	$id + id * id \$$	
	$TE' \$$	$id + id * id \$$	Output $E \rightarrow TE'$
	$FT'E' \$$	$id + id * id \$$	Output $T \rightarrow FT'$

Predictive Parsing Algorithm

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

Matched	Stack	Input	Action
	$E \$$	$id + id * id \$$	
	$TE' \$$	$id + id * id \$$	Output $E \rightarrow TE'$
	$FT'E' \$$	$id + id * id \$$	Output $T \rightarrow FT'$
	$idT'E' \$$	$id + id * id \$$	Output $F \rightarrow id$

Predictive Parsing Algorithm

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

Matched	Stack	Input	Action
	$E \$$	$id + id * id \$$	
	$TE' \$$	$id + id * id \$$	Output $E \rightarrow TE'$
	$FT'E' \$$	$id + id * id \$$	Output $T \rightarrow FT'$
	$idT'E' \$$	$id + id * id \$$	Output $F \rightarrow id$
id	$T'E' \$$	$+ id * id \$$	match id

Predictive Parsing Algorithm

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

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

Predictive Parsing Algorithm

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

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

Predictive Parsing Algorithm

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

Matched	Stack	Input	Action
id +	TE' \$	id * id \$	match +

Predictive Parsing Algorithm

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

Matched	Stack	Input	Action
id +	TE' \$	id * id \$	match +
id +	FT'E' \$	id * id \$	Output $T \rightarrow FT'$

Predictive Parsing Algorithm

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

Matched	Stack	Input	Action
id +	TE' \$	id * id \$	match +
id +	FT'E' \$	id * id \$	Output T \rightarrow FT'
id +	idT'E' \$	id * id \$	Output F \rightarrow id

Predictive Parsing Algorithm

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

Matched	Stack	Input	Action
$id +$	$TE' \$$	$id * id \$$	match +
$id +$	$FT'E' \$$	$id * id \$$	Output $T \rightarrow FT'$
$id +$	$idT'E' \$$	$id * id \$$	Output $F \rightarrow id$
$id + id$	$T'E' \$$	$* id \$$	match id

Predictive Parsing Algorithm

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

Matched	Stack	Input	Action
$id +$	$TE' \$$	$id * id \$$	match +
$id +$	$FT'E' \$$	$id * id \$$	Output $T \rightarrow FT'$
$id +$	$idT'E' \$$	$id * id \$$	Output $F \rightarrow id$
$id + id$	$T'E' \$$	$* id \$$	match id
$id + id$	$*FT'E' \$$	$* id \$$	Output $T' \rightarrow *FT'$

Predictive Parsing Algorithm

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

Matched	Stack	Input	Action
$id +$	$TE' \$$	$id * id \$$	match +
$id +$	$FT'E' \$$	$id * id \$$	Output $T \rightarrow FT'$
$id +$	$idT'E' \$$	$id * id \$$	Output $F \rightarrow id$
$id + id$	$T'E' \$$	$* id \$$	match id
$id + id$	$*FT'E' \$$	$* id \$$	Output $T' \rightarrow *FT'$
$id + id *$	$FT'E' \$$	$id \$$	match *

Predictive Parsing Algorithm

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

Matched	Stack	Input	Action
$id +$	$TE' \$$	$id * id \$$	match +
$id +$	$FT'E' \$$	$id * id \$$	Output $T \rightarrow FT'$
$id +$	$idT'E' \$$	$id * id \$$	Output $F \rightarrow id$
$id + id$	$T'E' \$$	$* id \$$	match id
$id + id$	$*FT'E' \$$	$* id \$$	Output $T' \rightarrow *FT'$
$id + id *$	$FT'E' \$$	$id \$$	match *
$id + id *$	$idT'E' \$$	$id \$$	Output $F \rightarrow id$

Predictive Parsing Algorithm

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

Matched	Stack	Input	Action
$id + id * id$	$T'E' \$$	$\$$	match id
$id + id * id$	$E' \$$	$\$$	Output $T' \rightarrow \epsilon$
$id + id * id$	$\$$	$\$$	Output $E' \rightarrow \epsilon$

SYNTAX ANALYSIS



BOTTOM-UP
PARSING

Bottom-Up Parsing

- Construction of Parse Tree for an input string beginning at leaves and working towards the root
- Can be visualized as reducing a string "w" to the start symbol of Grammar
- At each reduction step, a specific substring matching the body of production is replaced by Non-Terminal at the head of the production

Bottom-Up

Parsing

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

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

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

To reduce the string and move towards root symbol

$$\begin{aligned} id * id &\Rightarrow F * id \\ &\Rightarrow T * id \\ &\Rightarrow T * F \\ &\Rightarrow T \\ &\Rightarrow E \end{aligned}$$

Now if we write above derivation from bottom to top, we get rightmost derivation.

Conclusion: Bottom-Up parsing during left to right scan of the input constructs a rightmost derivation in reverse

Handle Pruning

A **Handle** is a substring that matches the body of the production and whose reduction represents one step along the reverse of Rightmost Derivation

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

Right Sentential Form	Handle	Reducing Production
$id1 * id2$	$id1$	$F \rightarrow id$
$F * id2$	F	$T \rightarrow F$
$T * id2$	$id2$	$F \rightarrow id$
$T * F$	$T * F$	$T \rightarrow T * F$
T	T	$E \rightarrow T$

Handle Pruning:

- We start with a string of terminals w to be parsed
- If w is a sentence of the grammar, then let w = γ_n

Where γ_n is the nth right sentential form of some unknown right derivation as

Handle Pruning

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n$$

- To reconstruct this derivation in reverse order, we locate the handle β_n in γ_n by relevant head of the production $A \rightarrow \beta_n$
- The β_n will be replaced by A to get previous right sentential form γ_{n-1}
- This process we called as Handle Pruning

Shift Reduce Parser

The Process:

Stack	Input
\$	w \$
...	
\$ S	\$

- Initially stack is empty and string w is on the input
- The parser operates by shifting zero or more input symbols onto stack until a handle is on top of stack
- The parser then reduces the handle to the left side of the appropriate production
- The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty

Shift Reduce Parser

Actions of a Shift Reduce Parser:-

1. **Shift:** Shift the next input symbol onto the top of the stack.
2. **Reduce:**
 - a. The right end of the string to be reduced must be at the top of the stack.
 - b. Locate the left end of the string within the stack and decide with what non-terminal to replace the string.
3. **Accept:** Announce successful completion of parsing.
4. **Error:** Discover a syntax error and call an error recovery routine.

Shift Reduce Parser

Input: id * id

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Stack	Input	Action
\$	id * id \$	Shift

Shift Reduce Parser

Input: id * id

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Stack	Input	Action
\$	id * id \$	Shift
\$ id	* id \$	Reduce by $F \rightarrow id$

Shift Reduce Parser

Input: id * id

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Stack	Input	Action
\$	id * id \$	Shift
\$ id	* id \$	Reduce by $F \rightarrow id$
\$ F	* id \$	Reduce by $T \rightarrow F$

Shift Reduce Parser

Input: id * id

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Stack	Input	Action
\$	id * id \$	Shift
\$ id	* id \$	Reduce by $F \rightarrow id$
\$ F	* id \$	Reduce by $T \rightarrow F$
\$ T	* id \$	Shift

Shift Reduce Parser

Input: id * id

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Stack	Input	Action
\$	id * id \$	Shift
\$ id	* id \$	Reduce by $F \rightarrow id$
\$ F	* id \$	Reduce by $T \rightarrow F$
\$ T	* id \$	Shift
\$ T *	id \$	Shift

Shift Reduce Parser

Input: id * id

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Stack	Input	Action
\$	id * id \$	Shift
\$ id	* id \$	Reduce by $F \rightarrow id$
\$ F	* id \$	Reduce by $T \rightarrow F$
\$ T	* id \$	Shift
\$ T *	id \$	Shift
\$ T * id	\$	Reduce by $F \rightarrow id$

Shift Reduce Parser

Input: id * id

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Stack	Input	Action
\$	id * id \$	Shift
\$ id	* id \$	Reduce by $F \rightarrow id$
\$ F	* id \$	Reduce by $T \rightarrow F$
\$ T	* id \$	Shift
\$ T *	id \$	Shift
\$ T * id	\$	Reduce by $F \rightarrow id$
\$ T * F	\$	Reduce by $T \rightarrow T * F$

Shift Reduce Parser

Input: id * id

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Stack	Input	Action
\$	id * id \$	Shift
\$ id	* id \$	Reduce by $F \rightarrow id$
\$ F	* id \$	Reduce by $T \rightarrow F$
\$ T	* id \$	Shift
\$ T *	id \$	Shift
\$ T * id	\$	Reduce by $F \rightarrow id$
\$ T * F	\$	Reduce by $T \rightarrow T * F$
\$ T	\$	Reduce by $E \rightarrow T$

Shift Reduce Parser

Input: id * id

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Stack	Input	Action
\$	id * id \$	Shift
\$ id	* id \$	Reduce by $F \rightarrow id$
\$ F	* id \$	Reduce by $T \rightarrow F$
\$ T	* id \$	Shift
\$ T *	id \$	Shift
\$ T * id	\$	Reduce by $F \rightarrow id$
\$ T * F	\$	Reduce by $T \rightarrow T * F$
\$ T	\$	Reduce by $E \rightarrow T$
\$ E	\$	Accept

Shift Reduce Parser

Input: id + id * id

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

Stack	Input	Action
\$	id + id * id \$	Shift

Shift Reduce Parser

Input: id + id * id

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

Stack	Input	Action
\$	id + id * id \$	Shift
\$ id	+ id * id \$	Reduce by $E \rightarrow id$

Shift Reduce Parser

Input: id + id * id

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

Stack	Input	Action
\$	id + id * id \$	Shift
\$ id	+ id * id \$	Reduce by $E \rightarrow id$
\$ E	+ id * id \$	Shift

Shift Reduce Parser

Input: id + id * id

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

Stack	Input	Action
\$	id + id * id \$	Shift
\$ id	+ id * id \$	Reduce by $E \rightarrow id$
\$ E	+ id * id \$	Shift
\$ E +	id * id \$	Shift

Shift Reduce Parser

Input: id + id * id

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

Stack	Input	Action
\$	id + id * id \$	Shift
\$ id	+ id * id \$	Reduce by $E \rightarrow id$
\$ E	+ id * id \$	Shift
\$ E +	id * id \$	Shift
\$ E + id	* id \$	Reduce by $E \rightarrow id$

Shift Reduce Parser

Input: id + id * id

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

Stack	Input	Action
\$	id + id * id \$	Shift
\$ id	+ id * id \$	Reduce by $E \rightarrow id$
\$ E	+ id * id \$	Shift
\$ E +	id * id \$	Shift
\$ E + id	* id \$	Reduce by $E \rightarrow id$
\$ E + E	* id \$	Shift (Here Shift-Reduce Conflict)

Shift Reduce Parser

Input: id + id * id

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

Stack	Input	Action
\$	id + id * id \$	Shift
\$ id	+ id * id \$	Reduce by $E \rightarrow id$
\$ E	+ id * id \$	Shift
\$ E +	id * id \$	Shift
\$ E + id	* id \$	Reduce by $E \rightarrow id$
\$ E + E	* id \$	Shift (Here Shift-Reduce Conflict)
\$ E + E *	id \$	Shift

Shift Reduce Parser

Input: id + id * id

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

Stack	Input	Action
\$	id + id * id \$	Shift
\$ id	+ id * id \$	Reduce by $E \rightarrow id$
\$ E	+ id * id \$	Shift
\$ E +	id * id \$	Shift
\$ E + id	* id \$	Reduce by $E \rightarrow id$
\$ E + E	* id \$	Shift (Here Shift-Reduce Conflict)
\$ E + E *	id \$	Shift
\$ E + E * id	\$	Reduce by $E \rightarrow id$

Shift Reduce Parser

Input: id + id * id

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

Stack	Input	Action
\$	id + id * id \$	Shift
\$ id	+ id * id \$	Reduce by $E \rightarrow id$
\$ E	+ id * id \$	Shift
\$ E +	id * id \$	Shift
\$ E + id	* id \$	Reduce by $E \rightarrow id$
\$ E + E	* id \$	Shift (Here Shift-Reduce Conflict)
\$ E + E *	id \$	Shift
\$ E + E * id	\$	Reduce by $E \rightarrow id$
\$ E + E * E	\$	Reduce by $E \rightarrow E * E$

Shift Reduce Parser

Input: id + id * id

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

Stack	Input	Action
\$	id + id * id \$	Shift
\$ id	+ id * id \$	Reduce by $E \rightarrow id$
\$ E	+ id * id \$	Shift
\$ E +	id * id \$	Shift
\$ E + id	* id \$	Reduce by $E \rightarrow id$
\$ E + E	* id \$	Shift (Here Shift-Reduce Conflict)
\$ E + E *	id \$	Shift
\$ E + E * id	\$	Reduce by $E \rightarrow id$
\$ E + <u>E * E</u>	\$	Reduce by $E \rightarrow E * E$
\$ E + E	\$	Reduce by $E \rightarrow E + E$

Shift Reduce Parser

Input: id + id * id

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

Stack	Input	Action
\$	id + id * id \$	Shift
\$ id	+ id * id \$	Reduce by $E \rightarrow id$
\$ E	+ id * id \$	Shift
\$ E +	id * id \$	Shift
\$ E + id	* id \$	Reduce by $E \rightarrow id$
\$ E + E	* id \$	Shift (Here Shift-Reduce Conflict)
\$ E + E *	id \$	Shift
\$ E + E * id	\$	Reduce by $E \rightarrow id$
\$ E + <u>E * E</u>	\$	Reduce by $E \rightarrow E * E$
\$ E + E	\$	Reduce by $E \rightarrow E + E$
\$ E	\$	Accept

SYNTAX ANALYSIS



BOTTOM-UP
PARSING

Operator Precedence Parser

- Operator precedence parser can be constructed from Operator Grammar
- Operator Grammar: The grammar which has property that no production on right side is ϵ or has two adjacent non-terminal
- Example:

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

Operator Precedance Parser

- There are three disjoint precedance relations
 - < less than
 - = Equal to
 - > Greater than
- Suppose there are two operators a and b then relations give following meaning:
 - $a < b$ – a gives precedance to b
 - $a = b$ – a has same precedance as of b
 - $a > b$ – a takes precedance over b

Operator Precedance Parser

Rules for finding operator precedance relations if a₁ & a₂ are operators

1. If a₁ has higher precedance than a₂ then make a₁ > a₂ and a₂ < a₁
2. If a₁ has equal precedance with a₂ and if operators are left associative then make a₁ > a₂ and a₂ > a₁
3. If a₁ has equal precedance with a₂ and if operators are right associative then make a₁ < a₂ and a₂ < a₁

Operator Precedance Parser

Rules for finding operator precedance relations if a₁ & a₂ are operators

4. For all operators a,

a < id and id > a

a < (and (< a

a >) and) > a

a > \$ and \$ < a

5. Operator ↑ has highest precedance and right associativity

6. Operator * and / has next higher precedance and left associativity

Operator Precedance Parser

Rules for finding operator precedance relations if a₁ & a₂ are operators

7. **Operator + and - has lowest precedance and left associativity**
8. The blank entries in operator precedance relation table indicates an error

Operator Precedance Relation Table

	+	-	*	/	↑	id	()	\$
+									
-									
*									
/									
↑									
id									
(
)									
\$									

Operator Precedance Relation Table

	+	-	*	/	↑	id	()	\$
+	>	>	<	<	<	<	<	>	>
-									
*									
/									
↑									
id									
(
)									
\$									

Operator Precedance Relation Table

	+	-	*	/	↑	id	()	\$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*									
/									
↑									
id									
(
)									
\$									

Operator Precedance Relation Table

	+	-	*	/	↑	id	()	\$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	>	>	<	<	<	>	>
/									
↑									
id									
(
)									
\$									

Operator Precedance Relation Table

	+	-	*	/	↑	id	()	\$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	>	>	<	<	<	>	>
/	>	>	>	>	<	<	<	>	>
↑									
id									
(
)									
\$									

Operator Precedance Relation Table

	+	-	*	/	\uparrow	id	()	\$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	>	>	<	<	<	>	>
/	>	>	>	>	<	<	<	>	>
\uparrow	>	>	>	>	<	<	<	>	>
id	>	>	>	>	>			>	>
(
)									
\$									

Operator Precedance Relation Table

	+	-	*	/	↑	id	()	\$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	>	>	>	<	<	>	>
/	>	>	>	>	>	<	<	>	>
↑	>	>	>	>	<	<	<	>	>
id	>	>	>	>	>			>	>
(<	<	<	<	<	<	<	=	
)									
\$									

Operator Precedance Relation Table

	+	-	*	/	\uparrow	id	()	\$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	>	>	>	<	<	>	>
/	>	>	>	>	>	<	<	>	>
\uparrow	>	>	>	>	<	<	<	>	>
id	>	>	>	>	>			>	>
(<	<	<	<	<	<	<	=	
)	>	>	>	>	>			>	>
\$									

Operator Precedence Relation Table

	+	-	*	/	\uparrow	id	()	\$
+	>	>	<	<	<	<	<	<	>
-	>	>	<	<	<	<	<	<	>
*	>	>	>	>	>	<	<	<	>
/	>	>	>	>	>	<	<	<	>
\uparrow	>	>	>	>	<	<	<	>	>
id	>	>	>	>	>			>	>
(<	<	<	<	<	<	<	=	
)	>	>	>	>	>	>		>	>
\$	<	<	<	<	<	<	<		Accept

Operator Precedence Algorithm

Input: A string w and table of precedence relations

Output: If w is well formed, a Skeletal parse tree, with a placeholder non-terminal E labeling all interior nodes otherwise an error indication

Method: Initially the stack contains $\$$ and the input buffer the string $w \$$

Operator Precedence Algorithm

```
Set ip to point to the first symbol of w$  
Repeat forever  
    If $ is on top of stack and ip points to $ then  
        Return  
    else begin  
        Let a be the topmost terminal symbol on top of stack  
        And let b be the symbol pointed to by ip;  
        If a < b or a = b then begin  
            Push b on the top of the stack  
            Advance ip to the next symbol  
        End  
        Else if a > b then  
            Repeat  
                Pop of the stack  
                Until the top stack terminal is related by <  
                to the terminal most recently popped  
        Else  
            Error( )  
    End
```

Operator Precedence Parser

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

Input: id + id * id

	id	+	*	\$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
\$	<	<	<	A

Operator Precedence Parser

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

Input: id + id * id

	id	+	*	\$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
\$	<	<	<	A

Stack	Relation	Input	Comment
\$	<	id + id * id \$	Push id

Operator Precedence Parser

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

Input: id + id * id

	id	+	*	\$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
\$	<	<	<	A

Stack	Relation	Input	Comment
\$	<	id + id * id \$	Push id
\$ id	>	+ id * id \$	Pop id

Operator Precedence Parser

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

Input: id + id * id

	id	+	*	\$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
\$	<	<	<	A

Stack	Relation	Input	Comment
\$	<	id + id * id \$	Push id
\$ id	>	+ id * id \$	Pop id
\$	<	+ id * id \$	Push +

Operator Precedence Parser

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

Input: id + id * id

	id	+	*	\$
id		>	>	>
+	<		>	<
*	<	>		>
\$	<	<	<	A

Stack	Relation	Input	Comment
\$	<	id + id * id \$	Push id
\$ id	>	+ id * id \$	Pop id
\$	<	+ id * id \$	Push +
\$ +	<	id * id \$	Push id

Operator Precedence Parser

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

Input: id + id * id

	id	+	*	\$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
\$	<	<	<	A

Stack	Relation	Input	Comment
\$	<	id + id * id \$	Push id
\$ id	>	+ id * id \$	Pop id
\$	<	+ id * id \$	Push +
\$ +	<	id * id \$	Push id
\$ + id	>	* id \$	Pop id

Operator Precedence Parser

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

Input: id + id * id

	id	+	*	\$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
\$	<	<	<	A

Stack	Relation	Input	Comment
\$	<	id + id * id \$	Push id
\$ id	>	+ id * id \$	Pop id
\$	<	+ id * id \$	Push +
\$ +	<	id * id \$	Push id
\$ + id	>	* id \$	Pop id
\$ +	<	* id \$	Push *

Operator Precedence Parser

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

Input: id + id * id

	id	+	*	\$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
\$	<	<	<	A

Stack	Relation	Input	Comment
\$	<	id + id * id \$	Push id
\$ id	>	+ id * id \$	Pop id
\$	<	+ id * id \$	Push +
\$ +	<	id * id \$	Push id
\$ + id	>	* id \$	Pop id
\$ +	<	* id \$	Push *
\$ + *	<	id \$	Push id

Operator Precedence Parser

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

Input: id + id * id

	id	+	*	\$
id		>	>	>
+	<		>	<
*	<		>	>
\$	<		<	A

Stack	Relation	Input	Comment
\$	<	id + id * id \$	Push id
\$ id	>	+ id * id \$	Pop id
\$	<	+ id * id \$	Push +
\$ +	<	id * id \$	Push id
\$ + id	>	* id \$	Pop id
\$ +	<	* id \$	Push *
\$ + *	<	id \$	Push id
\$ + * id	>	\$	Pop id

Operator Precedence Parser

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

Input: id + id * id

	id	+	*	\$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
\$	<	<	<	A

Stack	Relation	Input	Comment
\$	<	id + id * id \$	Push id
\$ id	>	+ id * id \$	Pop id
\$	<	+ id * id \$	Push +
\$ +	<	id * id \$	Push id
\$ + id	>	* id \$	Pop id
\$ +	<	* id \$	Push *
\$ + *	<	id \$	Push id
\$ + * id	>	\$	Pop id
\$ + *	>	\$	Pop *

Operator Precedence Parser

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

Input: id + id * id

	id	+	*	\$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
\$	<	<	<	A

Stack	Relation	Input	Comment
\$	<	id + id * id \$	Push id
\$ id	>	+ id * id \$	Pop id
\$	<	+ id * id \$	Push +
\$ +	<	id * id \$	Push id
\$ + id	>	* id \$	Pop id
\$ +	<	* id \$	Push *
\$ + *	<	id \$	Push id
\$ + * id	>	\$	Pop id
\$ + *	>	\$	Pop *
\$ +	>	\$	Pop +

Operator Precedence Parser

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

Input: id + id * id

	id	+	*	\$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
\$	<	<	<	A

Stack	Relation	Input	Comment
\$	<	id + id * id \$	Push id
\$ id	>	+ id * id \$	Pop id
\$	<	+ id * id \$	Push +
\$ +	<	id * id \$	Push id
\$ + id	>	* id \$	Pop id
\$ +	<	* id \$	Push *
\$ + *	<	id \$	Push id
\$ + * id	>	\$	Pop id
\$ + *	>	\$	Pop *
\$ +	>	\$	Pop +
\$	Accept	\$	Accept

Operator Precedence Parser

Steps to construct syntax tree (Expression Tree):

1. Keep track of elements that are popped in the same order.
Consider that sequence as input string for processing using stack.
2. Read the processing sequence from left to right
3. If operand (identifier) is found, then push that onto stack
4. If an operator is found, then pop out top 2 elements and construct subtree with operator as root node.
5. Push this newly constructed subtree on top of stack
6. Repeat step 3 to 5 until all the symbols in input string are processed.
7. When input string is completed then pop out topmost element of stack.
8. If stack is not empty after pop then declare ERROR otherwise POPPED ELEMENT is final SYNTAX TREE

Operator Precedence Parser

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

Input: id + id * id

Comment
Push id
Pop id
Push +
Push id
Pop id
Push *
Push id
Pop id
Pop *
Pop +
Accept

Operator Precedence Parser

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

Input: id + id * id

Comment
Pop id (1)
Pop id (2)
Pop id (3)
Pop * (4)
Pop + (5)
Accept

Sequence to be processed:

id , id , id, *, +

Operator Precedence Parser

$E \rightarrow E + E \mid E - E \mid E^* E \mid E/E \mid E \uparrow E \mid (E) \mid id$

Input: id + id * id

Stack

Sequence to be processed:

id , id , id, *, +

Operator Precedence Parser

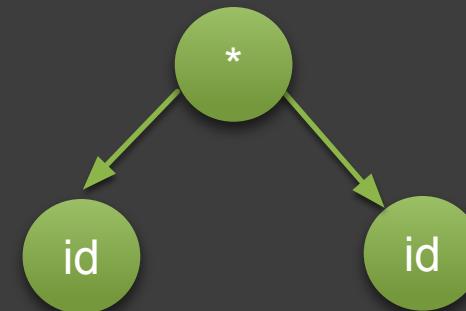
$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid id$

Input: id + id * id

Stack

Sequence to be processed:

id , id , id, *, +



Sub Tree 1

Operator Precedence Parser

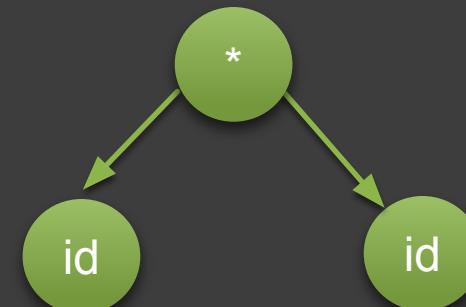
$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid id$

Input: id + id * id

Stack
Sub Tree 1
id
\$

Sequence to be processed:

id , id , id, *, +



Sub Tree 1

Operator Precedence Parser

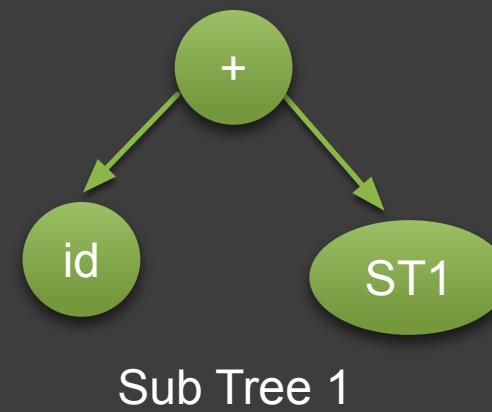
$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid id$

Input: id + id * id

Stack
\$

Sequence to be processed:

id , id , id, *, +



Operator Precedence Parser

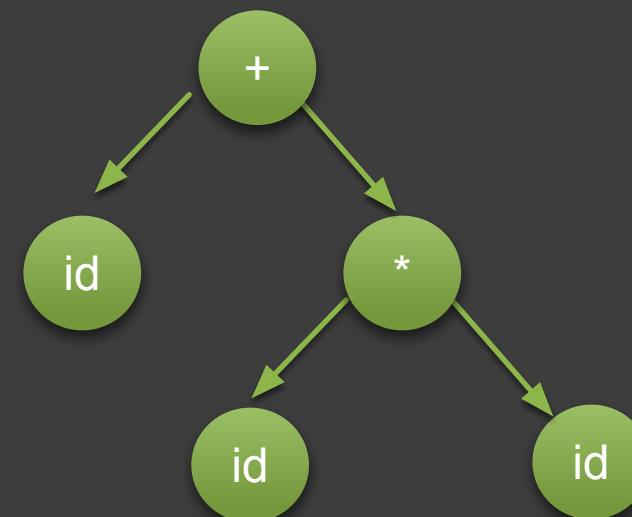
$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid id$

Input: id + id * id

Stack
\$

Sequence to be processed:

id , id , id, *, +



Sub Tree 2

Operator Precedence Parser

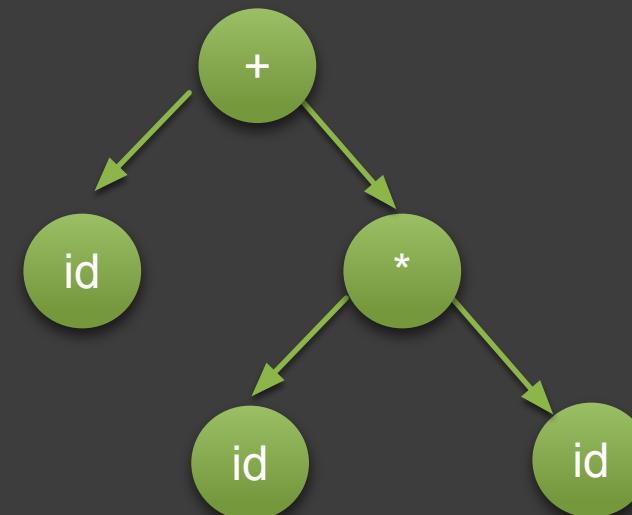
$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid id$

Input: id + id * id

Stack
Sub Tree 2
\$

Sequence to be processed:

id , id , id, *, +



Sub Tree 2

Operator Precedence Parser

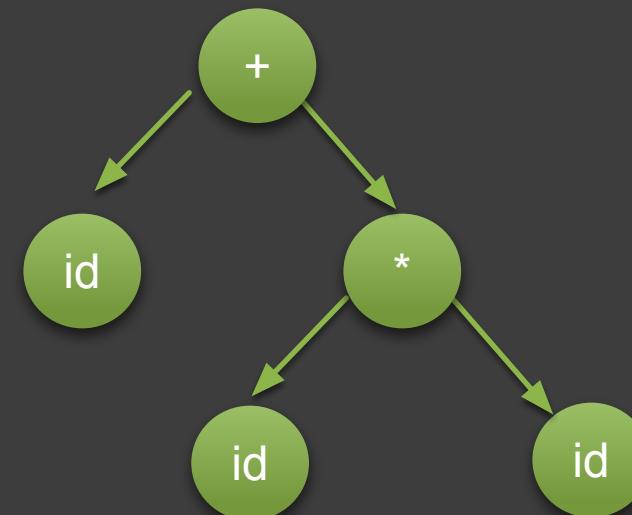
$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid id$

Input: id + id * id

Stack
\$

Sequence to be processed:

id , id , id, *, +



Final Tree

Operator Precedence Parser

Steps to construct Entire Derivation Tree:

1. Keep track of elements that are popped in the reverse order. (Start from Bottom and move in Top Direction)
2. Select start symbol as a root node.
3. Select the Next unprocessed symbol from list of popped elements.
4. Generate sub tree in which this symbol act as a leaf node by choosing one (or set) of the suitable production
5. Select rightmost node of this newly generated subtree such that it is non terminal. This node will act as a root node.
6. Repeat step 3 to 5 until all the symbols in popped sequence are processed.
7. The final tree is Required Derivation Tree

Operator Precedence Parser

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

Input: id + id * id

Construction of Entire Tree:

Comment
Pop id (5)
Pop id (4)
Pop id (3)
Pop * (2)
Pop + (1)
Accept

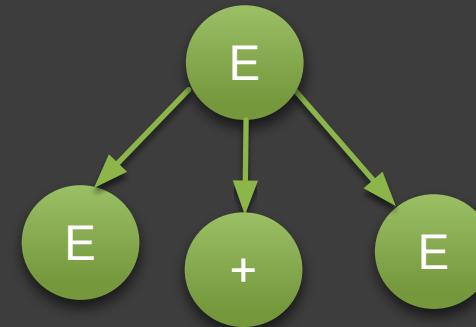
Operator Precedence Parser

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

Input: id + id * id

Comment
Pop id (5)
Pop id (4)
Pop id (3)
Pop * (2)
Pop + (1): $E \rightarrow E + E$
Accept

Construction of Entire Tree:



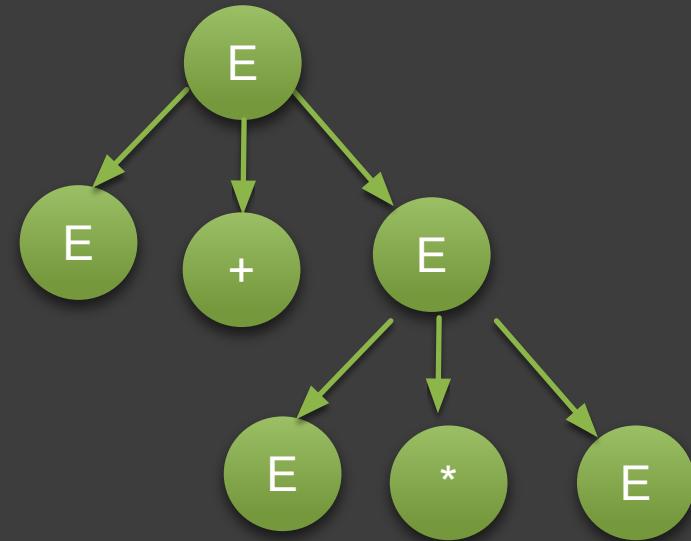
Operator Precedence Parser

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

Input: id + id * id

Comment
Pop id (5)
Pop id (4)
Pop id (3)
Pop * (2) : $E \rightarrow E * E$
Pop + (1): $E \rightarrow E + E$
Accept

Construction of Entire Tree:



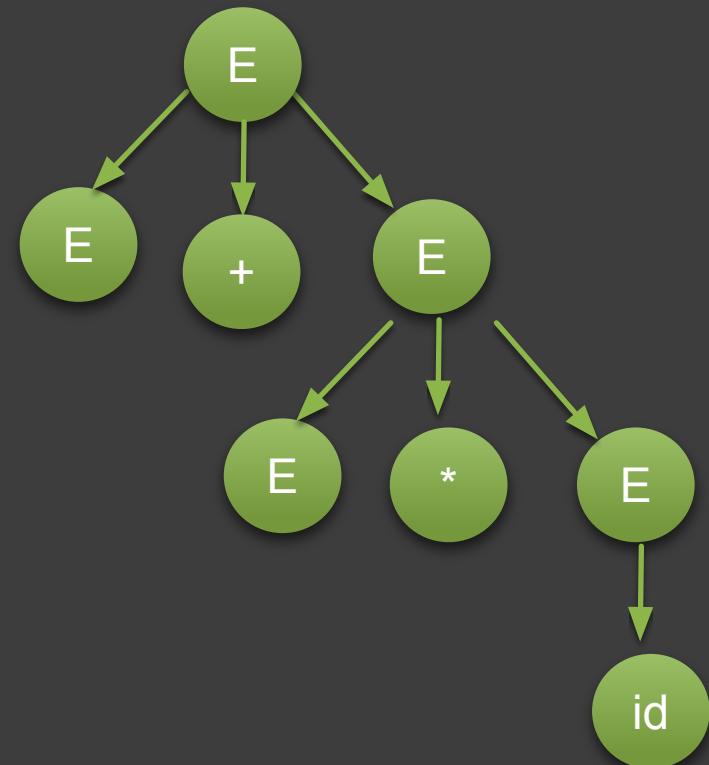
Operator Precedence Parser

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

Input: id + id * id

Comment
Pop id (5)
Pop id (4)
Pop id (3) : $E \rightarrow id$
Pop * (2) : $E \rightarrow E * E$
Pop + (1): $E \rightarrow E + E$
Accept

Construction of Entire Tree:



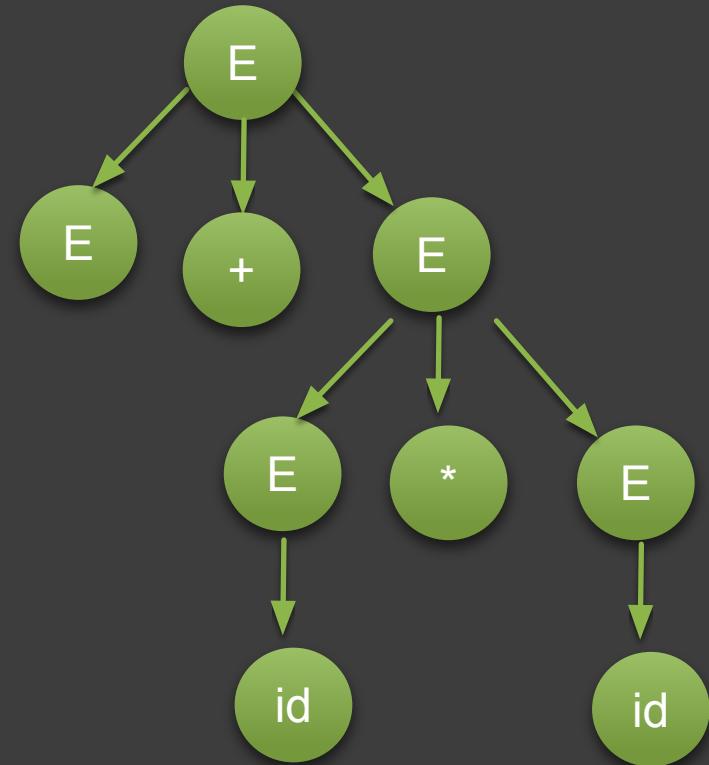
Operator Precedence Parser

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

Input: id + id * id

Comment
Pop id (5)
Pop id (4): $E \rightarrow id$
Pop id (3) : $E \rightarrow id$
Pop * (2) : $E \rightarrow E * E$
Pop + (1): $E \rightarrow E + E$
Accept

Construction of Entire Tree:



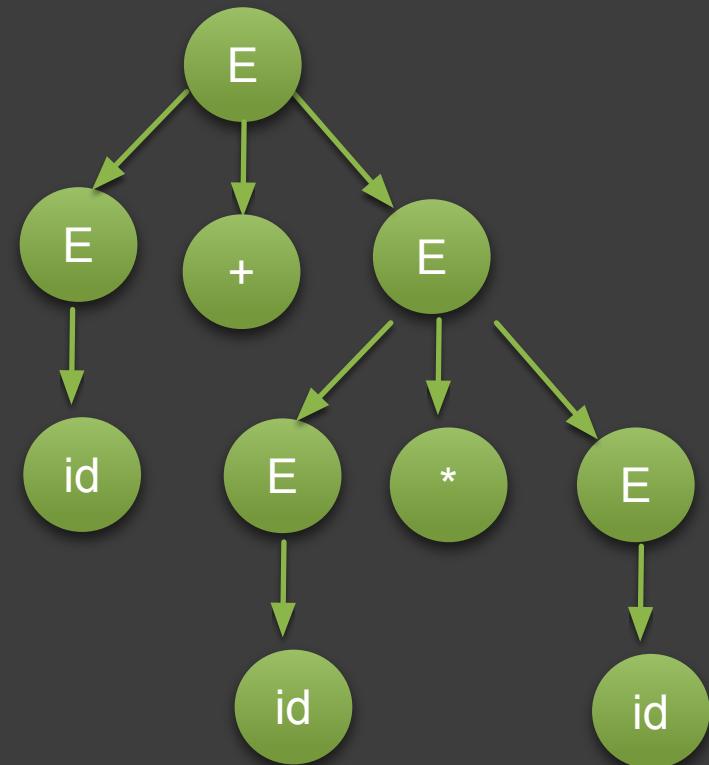
Operator Precedence Parser

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

Input: id + id * id

Comment
Pop id (5): $E \rightarrow id$
Pop id (4): $E \rightarrow id$
Pop id (3) : $E \rightarrow id$
Pop * (2) : $E \rightarrow E * E$
Pop + (1): $E \rightarrow E + E$
Accept

Construction of Entire Tree:



Operator Precedence Parser

$E \rightarrow E + T \mid T$

$T \rightarrow T * V \mid V$

$V \rightarrow a \mid b \mid c \mid d$

Input: $a + b * c * d$

	a	b	c	d	+	*	\$
a					>	>	>
b					>	>	>
c					>	>	>
d					>	>	>
+	<	<	<	<	>	<	>
*	<	<	<	<	>	>	>
\$	<	<	<	<	<	<	A

Operator Precedence Parser

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * V \mid V$$
$$V \rightarrow a \mid b \mid c \mid d$$

Input: $a + b * c * d$

Stack	Relation	Input	Comment
\$	<	$a + b * c * d \$$	Push a

Operator Precedence Parser

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * V \mid V$$
$$V \rightarrow a \mid b \mid c \mid d$$

Input: $a + b * c * d$

Stack	Relation	Input	Comment
\$	<	$a + b * c * d \$$	Push a
\$ a	>	$+ b * c * d \$$	Pop a

Operator Precedence Parser

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * V \mid V$$
$$V \rightarrow a \mid b \mid c \mid d$$

Input: $a + b * c * d$

Stack	Relation	Input	Comment
\$	<	$a + b * c * d \$$	Push a
\$ a	>	$+ b * c * d \$$	Pop a
\$	<	$+ b * c * d \$$	Push +

Operator Precedence Parser

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * V \mid V$$
$$V \rightarrow a \mid b \mid c \mid d$$

Input: $a + b * c * d$

Stack	Relation	Input	Comment
\$	<	$a + b * c * d \$$	Push a
\$ a	>	$+ b * c * d \$$	Pop a
\$	<	$+ b * c * d \$$	Push +
\$ +	<	$b * c * d \$$	Push b

Operator Precedence Parser

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * V \mid V$$
$$V \rightarrow a \mid b \mid c \mid d$$

Input: $a + b * c * d$

Stack	Relation	Input	Comment
\$	<	$a + b * c * d \$$	Push a
\$ a	>	$+ b * c * d \$$	Pop a
\$	<	$+ b * c * d \$$	Push +
\$ +	<	$b * c * d \$$	Push b
\$ + b	>	$* c * d \$$	Pop b

Operator Precedence Parser

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * V \mid V$$
$$V \rightarrow a \mid b \mid c \mid d$$

Input: $a + b * c * d$

Stack	Relation	Input	Comment
\$	<	$a + b * c * d \$$	Push a
\$ a	>	$+ b * c * d \$$	Pop a
\$	<	$+ b * c * d \$$	Push +
\$ +	<	$b * c * d \$$	Push b
\$ + b	>	$* c * d \$$	Pop b
\$ +	<	$* c * d \$$	Push *

Operator Precedence Parser

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * V \mid V$$
$$V \rightarrow a \mid b \mid c \mid d$$

Input: $a + b * c * d$

Stack	Relation	Input	Comment
\$	<	$a + b * c * d \$$	Push a
\$ a	>	$+ b * c * d \$$	Pop a
\$	<	$+ b * c * d \$$	Push +
\$ +	<	$b * c * d \$$	Push b
\$ + b	>	$* c * d \$$	Pop b
\$ +	<	$* c * d \$$	Push *
\$ + *	<	$c * d \$$	Push c

Operator Precedence Parser

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * V \mid V$$
$$V \rightarrow a \mid b \mid c \mid d$$

Input: $a + b * c * d$

Stack	Relation	Input	Comment
\$	<	$a + b * c * d \$$	Push a
\$ a	>	$+ b * c * d \$$	Pop a
\$	<	$+ b * c * d \$$	Push +
\$ +	<	$b * c * d \$$	Push b
\$ + b	>	$* c * d \$$	Pop b
\$ +	<	$* c * d \$$	Push *
\$ + *	<	$c * d \$$	Push c
\$ + * c	>	$* d \$$	Pop c

Operator Precedence Parser

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * V \mid V$$
$$V \rightarrow a \mid b \mid c \mid d$$

Input: $a + b * c * d$

Stack	Relation	Input	Comment
\$	<	$a + b * c * d \$$	Push a
\$ a	>	$+ b * c * d \$$	Pop a
\$	<	$+ b * c * d \$$	Push +
\$ +	<	$b * c * d \$$	Push b
\$ + b	>	$* c * d \$$	Pop b
\$ +	<	$* c * d \$$	Push *
\$ + *	<	$c * d \$$	Push c
\$ + * c	>	$* d \$$	Pop c
\$ + *	>	$* d \$$	Pop *

Operator Precedence Parser

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * V \mid V$$
$$V \rightarrow a \mid b \mid c \mid d$$

Input: $a + b * c * d$

Stack	Relation	Input	Comment
\$	<	$a + b * c * d \$$	Push a
\$ a	>	$+ b * c * d \$$	Pop a
\$	<	$+ b * c * d \$$	Push +
\$ +	<	$b * c * d \$$	Push b
\$ + b	>	$* c * d \$$	Pop b
\$ +	<	$* c * d \$$	Push *
\$ + *	<	$c * d \$$	Push c
\$ + * c	>	$* d \$$	Pop c
\$ + *	>	$* d \$$	Pop *
\$ +	<	$* d \$$	Push *

Operator Precedence Parser

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

$$T \rightarrow T * V \mid V$$

$$V \rightarrow a \mid b \mid c \mid d$$

Input: $a + b * c * d$

Stack	Relation	Input	Comment
\$	<	$a + b * c * d \$$	Push a
\$ a	>	$+ b * c * d \$$	Pop a
\$	<	$+ b * c * d \$$	Push +
\$ +	<	$b * c * d \$$	Push b
\$ + b	>	$* c * d \$$	Pop b
\$ +	<	$* c * d \$$	Push *
\$ + *	<	$c * d \$$	Push c
\$ + * c	>	$* d \$$	Pop c
\$ + *	>	$* d \$$	Pop *
\$ +	<	$* d \$$	Push *
\$ + *	<	$d \$$	Push d

Operator Precedence Parser

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

$$T \rightarrow T * V \mid V$$

$$V \rightarrow a \mid b \mid c \mid d$$

Input: $a + b * c * d$

Stack	Relation	Input	Comment
\$	<	$a + b * c * d \$$	Push a
\$ a	>	$+ b * c * d \$$	Pop a
\$	<	$+ b * c * d \$$	Push +
\$ +	<	$b * c * d \$$	Push b
\$ + b	>	$* c * d \$$	Pop b
\$ +	<	$* c * d \$$	Push *
\$ + *	<	$c * d \$$	Push c
\$ + * c	>	$* d \$$	Pop c
\$ + *	>	$* d \$$	Pop *
\$ +	<	$* d \$$	Push *
\$ + *	<	$d \$$	Push d
\$ + * d	>	\$	Pop d

Operator Precedence Parser

 $E \rightarrow E + T \mid T$
 $T \rightarrow T * V \mid V$
 $V \rightarrow a \mid b \mid c \mid d$

 Input: $a + b * c * d$

Stack	Relation	Input	Comment
\$	<	$a + b * c * d \$$	Push a
\$ a	>	$+ b * c * d \$$	Pop a
\$	<	$+ b * c * d \$$	Push +
\$ +	<	$b * c * d \$$	Push b
\$ + b	>	$* c * d \$$	Pop b
\$ +	<	$* c * d \$$	Push *
\$ + *	<	$c * d \$$	Push c
\$ + * c	>	$* d \$$	Pop c
\$ + *	>	$* d \$$	Pop *
\$ +	<	$* d \$$	Push *
\$ + *	<	$d \$$	Push d
\$ + * d	>	\$	Pop d
\$ + *	>	\$	Pop *

$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b^* c^* d$

Stack	Relation	Input	Comment
\$	<	$a + b^* c^* d \$$	Push a
\$ a	>	$+ b^* c^* d \$$	Pop a
\$	<	$+ b^* c^* d \$$	Push +
\$ +	<	$b^* c^* d \$$	Push b
\$ + b	>	$* c^* d \$$	Pop b
\$ +	<	$* c^* d \$$	Push *
\$ + *	<	$c^* d \$$	Push c
\$ + * c	>	$* d \$$	Pop c
\$ + *	>	$* d \$$	Pop *
\$ +	<	$* d \$$	Push *
\$ + *	<	$d \$$	Push d
\$ + * d	>	\$	Pop d
\$ + *	>	\$	Pop *
\$ +	>	\$	Pop +

$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b^* c^* d$

Stack	Relation	Input	Comment
\$	<	$a + b^* c^* d \$$	Push a
\$ a	>	$+ b^* c^* d \$$	Pop a
\$	<	$+ b^* c^* d \$$	Push +
\$ +	<	$b^* c^* d \$$	Push b
\$ + b	>	$* c^* d \$$	Pop b
\$ +	<	$* c^* d \$$	Push *
\$ + *	<	$c^* d \$$	Push c
\$ + * c	>	$* d \$$	Pop c
\$ + *	>	$* d \$$	Pop *
\$ +	<	$* d \$$	Push *
\$ + *	<	$d \$$	Push d
\$ + * d	>	\$	Pop d
\$ + *	>	\$	Pop *
\$ +	>	\$	Pop +
\$	Accept	\$	Accept

$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$

Input: a + b * c * d

Comment
Push a
Pop a
Push +
Push b
Pop b
Push *
Push c
Pop c
Pop *
Push *
Push d
Pop d
Pop *
Pop +
Accept

$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Comment
Pop a (1)
Pop b (2)
Pop c (3)
Pop * (4)
Pop d (5)
Pop * (6)
Pop + (7)
Accept

Sequence to be processed:

 $a, b, c, *, d, *, +$

$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$

Input: a + b * c * d

Sequence to be processed:

a , b , c , * , d , * , +

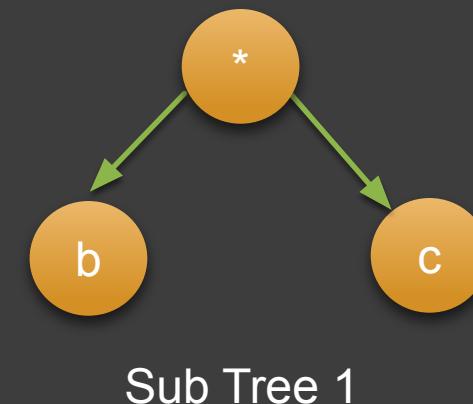
Stack
c
b
a
\$

$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Sequence to be processed:

 $a, b, c, *, d, *, +$

Stack
a
$\$$



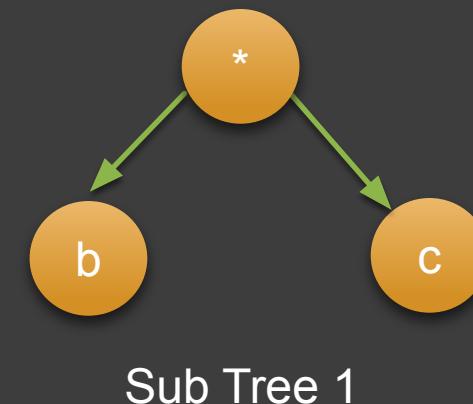
$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$

Input: a + b * c * d

Sequence to be processed:

a , b , c , * , d , * , +

Stack
Sub Tree 1
a
\$



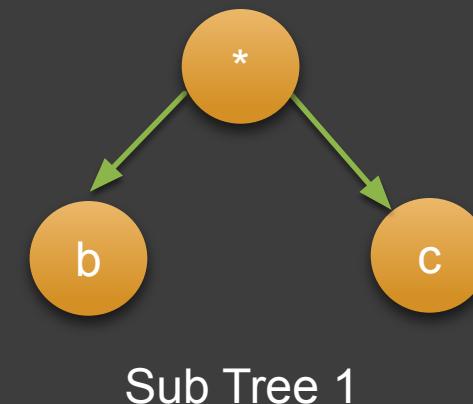
$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$

Input: a + b * c * d

Sequence to be processed:

a , b , c , * , d , * , +

Stack
d
Sub Tree 1
a
\$

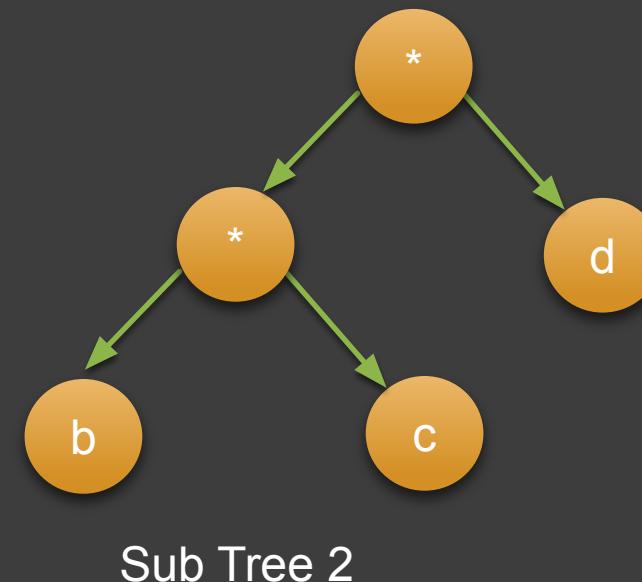


$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Sequence to be processed:

 $a, b, c, *, d, *, +$

Stack
a
$\$$



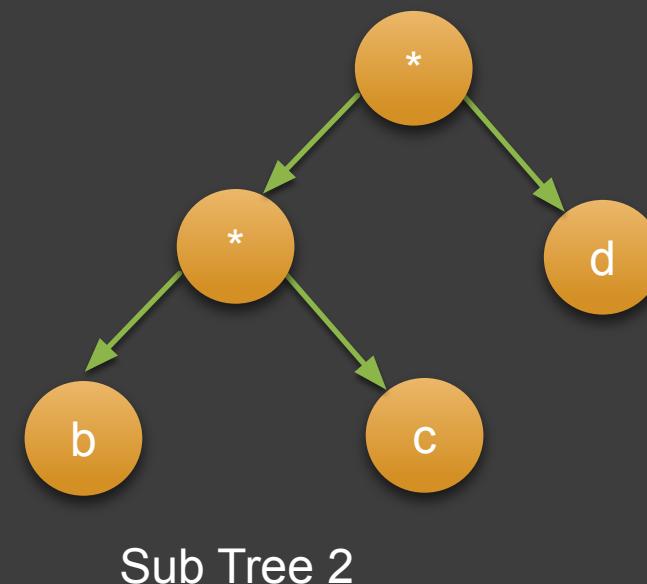
$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$

Input: a + b * c * d

Sequence to be processed:

a , b , c , * , d , * , +

Stack
Sub Tree 2
a
\$

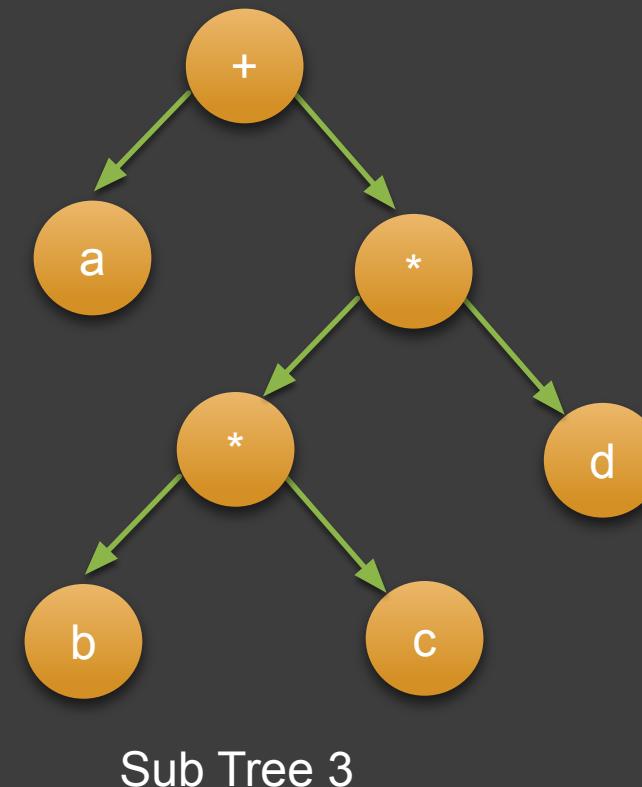


$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Sequence to be processed:

 $a, b, c, *, d, *, +$

Stack
\$



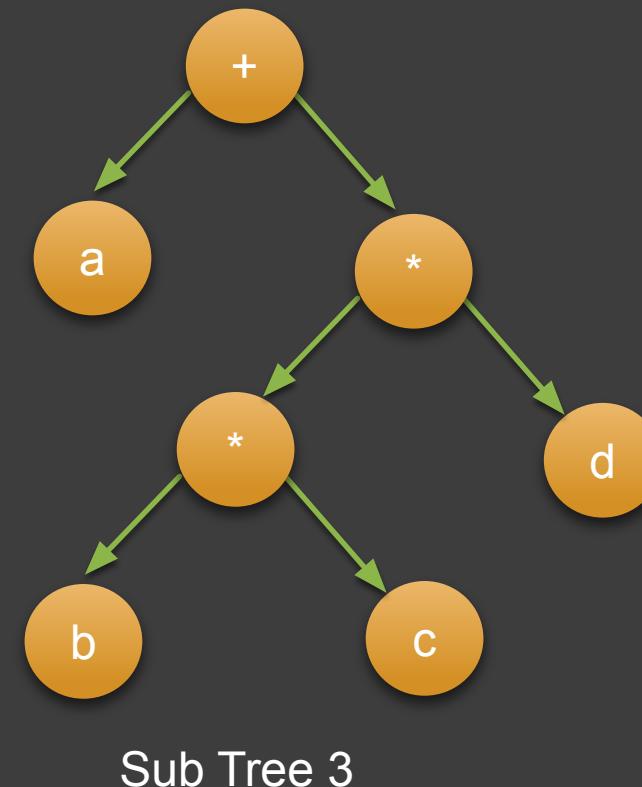
$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$

Input: a + b * c * d

Sequence to be processed:

a , b , c , * , d , * , +

Stack
Sub Tree 3
\$

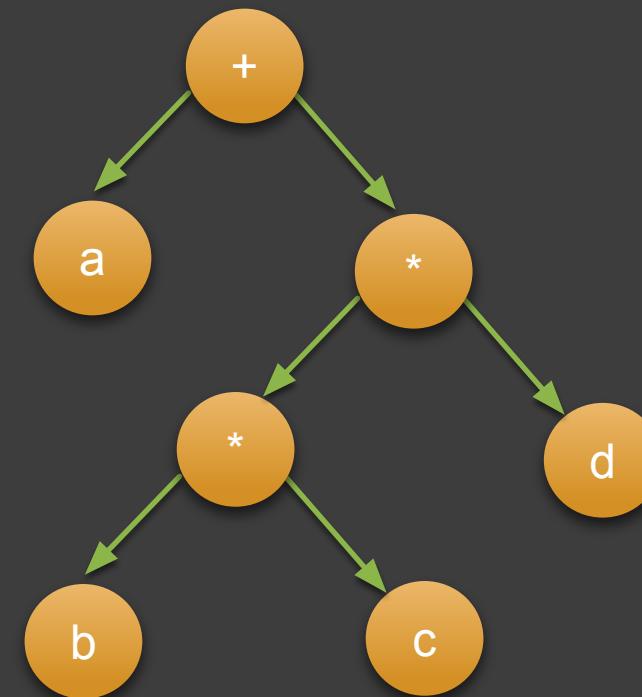


$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Sequence to be processed:

 $a, b, c, *, d, *, +$

Stack
\$

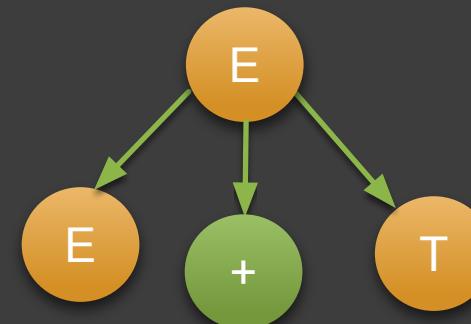


Final Tree

$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Comment
Pop a (7)
Pop b (6)
Pop c (5)
Pop * (4)
Pop d (3)
Pop * (2)
Pop + (1) : $E \rightarrow E + T$
Accept

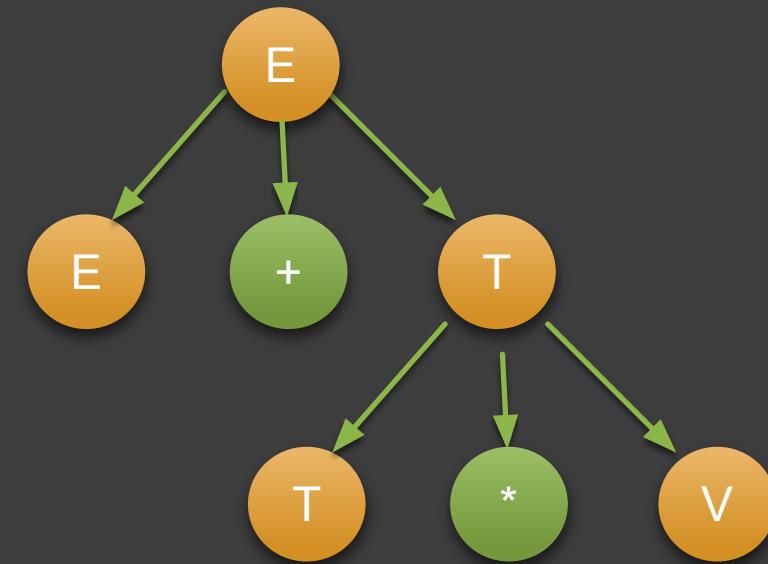
Construction of Entire Tree:



$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Comment
Pop a (7)
Pop b (6)
Pop c (5)
Pop * (4)
Pop d (3)
Pop * (2) : $T \rightarrow T^* V$
Pop + (1) : $E \rightarrow E + T$
Accept

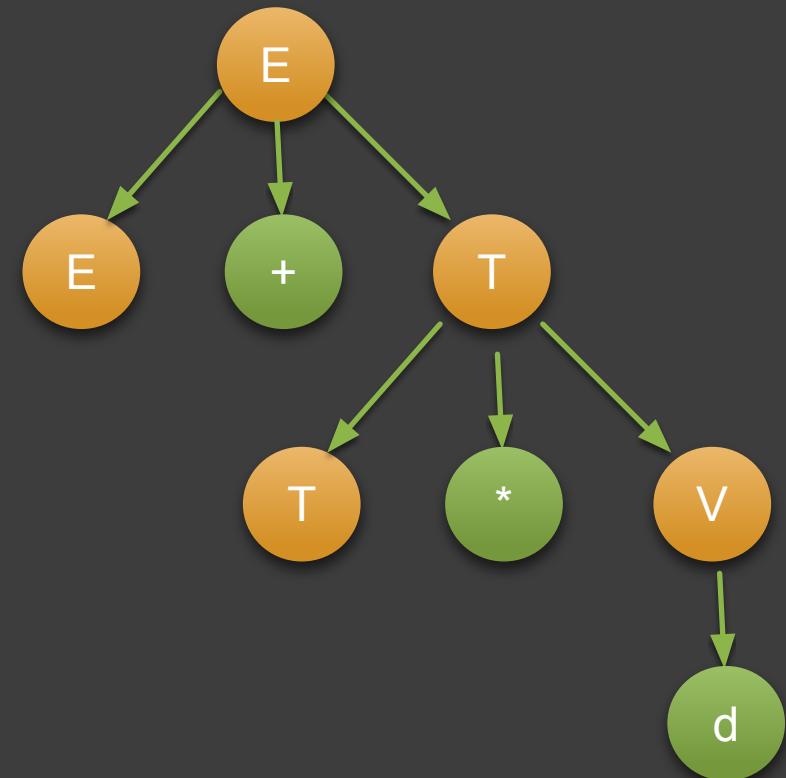
Construction of Entire Tree:



$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Comment
Pop a (7)
Pop b (6)
Pop c (5)
Pop * (4)
Pop d (3) : $V \rightarrow d$
Pop * (2) : $T \rightarrow T^* V$
Pop + (1) : $E \rightarrow E + T$
Accept

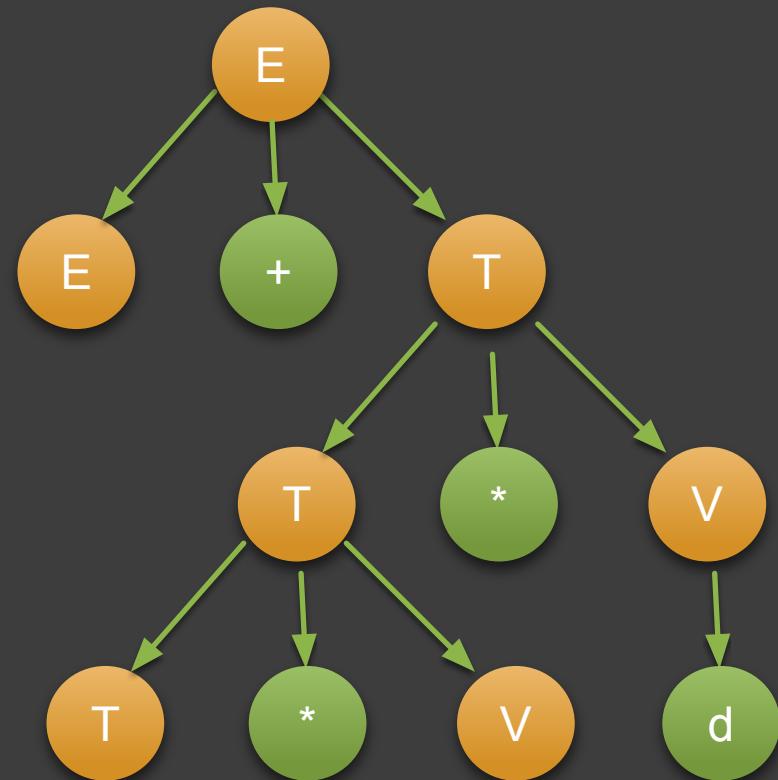
Construction of Entire Tree:



$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Comment
Pop a (7)
Pop b (6)
Pop c (5)
Pop * (4) : $T \rightarrow T^* V$
Pop d (3) : $V \rightarrow d$
Pop * (2) : $T \rightarrow T^* V$
Pop + (1) : $E \rightarrow E + T$
Accept

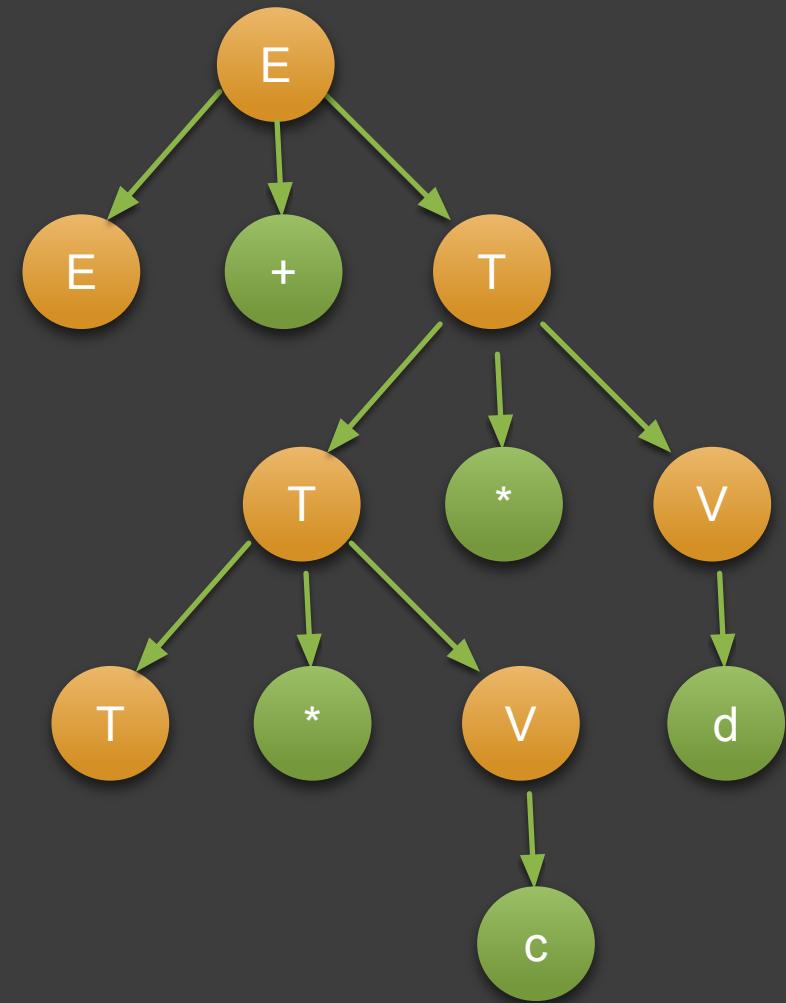
Construction of Entire Tree:



$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Comment
Pop a (7)
Pop b (6)
Pop c (5) : $V \rightarrow c$
Pop * (4) : $T \rightarrow T^* V$
Pop d (3) : $V \rightarrow d$
Pop * (2) : $T \rightarrow T^* V$
Pop + (1) : $E \rightarrow E + T$
Accept

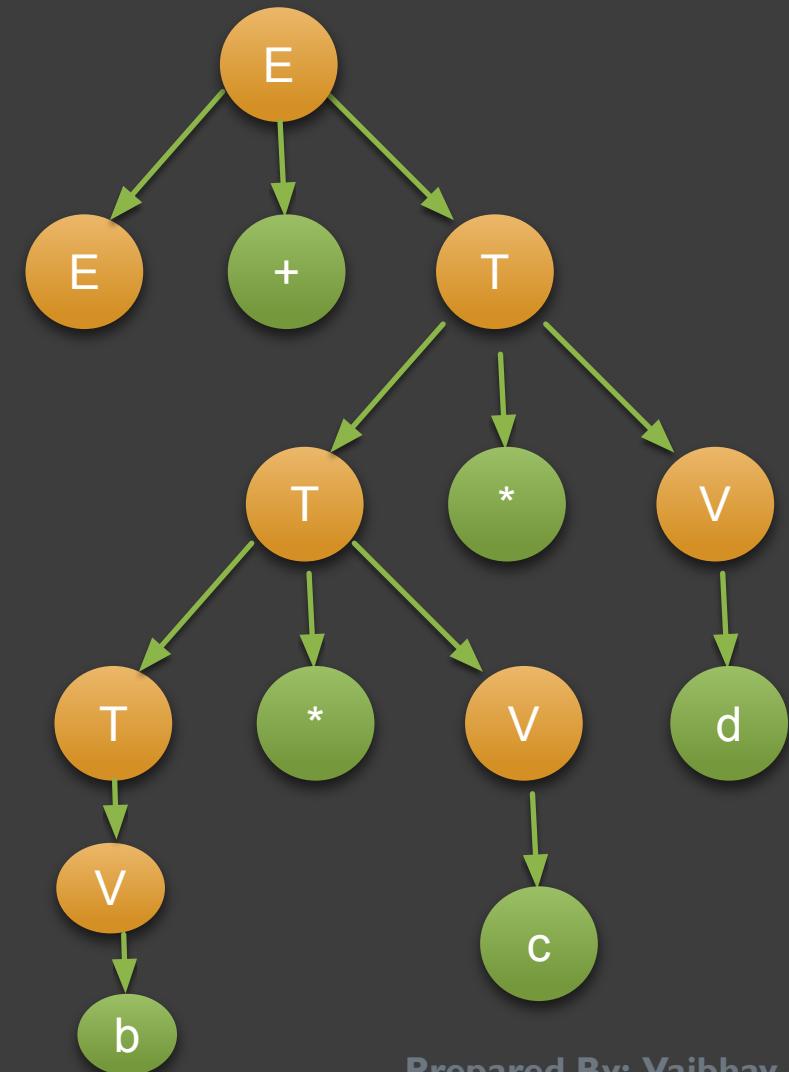
Construction of Entire Tree:



$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Comment
Pop a (7)
Pop b (6) : $T \rightarrow V, V \rightarrow b$
Pop c (5) : $V \rightarrow c$
Pop * (4) : $T \rightarrow T^* V$
Pop d (3) : $V \rightarrow d$
Pop * (2) : $T \rightarrow T^* V$
Pop + (1) : $E \rightarrow E + T$
Accept

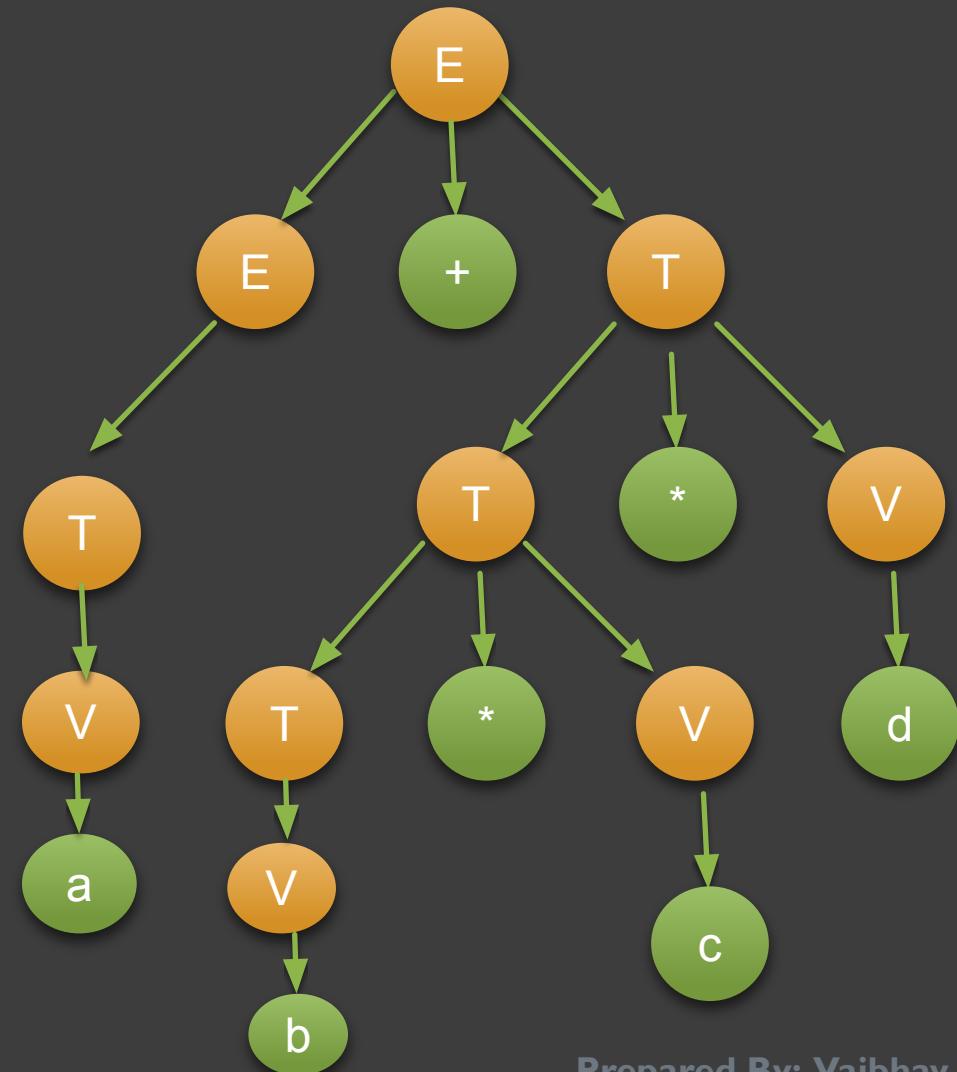
Construction of Entire Tree:



$E \rightarrow E + T \mid T$ $T \rightarrow T^* V \mid V$ $V \rightarrow a \mid b \mid c \mid d$ Input: $a + b * c * d$

Comment
Pop a (7) : $E \rightarrow T, T \rightarrow V, V \rightarrow a$
Pop b (6) : $T \rightarrow V, V \rightarrow b$
Pop c (5) : $V \rightarrow c$
Pop * (4) : $T \rightarrow T^* V$
Pop d (3) : $V \rightarrow d$
Pop * (2) : $T \rightarrow T^* V$
Pop + (1) : $E \rightarrow E + T$
Accept

Construction of Entire Tree:





■ SLR PARSER

SYNTAX ANALYSIS

Introduction To LR Parser

- Most prevalent type of Bottom-Up Parser
- Known as LR (k) parser
 - L stands for Left to Right scanning of input
 - R stands for Rightmost Derivation in reverse
 - k used for number of input symbols of look ahead for making parsing decisions
- k = 0 or k = 1 is used for practical interest.
- When k is omitted then it is considered as 1
- Examples: SLR, Canonical LR and LALR

Why LR Parser??

- Table driven similar to Non recursive LL parser
- They can be constructed to recognize virtually all programming language construct for which context free grammar can be written
- Parsing method is the most General non-backtracking Shift Reduce
- Parsing Method can be implemented efficiently
- Can detect syntactic error at earliest from left to right scan
- The class of grammar that can be parsed using LR methods is the proper SUPERSET of the class of the grammar that can be parsed using LL Method

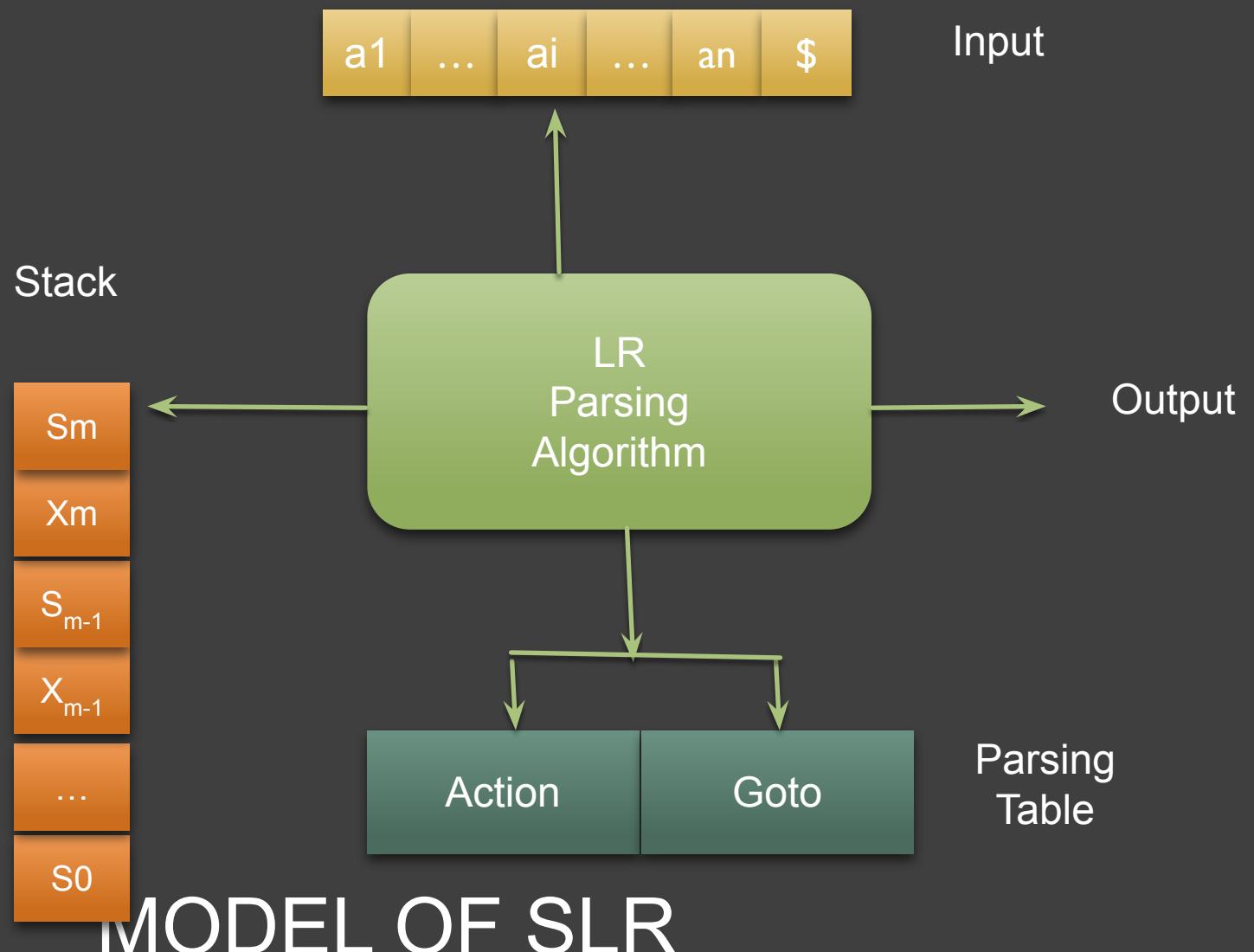
Why LR Parser??

Drawback:

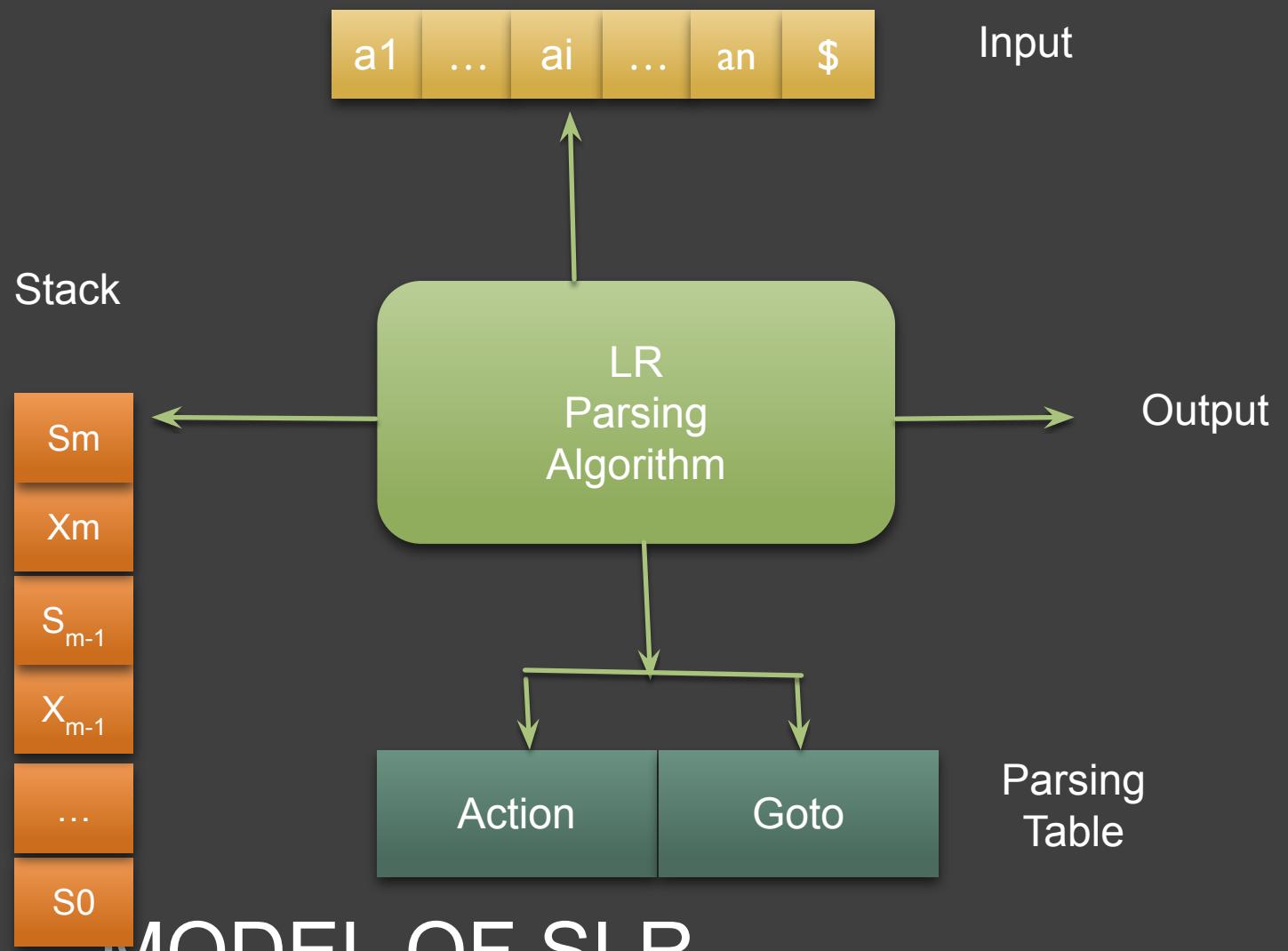
- Too much work to construct LR Parser by hand for typical programming-language grammar

Solution: LR Parser generator is needed.

Example: YACC tool



MODEL OF SLR PARSER



The program reads $[S_m, a_i]$

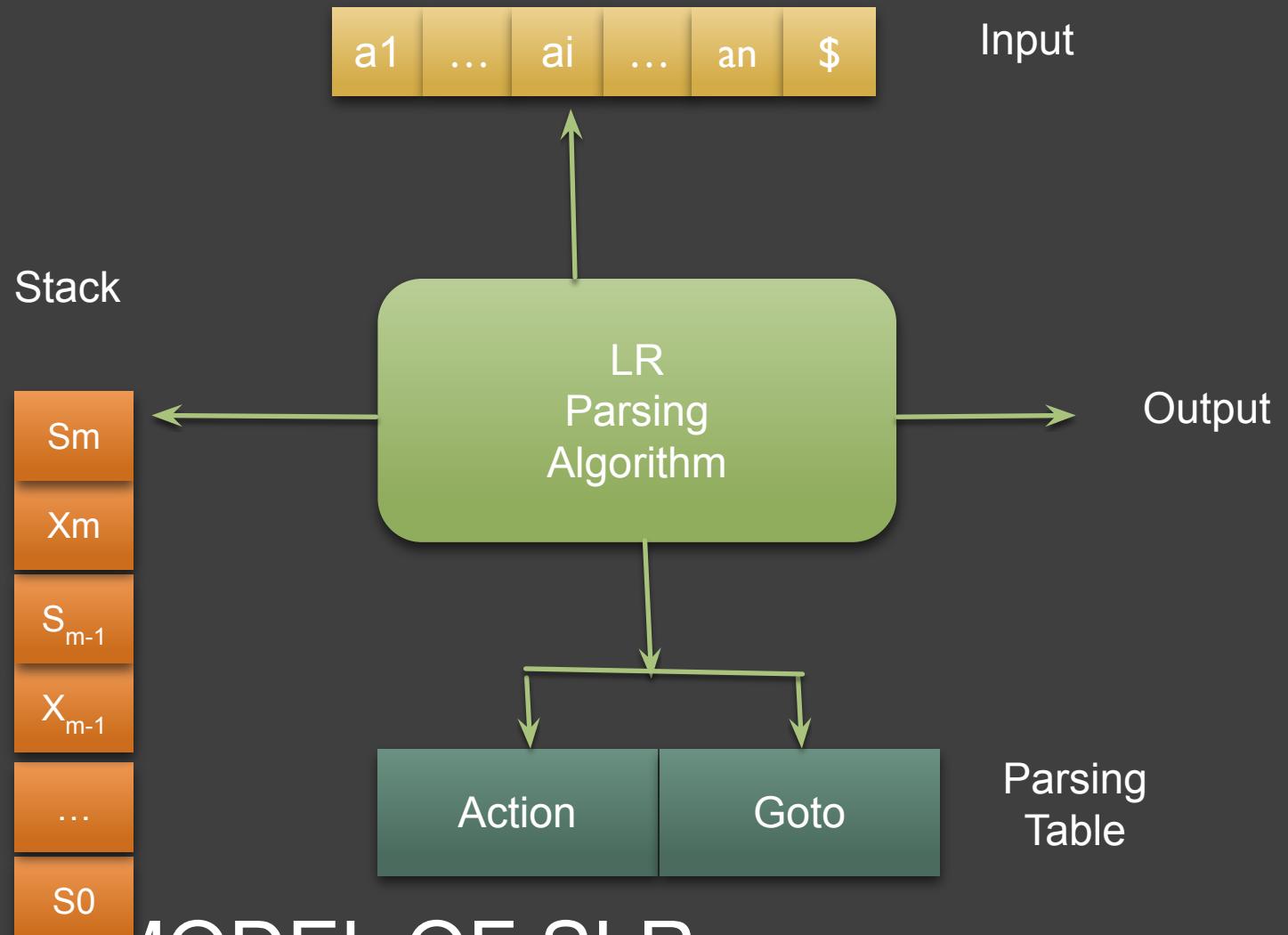
Where S_m : Top of the stack

a_i : Current input symbol

Four Possible Actions are:

1. Shift S , where S is a state
2. Reduce by a production $A \rightarrow b$
3. Accept
4. Error

MODEL OF SLR PARSER



The function Goto takes a state and grammar symbol as arguments and produce states

MODEL OF SLR PARSER

Construction Of SLR Parsing Table

Central Idea

To construct a DFA from the given grammar to
recognize viable prefixes

Construction of LR(0) Items

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

Example:

If $A \rightarrow XYZ$ then there are four possible LR(0) items as:

$A \rightarrow .XYZ$

$A \rightarrow X .YZ$

$A \rightarrow XY .Z$

$A \rightarrow XYZ .$

Augmented Grammar

Given: If a grammar G is with start symbol S

then G' is augmented Grammar for G

Two elements are added in G to get G'

1. New start symbol G'
2. New production $S' \rightarrow S$

Acceptance of string is announced only when parser is about to reduce $S' \rightarrow S$

Closure Operation

If I is a set of items for a grammar G then

The closure (I) is the set of items constructed from I by the two rules as:

1. Initially every item in I is added in closure (I)
2. If $A \rightarrow a . B \beta$ is in closure (I) and $B \rightarrow r$ then add the item $B \rightarrow . r$ to closure(I) if it is not already there.

Apply the rule until no more rules can be added

Closure Operation

Example:

Given Grammar as

$$E' \rightarrow E$$

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

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

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

If I is the set of one item { [$E' \rightarrow . E$] } then closure of I contains

$$E' \rightarrow . E$$

$$E \rightarrow . E + T$$

$$E \rightarrow . T$$

$$T \rightarrow . T * F$$

$$T \rightarrow . F$$

$$F \rightarrow . (E)$$

$$F \rightarrow . id$$

GOTO Operation

If GOTO (I, X) where I is the set of items and X is a grammar symbol.

If I contains [A → α . X β] then

GOTO (I, X) - Closure of the set of all items [A → α X . β]

Example:

If I is set of two items as { [E' → E .] , [E → E . + T] }

Then GOTO (I, +) consists of

E → E + . T

T → . T * F

T → . F

F → . (E)

F → . id

Construction of LR (0) Automaton

Grammar

State 1: I0

Consider Production with start symbol

$E' \rightarrow . E$

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

Add to I0

$T \rightarrow T * F \mid F$

$E \rightarrow . E + T$

$F \rightarrow (E) \mid id$

$E \rightarrow . T$

$T \rightarrow . T * F$

$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . id$

Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

State I0:

$E' \rightarrow . E$

$E \rightarrow . E + T$

$E \rightarrow . T$

$T \rightarrow . T * F$

$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . id$

GOTO (I0 , E) :

$E' \rightarrow E .$

$E \rightarrow E . + T$ [New State: I1]

GOTO (I0 , T) :

$E \rightarrow T .$

$T \rightarrow T . * F$ [New State: I2]

Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

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

$F \rightarrow (E) \mid id$

State I0:

$E' \rightarrow . E$

$E \rightarrow . E + T$

$E \rightarrow . T$

$T \rightarrow . T^* F$

$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . id$

GOTO (I0 , F) :

$T \rightarrow F . \quad [\text{New State: I3}]$

GOTO (I0 , ()) :

$F \rightarrow (. E)$

$E \rightarrow . E + T$

$E \rightarrow . T$

$T \rightarrow . T^* F$

$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . id$

[New State: I4]

Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

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

$F \rightarrow (E) \mid id$

State I0:

$E' \rightarrow . E$

$E \rightarrow . E + T$

$E \rightarrow . T$

$T \rightarrow . T^* F$

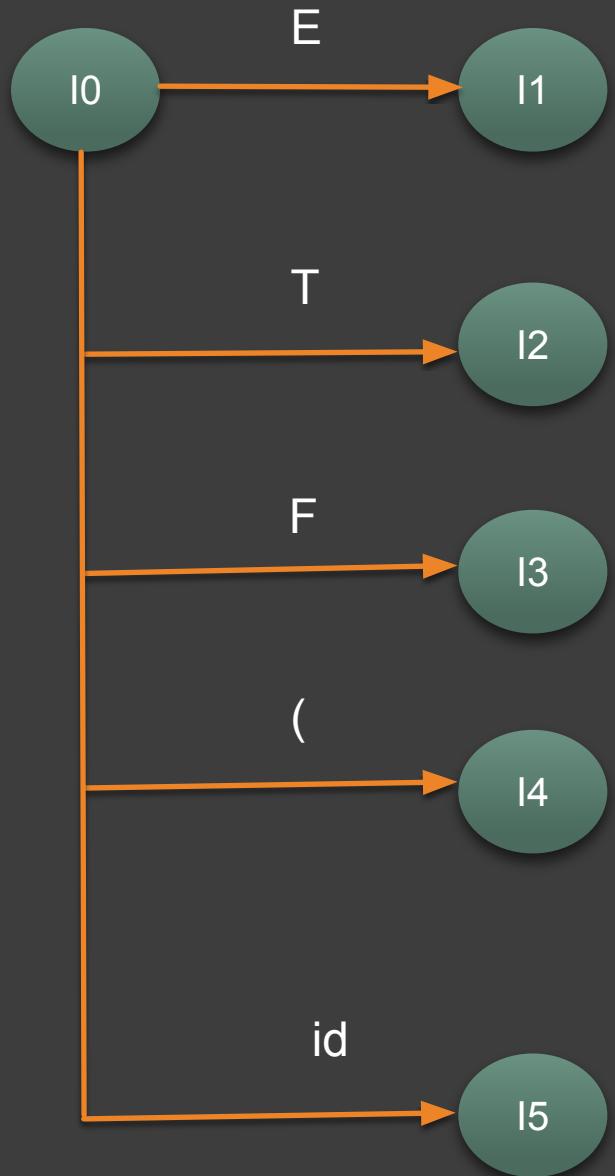
$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . id$

GOTO (I0 , id) :

$F \rightarrow id . \quad [New\ State: I5]$



Grammar

$$E' \rightarrow E$$

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

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

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

GOTO (I1 , +) :

$$E \rightarrow E + . T$$

$$T \rightarrow . T^* F$$

$$T \rightarrow . F$$

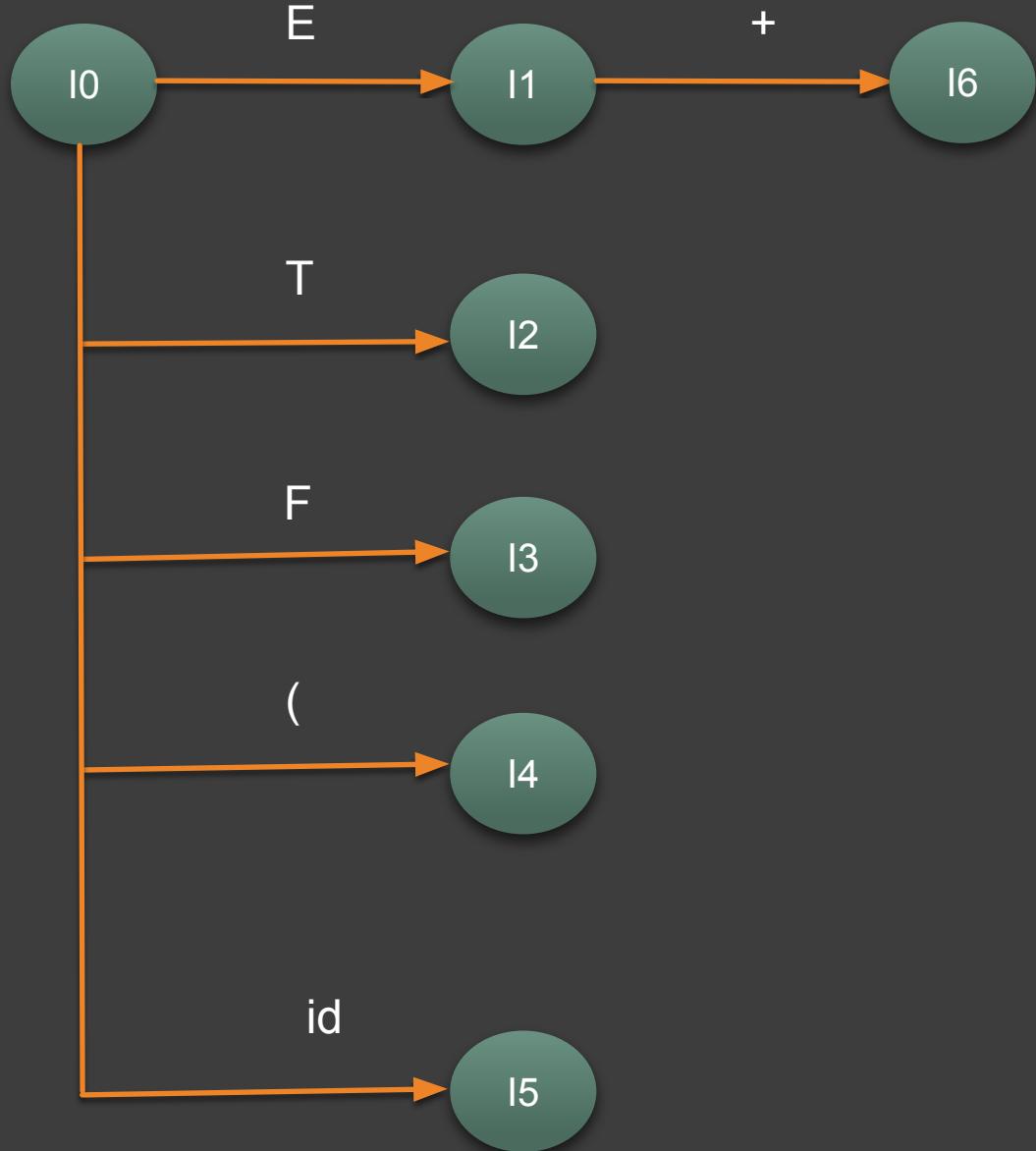
$$F \rightarrow . (E)$$

F → . id [New State: I6]

State I1:

$$E' \rightarrow E .$$

$$E \rightarrow E . + T$$



Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

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

$F \rightarrow (E) \mid id$

GOTO (I2 , *) :

$T \rightarrow T^* . F$

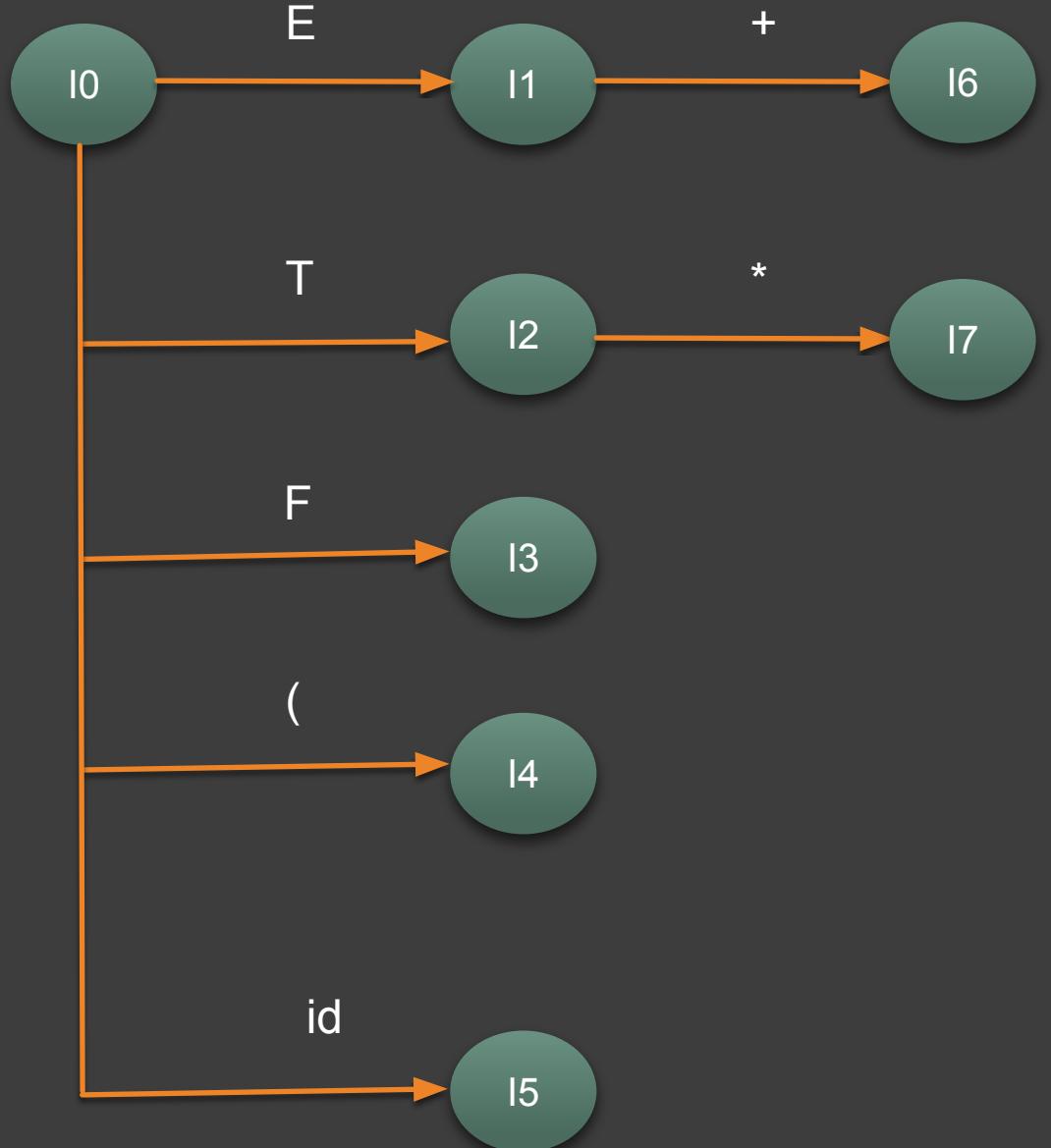
$F \rightarrow . (E)$

$F \rightarrow . id$ [New State: I7]

State I2:

$E \rightarrow T .$

$T \rightarrow T . ^* F$



Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

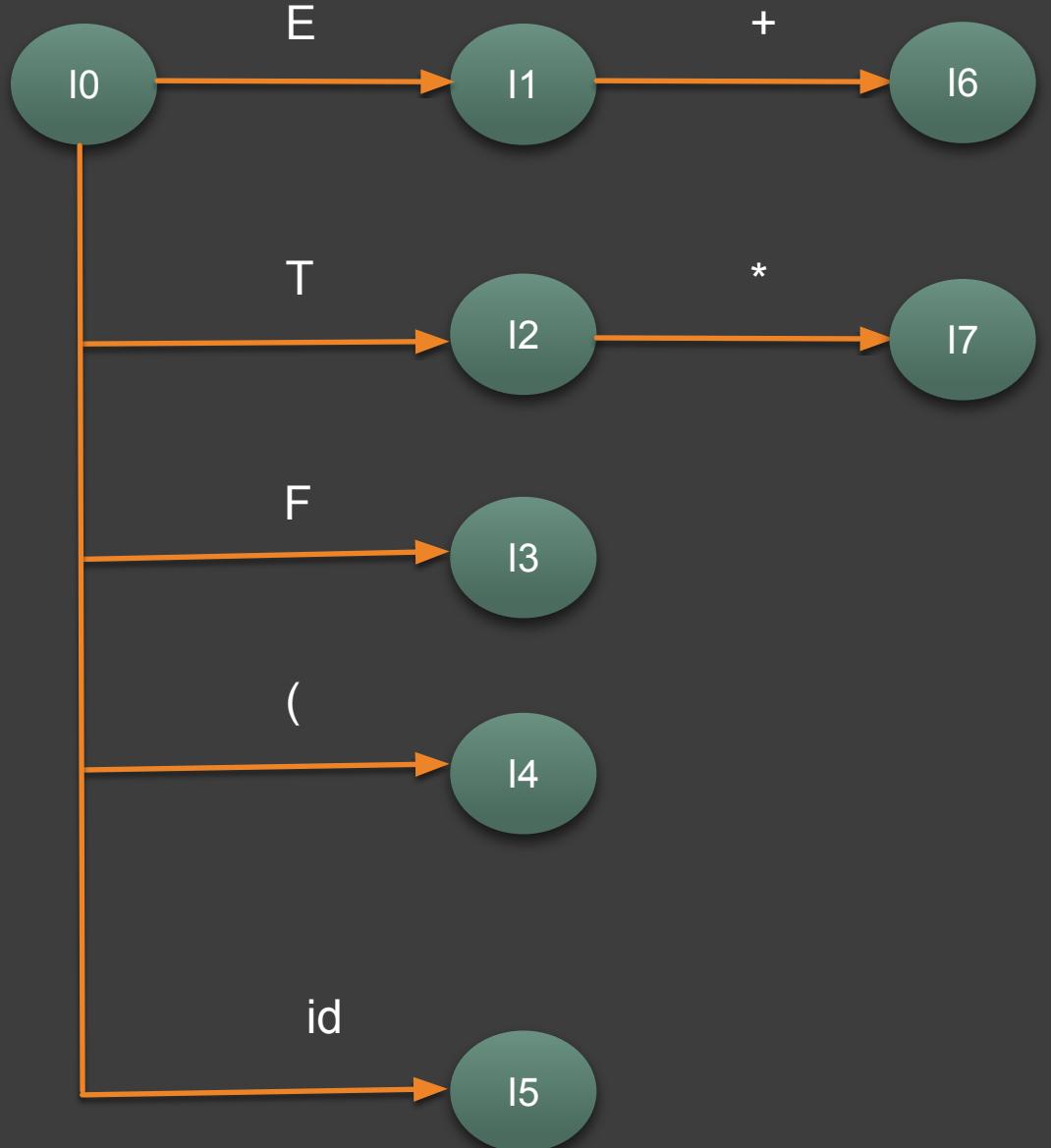
$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

No possible GOTO operation

State I3:

$T \rightarrow F .$



Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

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

$F \rightarrow (E) \mid id$

State I4:

$F \rightarrow (. E)$

$E \rightarrow . E + T$

$E \rightarrow . T$

$T \rightarrow . T^* F$

$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . id$

GOTO (I4 , E) :

$F \rightarrow (E .)$

$E \rightarrow E . + T$ [New State: I8]

GOTO (I4 , T) :

$E \rightarrow T .$

$T \rightarrow T . ^* F$ [Existing State: I2]

GOTO (I4 , F) :

$T \rightarrow F .$ [Existing State: I3]

State I2:

$E \rightarrow T .$

$T \rightarrow T . ^* F$

State I3:

$T \rightarrow F .$

Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

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

$F \rightarrow (E) \mid id$

State I4:

$F \rightarrow (. E)$

$E \rightarrow . E + T$

$E \rightarrow . T$

$T \rightarrow . T^* F$

$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . id$

GOTO (I4 , () :

$F \rightarrow (. E)$

$E \rightarrow . E + T$

$E \rightarrow . T$

$T \rightarrow . T^* F$

$T \rightarrow . F$

$F \rightarrow . (E)$

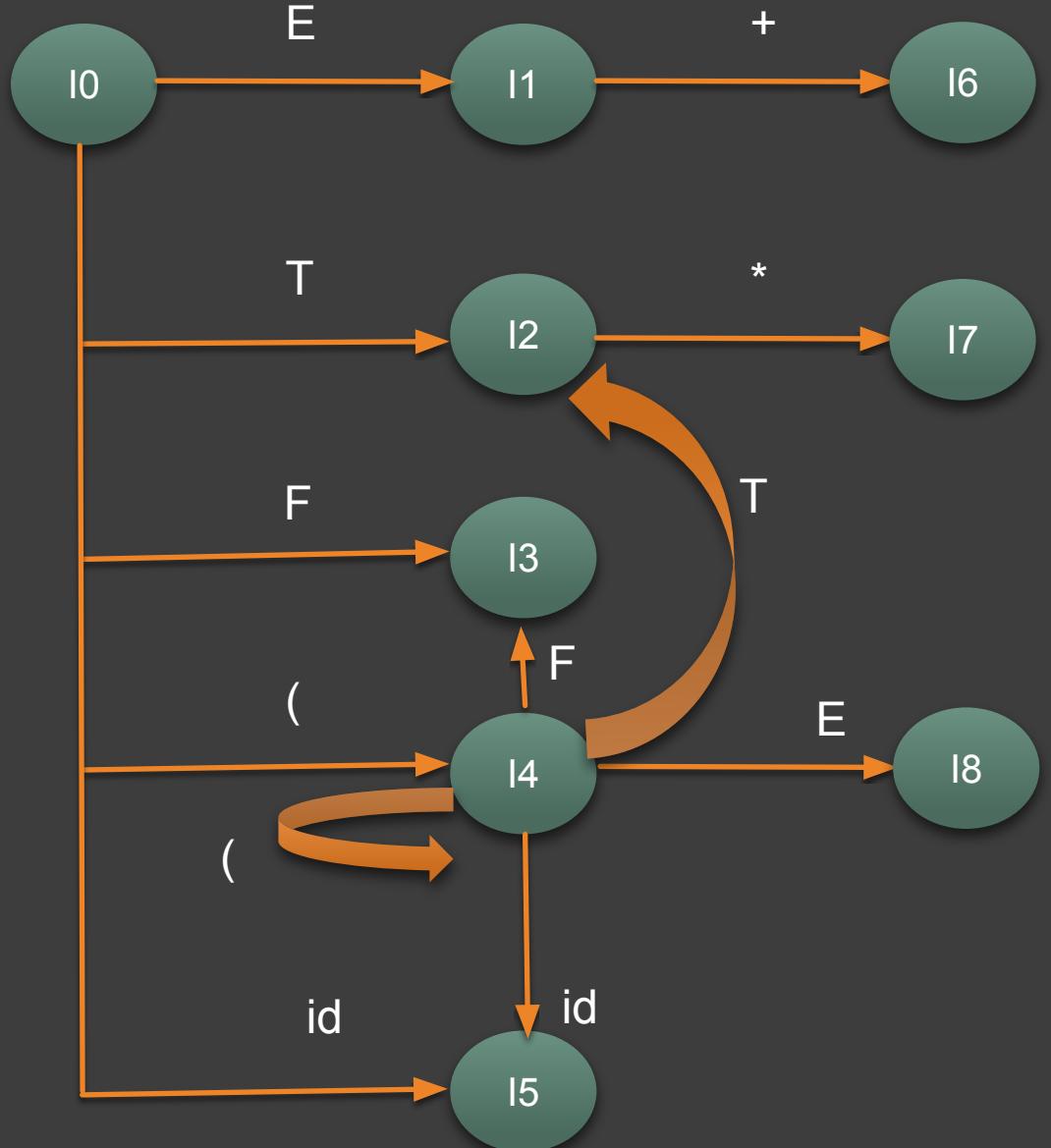
$F \rightarrow . Id$ [Existing State: I4]

GOTO (I4 , id) :

$F \rightarrow id .$ [Existing State: I5]

State I5:

$F \rightarrow id .$



Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

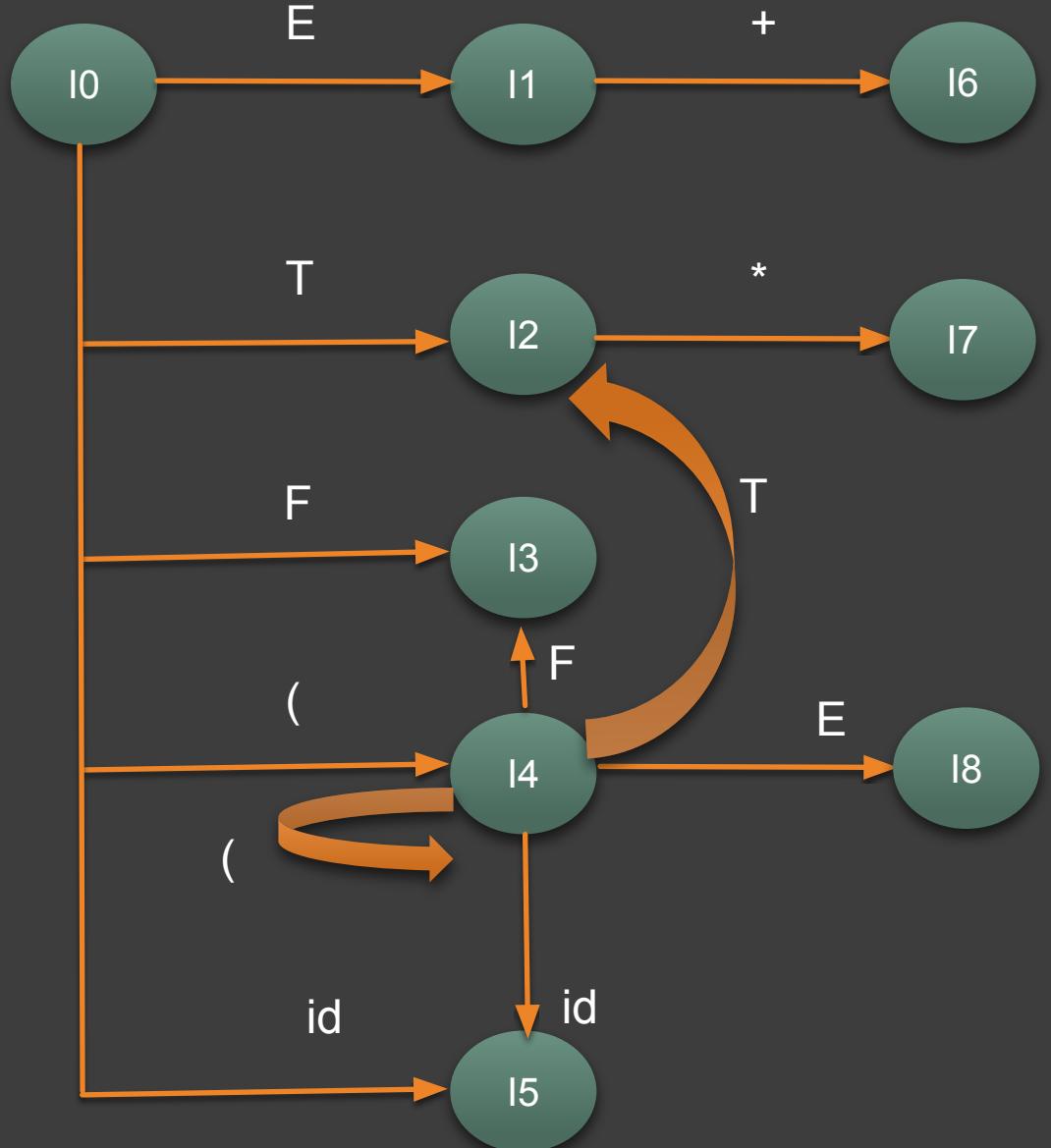
$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

No possible GOTO operation

State I5:

$F \rightarrow id .$



Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

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

$F \rightarrow (E) \mid id$

State I6:

$E \rightarrow E + . T$

$T \rightarrow . T^* F$

$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . id$

GOTO (I6 , T) :

$E \rightarrow E + T .$

$T \rightarrow T . * F \quad [New\ State: I9]$

GOTO (I6 , F) :

$T \rightarrow F . \quad [Existing\ State: I3]$

State I3:

$T \rightarrow F .$

Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

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

$F \rightarrow (E) \mid id$

State I6:

$E \rightarrow E + . T$

$T \rightarrow . T^* F$

$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . id$

GOTO (I6 , ()) :

$F \rightarrow (. E)$

$E \rightarrow . E + T$

$E \rightarrow . T$

$T \rightarrow . T^* F$

$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . Id$ [Existing State: I4]

State I4:

$F \rightarrow (. E)$

$E \rightarrow . E + T$

$E \rightarrow . T$

$T \rightarrow . T^* F$

$T \rightarrow . F$

$F \rightarrow . (E)$

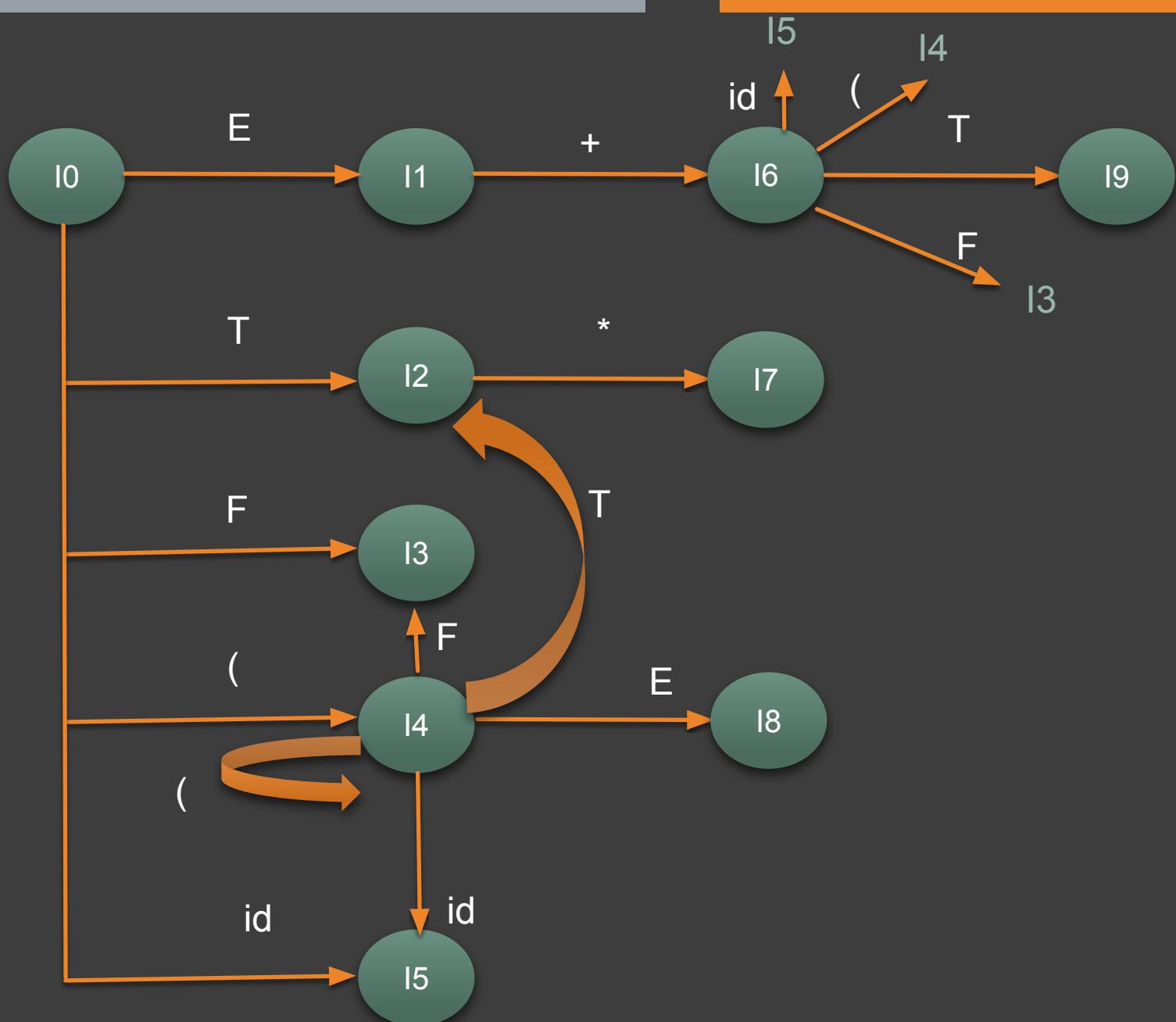
$F \rightarrow . id$

GOTO (I6 , id) :

$F \rightarrow id .$ [Existing State: I5]

State I5:

$F \rightarrow id .$



Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

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

$F \rightarrow (E) \mid id$

State I7:

$T \rightarrow T^* . F$

$F \rightarrow . (E)$

$F \rightarrow . id$

GOTO (I7 , F) :

$T \rightarrow T^* F .$ [New State: I10]

GOTO (I7 , ()) :

$F \rightarrow (. E)$

$E \rightarrow . E + T$

$E \rightarrow . T$

$T \rightarrow . T^* F$

$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . Id$ [Existing State: I4]

GOTO (I7 , id) :

$F \rightarrow id .$ [Existing State: I5]

State I4:

$F \rightarrow (. E)$

$E \rightarrow . E + T$

$E \rightarrow . T$

$T \rightarrow . T^* F$

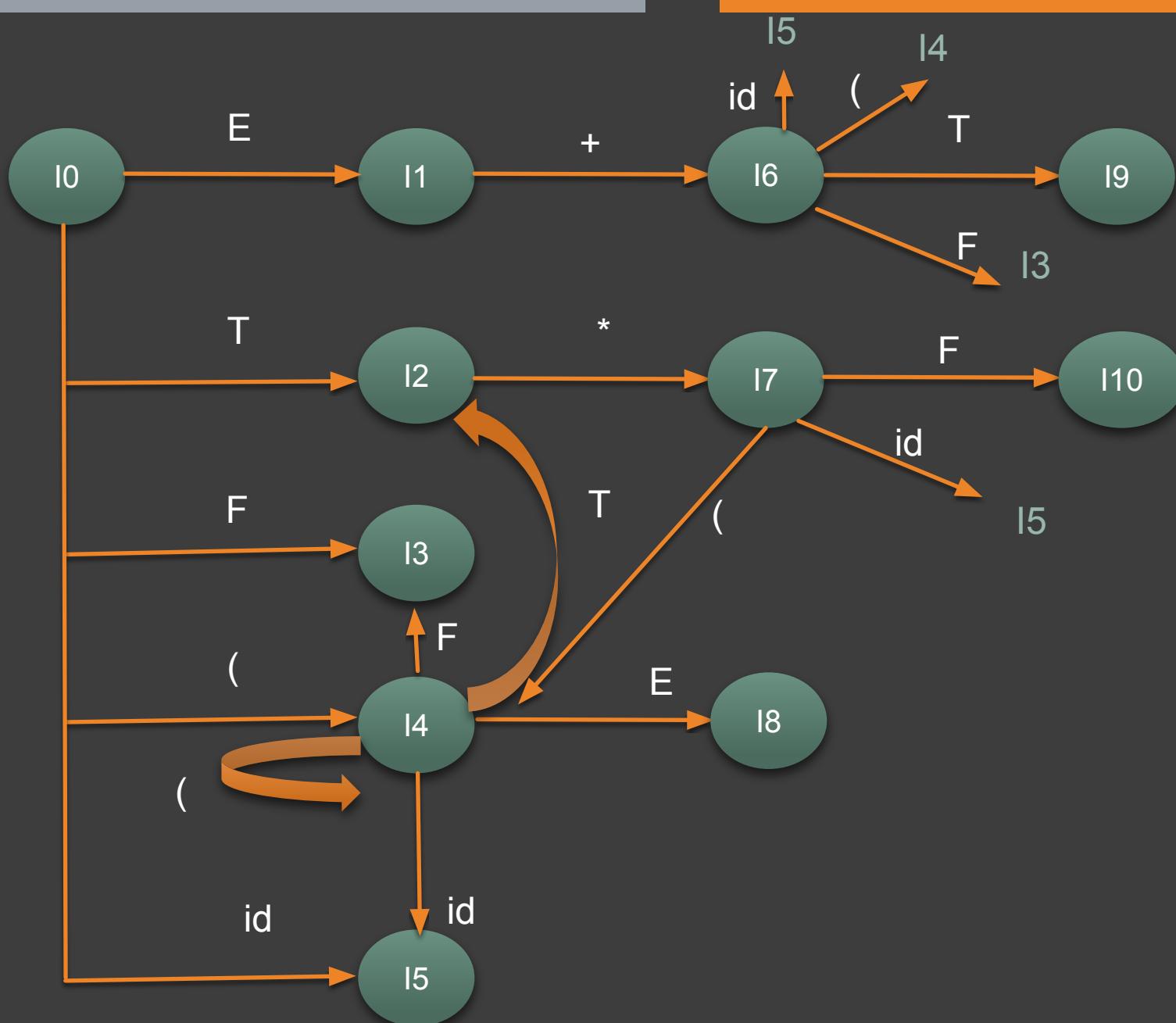
$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . id$

State I5:

$F \rightarrow id .$



Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

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

$F \rightarrow (E) \mid id$

State I8:

$F \rightarrow (E .)$

$E \rightarrow E . + T$

GOTO (I8 ,) :

$F \rightarrow (E) .$ [New State: I11]

GOTO (I8 , +) :

$E \rightarrow E + . T$

$T \rightarrow . T^* F$

$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . id$

[Existing State: I6]

State I6:

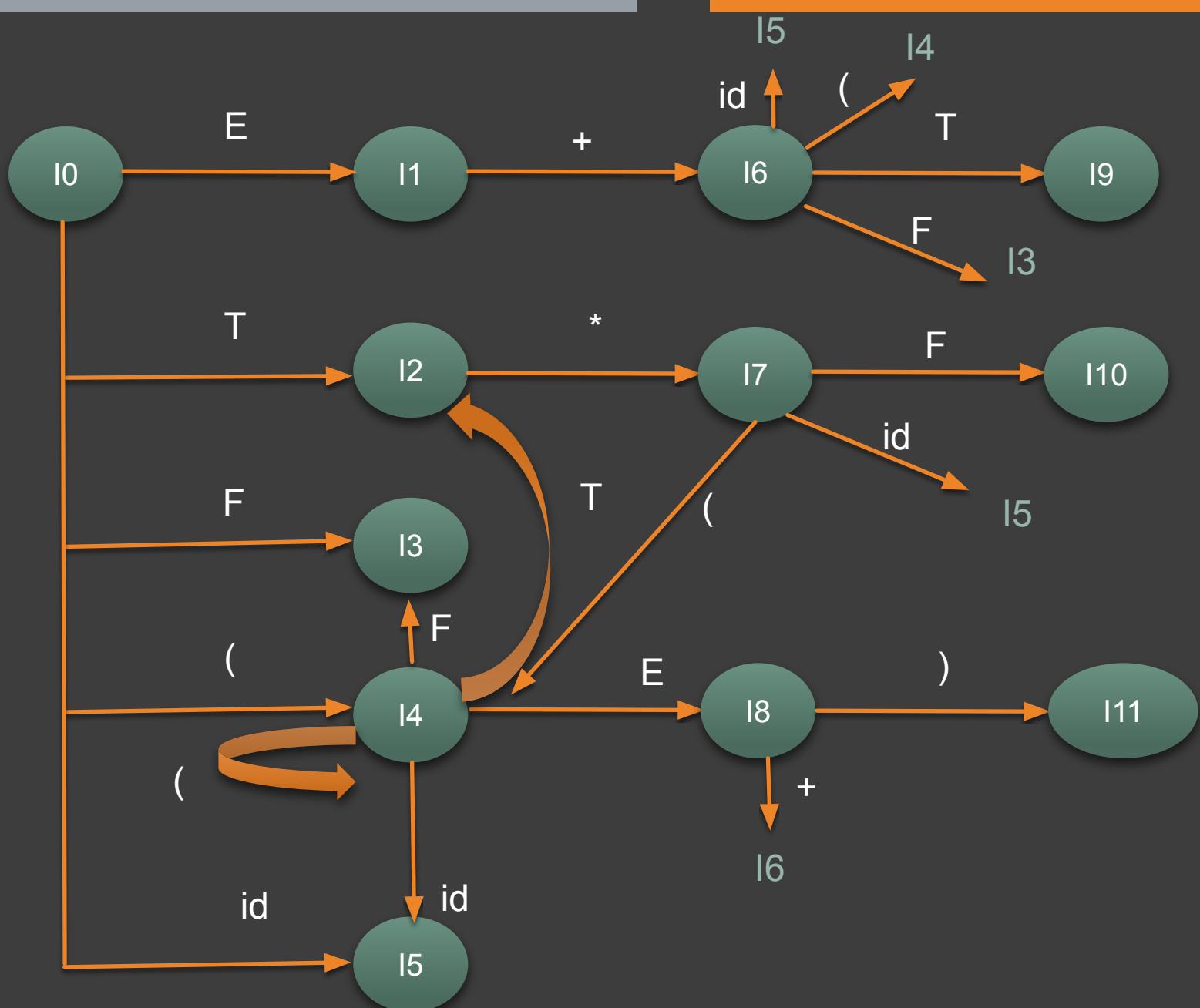
$E \rightarrow E + . T$

$T \rightarrow . T^* F$

$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . id$



Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

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

$F \rightarrow (E) \mid id$

State I9:

$F \rightarrow E + T .$

$T \rightarrow T . ^* F$

GOTO (I9 , *) :

$T \rightarrow T^* . F$

$F \rightarrow . (E)$

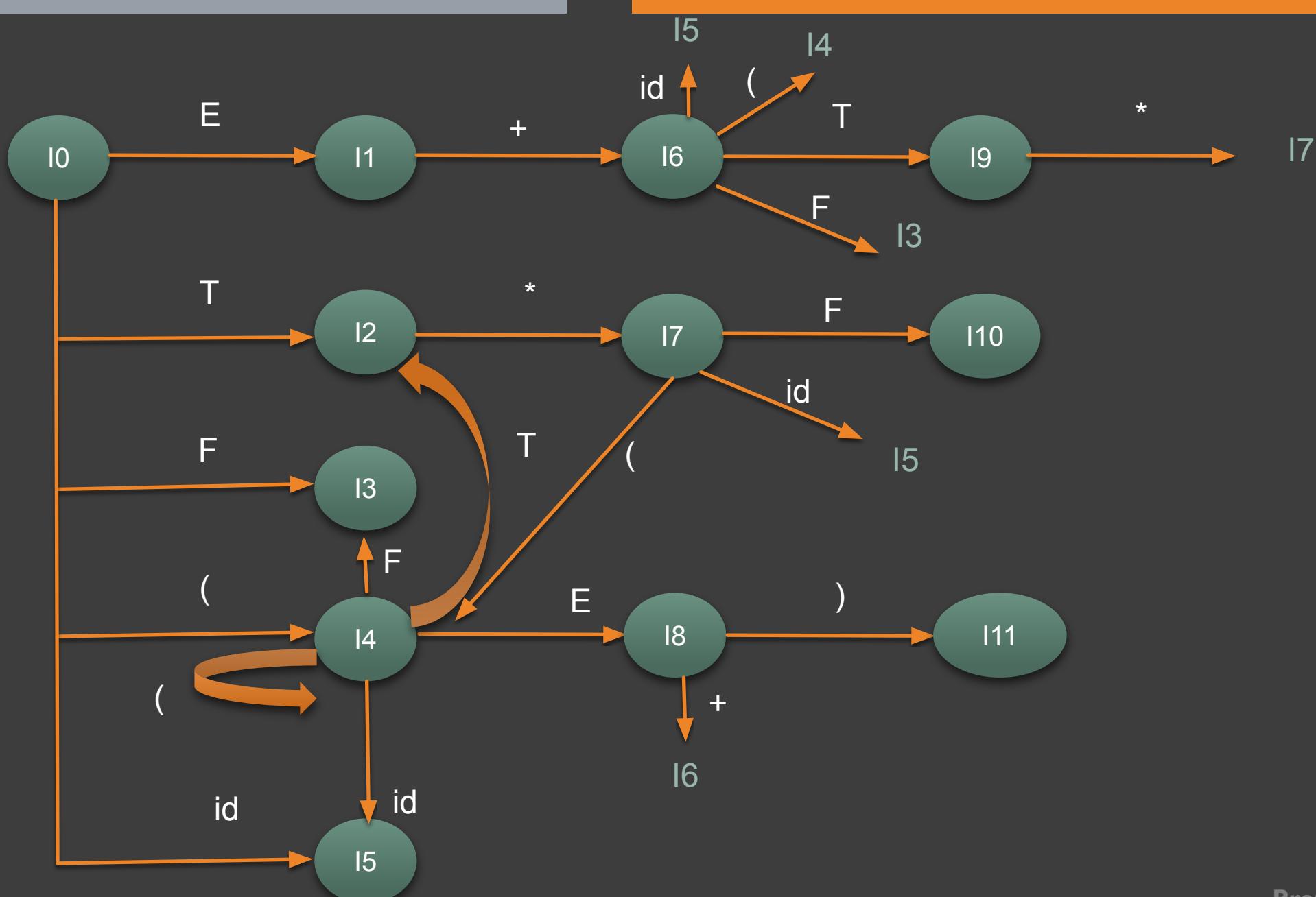
$F \rightarrow . id$ [Existing State: I7]

State I7:

$T \rightarrow T^* . F$

$F \rightarrow . (E)$

$F \rightarrow . id$



Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

No possible GOTO operation

State I10:

$T \rightarrow T * F .$

Grammar

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

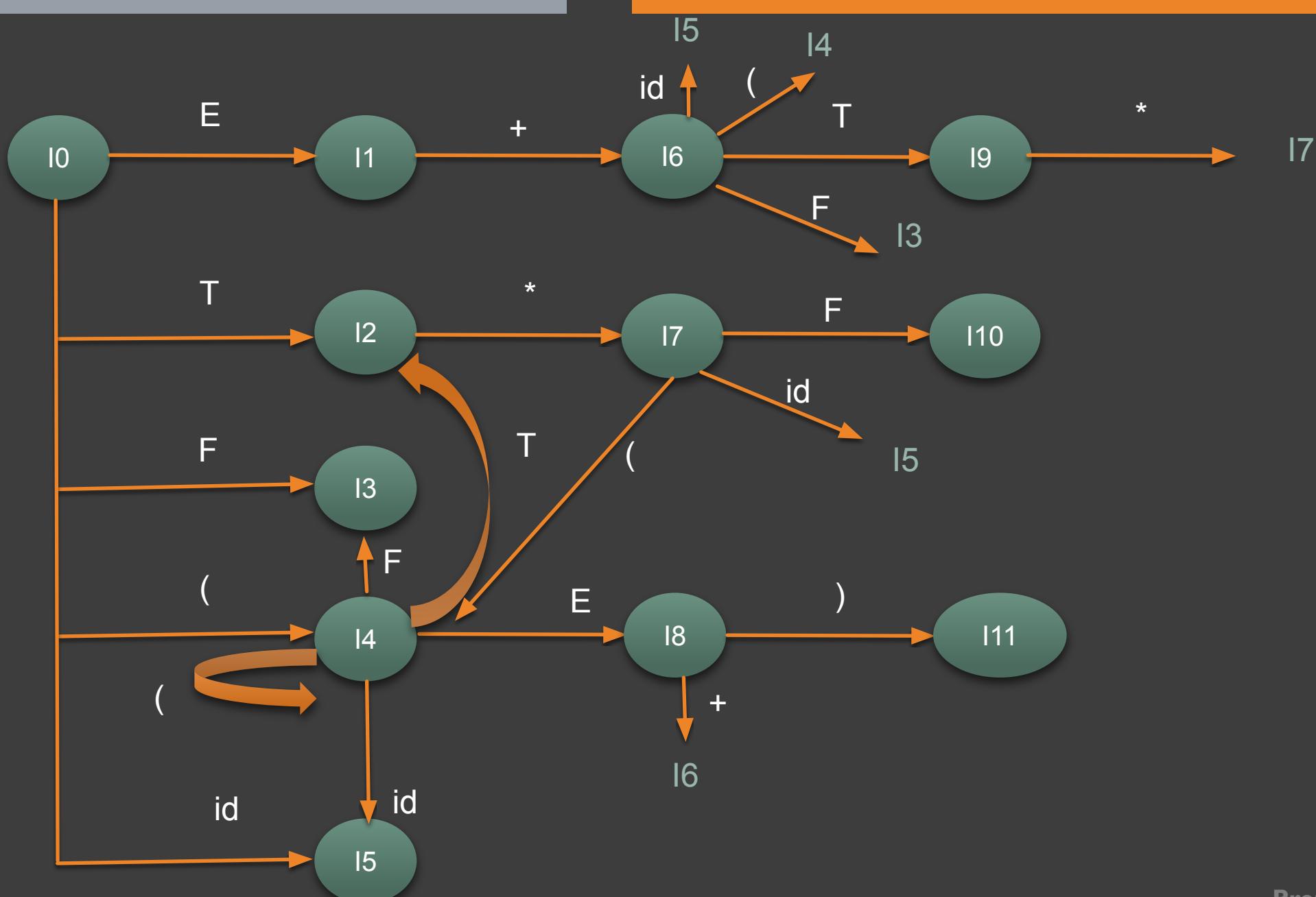
$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

No possible GOTO operation

State I11:

$F \rightarrow (E) .$



Construction of SLR Parsing Table

Input:

An augmented Grammar G'

Output:

The SLR Parsing Table with functions ACTION and GOTO for G'

Construction of SLR Parsing Table

Method:

1. Construct $C = \{ I_0, I_1, \dots, I_n \}$ the collection of sets of LR (0) items for G'
2. State i is constructed from I_i .

The parsing action for state i are determined as follows:

- a. If $[A \rightarrow \alpha . a \beta]$ is in I_i and $\text{GOTO} (I_i, a) = I_j$ then
Set Action $[i, a]$ to "**Shift j**"
Here a must be terminal
- b. If $[A \rightarrow \alpha .]$ then
Set Action $[i, a]$ to "**Reduce $A \rightarrow \alpha$** "
For all a in FOLLOW (A)
- c. If $[S' \rightarrow S .]$ is in I_i then
Set Action $[i, \$]$ to "**Accept**"

If any conflicting actions are generated by the above rules
Then grammar is not SLR (1)

Construction of SLR Parsing Table

Method:

3. The GOTO transitions for state i are constructed for all non-terminals A using the rule

If $\text{GOTO} [i , A] = l_j$ then
Set $\text{GOTO} [i , A] = j$

4. All entries not defined by rules (2) and (3) are made "ERROR"

5. The initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow . S]$

Construction of SLR Parsing Table

Given Grammar:

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

State	Action						GOTO		
	id	+	*	()	\$	E	T	F
0									
1									
2									
3									
4									
5									
6									
7									
8									
9									
10									
11									

Construction of SLR Parsing Table

Given Grammar:

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

State I0:

$$E' \rightarrow . E$$

$$E \rightarrow . E + T$$

$$E \rightarrow . T$$

$$T \rightarrow . T * F$$

$$T \rightarrow . F$$

$$F \rightarrow . (E)$$

$$F \rightarrow . id$$

(l4
id l5

State	Action						GOTO		
	id	+	*	()	\$	E	T	F
0	s5				s4				
1									
2									
3									
4									
5									
6									
7									
8									
9									
10									
11									

Construction of SLR Parsing Table

Given Grammar:

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

State I1:

$$E' \rightarrow E .$$

$$E \rightarrow E . + T$$

+
l6

State	Action						GOTO		
	id	+	*	()	\$	E	T	F
0	s5				s4				
1		s6					Accept		
2									
3									
4									
5									
6									
7									
8									
9									
10									
11									

Construction of SLR Parsing Table

Given Grammar:

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

State I2:

$$E \rightarrow T .$$

$$T \rightarrow T . * F$$

*
|7

State	Action						GOTO		
	id	+	*	()	\$	E	T	F
0	s5				s4				
1			s6				Accept		
2		r2	s7			r2	r2		
3									
4									
5									
6									
7									
8									
9									
10									
11									

Given Grammar:

$$\text{FOLLOW}(E) = \{ +,), \$ \}$$

$$\text{FOLLOW}(T) = \{ *, +,), \$ \}$$

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

Construction of SLR Parsing Table

Given Grammar:

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

State I3:

T → F .

Given Grammar:

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

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

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

State	Action						GOTO		
	id	+	*	()	\$	E	T	F
0	s5				s4				
1		s6					Accept		
2		r2	s7			r2	r2		
3		r4	r4			r4	r4		
4									
5									
6									
7									
8									
9									
10									
11									

Construction of SLR Parsing Table

Given Grammar:

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

State I4:

$$F \rightarrow (. E)$$

$$E \rightarrow . E + T$$

$$E \rightarrow . T$$

$$T \rightarrow . T * F$$

$$T \rightarrow . F$$

$$F \rightarrow . (E)$$

$$F \rightarrow . id$$

(I4

id I5

State	Action						GOTO		
	id	+	*	()	\$	E	T	F
0	s5				s4				
1			s6				Accept		
2		r2	s7			r2	r2		
3		r4	r4			r4	r4		
4	s5				s4				
5									
6									
7									
8									
9									
10									
11									

Construction of SLR Parsing Table

Given Grammar:

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

State I5:

$F \rightarrow id .$

Given Grammar:

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

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

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

State	Action						GOTO		
	id	+	*	()	\$	E	T	F
0	s5				s4				
1		s6					Accept		
2		r2	s7			r2	r2		
3		r4	r4			r4	r4		
4	s5				s4				
5		r6	r6			r6	r6		
6									
7									
8									
9									
10									
11									

Construction of SLR Parsing Table

Given Grammar:

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

State I6:

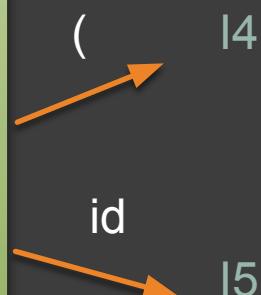
$$E \rightarrow E + . T$$

$$T \rightarrow . T * F$$

$$T \rightarrow . F$$

$$F \rightarrow . (E)$$

$$F \rightarrow . id$$



State	Action						GOTO		
	id	+	*	()	\$	E	T	F
0	s5				s4				
1			s6				Accept		
2		r2	s7			r2	r2		
3		r4	r4			r4	r4		
4	s5				s4				
5			r6	r6		r6	r6		
6	s5				s4				
7									
8									
9									
10									
11									

Construction of SLR Parsing Table

Given Grammar:

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

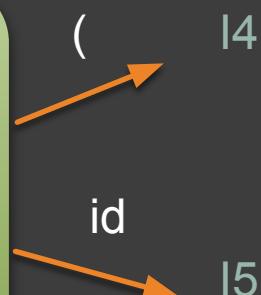
$$F \rightarrow id$$

State I7:

$$T \rightarrow T * . F$$

$$F \rightarrow . (E)$$

$$F \rightarrow . id$$



State	Action						GOTO		
	id	+	*	()	\$	E	T	F
0	s5				s4				
1			s6				Accept		
2		r2	s7			r2	r2		
3		r4	r4			r4	r4		
4	s5				s4				
5			r6	r6		r6	r6		
6	s5				s4				
7	s5				s4				
8									
9									
10									
11									

Construction of SLR Parsing Table

Given Grammar:

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

State I8:

$$F \rightarrow (E .)$$

$$E \rightarrow E . + T$$



State	Action						GOTO		
	id	+	*	()	\$	E	T	F
0	s5				s4				
1		s6					Accept		
2		r2	s7			r2	r2		
3		r4	r4			r4	r4		
4	s5				s4				
5		r6	r6			r6	r6		
6	s5				s4				
7	s5				s4				
8		s6				s11			
9									
10									
11									

Construction of SLR Parsing Table

Given Grammar:

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

State 19:
 $E \rightarrow E + T .$
 $T \rightarrow T . * F$

* | 7

State	Action						GOTO		
	id	+	*	()	\$	E	T	F
0	s5				s4				
1			s6				Accept		
2		r2	s7			r2	r2		
3		r4	r4			r4	r4		
4	s5				s4				
5			r6	r6		r6	r6		
6	s5				s4				
7	s5				s4				
8			s6				s11		
9			r1	s7		r1	r1		
10									
11									

Given Grammar:

$$\text{FOLLOW}(E) = \{ +,), \$ \}$$

$$\text{FOLLOW}(T) = \{ *, +,), \$ \}$$

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

Construction of SLR Parsing Table

Given Grammar:

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

State I10:

$$T \rightarrow T * F .$$

Given Grammar:

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

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

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

State	Action						GOTO		
	id	+	*	()	\$	E	T	F
0	s5				s4				
1		s6					Accept		
2		r2	s7			r2	r2		
3		r4	r4			r4	r4		
4	s5				s4				
5		r6	r6			r6	r6		
6	s5				s4				
7	s5				s4				
8		s6				s11			
9		r1	s7			r1	r1		
10		r3	r3			r3	r3		
11									

Construction of SLR Parsing Table

Given Grammar:

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

State I11:

$$F \rightarrow (E) .$$

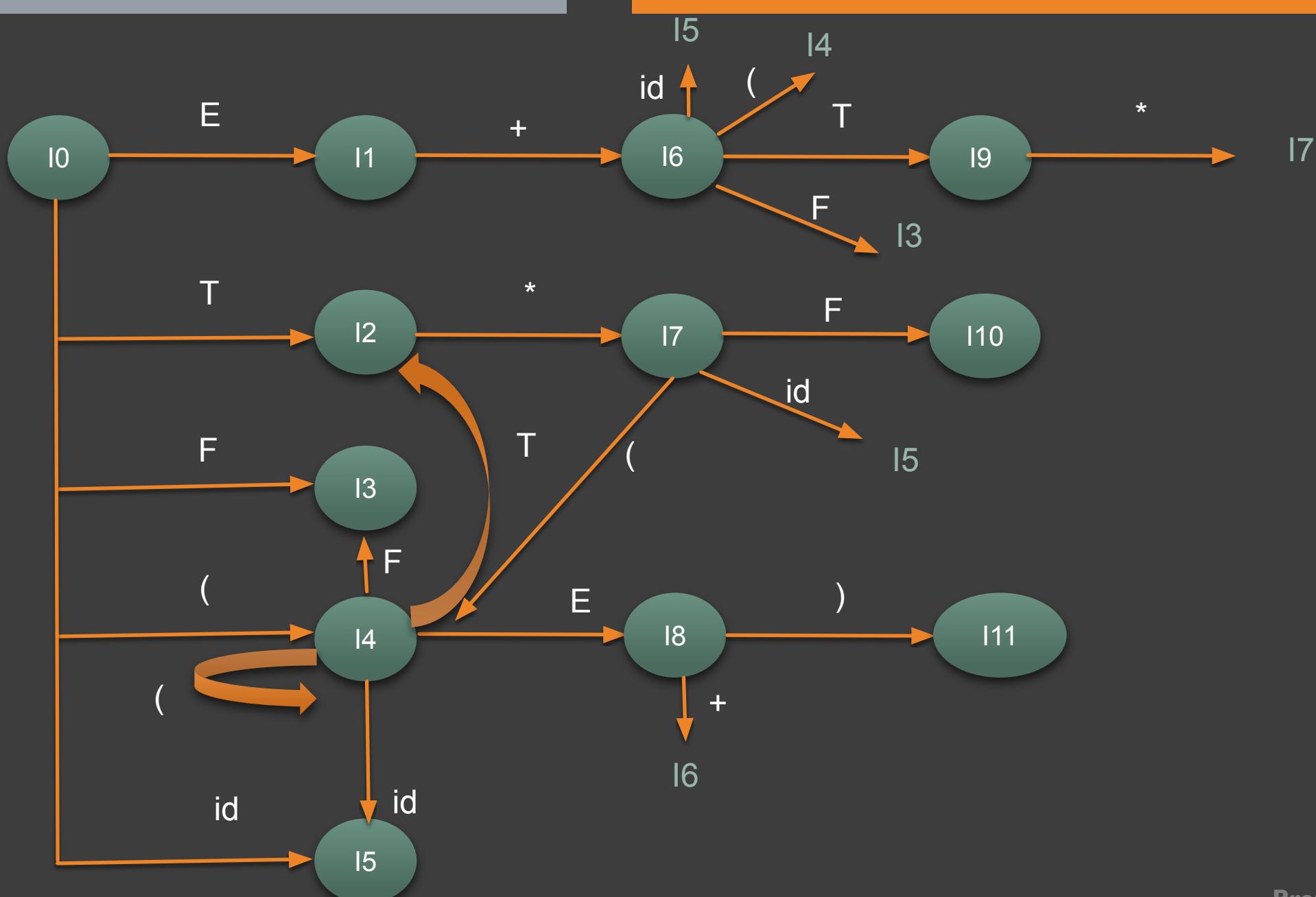
Given Grammar:

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

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

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

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



Construction of SLR Parsing Table

Given Grammar:

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

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				
5		r6	r6			r6	r6		
6	s5				s4				
7	s5				s4				
8		s6				s11			
9		r1	s7			r1	r1		
10		r3	r3			r3	r3		
11		r5	r5			r5	r5		

Construction of SLR Parsing Table

Given Grammar:

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

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					
7	s5			s4					
8		s6				s11			
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Construction of SLR Parsing Table

Given Grammar:

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

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
7	s5			s4					
8		s6				s11			
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Construction of SLR Parsing Table

Given Grammar:

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

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
7	s5			s4					10
8		s6				s11			
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Construction of SLR Parsing Table

Given Grammar:

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

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
7	s5			s4					10
8		s6				s11			
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

SLR Parsing Algorithm

Input:

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

Output:

If w is in $L(G)$, the reduction steps of bottom-up parse for w ; otherwise, there is an error

Method:

Initially, the parser has s_0 on the stack where s_0 is the initial state and $w\$$ is in the input buffer.

Execute the following Algorithm

SLR Parsing Algorithm

```
let a be the first symbol of w$;
while(1)                                /* repeat forever */
{
    let s be the state on top of the stack;
    if ( ACTION [ s , a ] = shift t )
    {
        push t onto the stack;
        let a be the next input symbol;
    }
    else if ( ACTION [ s , a ] = reduce A → β )
    {
        pop | β | symbols of the stack;
        let state t now be on top of the stack;
        push GOTO [ t , A] onto the stack;
        output the production A → β ;
    }
    else if ( ACTION [ s , a ] = accept )
        break; /* parsing is done */
    else call error-recovery routine;
}
```

SLR Parsing Algorithm

Stack	Symbols	Input	Action	Output
0		id * id + id \$	Shift	

SLR Parsing Algorithm

Stack	Symbols	Input	Action	Output
0		id * id + id \$	Shift	
0 5	id	* id + id \$	Reduce by F → id Pop '5' and check GOTO (0,F)	F → id

SLR Parsing Algorithm

Stack	Symbols	Input	Action	Output
0		id * id + id \$	Shift	
0 5	id	* id + id \$	Reduce by F → id Pop '5' and check GOTO (0,F)	F → id
0 3	F	* id + id \$	Reduce by T → F Pop '3' and check GOTO (0,T)	T → F

SLR Parsing Algorithm

Stack	Symbols	Input	Action	Output
0		id * id + id \$	Shift	
0 5	id	* id + id \$	Reduce by F → id Pop '5' and check GOTO (0,F)	F → id
0 3	F	* id + id \$	Reduce by T → F Pop '3' and check GOTO (0,T)	T → F
0 2	T	* id + id \$	Shift	

SLR Parsing Algorithm

Stack	Symbols	Input	Action	Output
0		id * id + id \$	Shift	
0 5	id	* id + id \$	Reduce by F → id Pop '5' and check GOTO (0,F)	F → id
0 3	F	* id + id \$	Reduce by T → F Pop '3' and check GOTO (0,T)	T → F
0 2	T	* id + id \$	Shift	
0 2 7	T *	id + id \$	Shift	

SLR Parsing Algorithm

Stack	Symbols	Input	Action	Output
0		id * id + id \$	Shift	
0 5	id	* id + id \$	Reduce by F → id Pop '5' and check GOTO (0,F)	F → id
0 3	F	* id + id \$	Reduce by T → F Pop '3' and check GOTO (0,T)	T → F
0 2	T	* id + id \$	Shift	
0 2 7	T *	id + id \$	Shift	
0 2 7 5	T * id	+ id \$	Reduce by F → id Pop '5' and check GOTO (7,F)	F → id

SLR Parsing Algorithm

Stack	Symbols	Input	Action	Output
0		id * id + id \$	Shift	
0 5	id	* id + id \$	Reduce by F → id Pop '5' and check GOTO (0,F)	F → id
0 3	F	* id + id \$	Reduce by T → F Pop '3' and check GOTO (0,T)	T → F
0 2	T	* id + id \$	Shift	
0 2 7	T *	id + id \$	Shift	
0 2 7 5	T * id	+ id \$	Reduce by F → id Pop '5' and check GOTO (7,F)	F → id
0 2 7 10	T * F	+ id \$	Reduce by T → T * F Pop '10', '7', '2' and check GOTO (0,T)	T → T * F

SLR Parsing Algorithm

Stack	Symbols	Input	Action	Output
0		id * id + id \$	Shift	
0 5	id	* id + id \$	Reduce by F → id Pop '5' and check GOTO (0,F)	F → id
0 3	F	* id + id \$	Reduce by T → F Pop '3' and check GOTO (0,T)	T → F
0 2	T	* id + id \$	Shift	
0 2 7	T *	id + id \$	Shift	
0 2 7 5	T * id	+ id \$	Reduce by F → id Pop '5' and check GOTO (7,F)	F → id
0 2 7 10	T * F	+ id \$	Reduce by T → T * F Pop '10','7','2' and check GOTO (0,T)	T → T * F
0 2	T	+ id \$	Reduce by E → T Pop '2' and check GOTO (0,E)	E → T

SLR Parsing Algorithm

Stack	Symbols	Input	Action	Output
0		id * id + id \$	Shift	
0 5	id	* id + id \$	Reduce by F → id Pop '5' and check GOTO (0,F)	F → id
0 3	F	* id + id \$	Reduce by T → F Pop '3' and check GOTO (0,T)	T → F
0 2	T	* id + id \$	Shift	
0 2 7	T *	id + id \$	Shift	
0 2 7 5	T * id	+ id \$	Reduce by F → id Pop '5' and check GOTO (7,F)	F → id
0 2 7 10	T * F	+ id \$	Reduce by T → T * F Pop '10','7','2' and check GOTO (0,T)	T → T * F
0 2	T	+ id \$	Reduce by E → T Pop '2' and check GOTO (0,E)	E → T
0 1	E	+ id \$	Shift	

SLR Parsing Algorithm

Stack	Symbols	Input	Action	Output
0		id * id + id \$	Shift	
0 5	id	* id + id \$	Reduce by F → id Pop '5' and check GOTO (0,F)	F → id
0 3	F	* id + id \$	Reduce by T → F Pop '3' and check GOTO (0,T)	T → F
0 2	T	* id + id \$	Shift	
0 2 7	T *	id + id \$	Shift	
0 2 7 5	T * id	+ id \$	Reduce by F → id Pop '5' and check GOTO (7,F)	F → id
0 2 7 10	T * F	+ id \$	Reduce by T → T * F Pop '10','7','2' and check GOTO (0,T)	T → T * F
0 2	T	+ id \$	Reduce by E → T Pop '2' and check GOTO (0,E)	E → T
0 1	E	+ id \$	Shift	
0 1 6	E +	id \$	Shift	

SLR Parsing Algorithm

Stack	Symbols	Input	Action	Output
0		id * id + id \$	Shift	
0 5	id	* id + id \$	Reduce by F → id Pop '5' and check GOTO (0,F)	F → id
0 3	F	* id + id \$	Reduce by T → F Pop '3' and check GOTO (0,T)	T → F
0 2	T	* id + id \$	Shift	
0 2 7	T *	id + id \$	Shift	
0 2 7 5	T * id	+ id \$	Reduce by F → id Pop '5' and check GOTO (7,F)	F → id
0 2 7 10	T * F	+ id \$	Reduce by T → T * F Pop '10','7','2' and check GOTO (0,T)	T → T * F
0 2	T	+ id \$	Reduce by E → T Pop '2' and check GOTO (0,E)	E → T
0 1	E	+ id \$	Shift	
0 1 6	E +	id \$	Shift	
0 1 6 5	E + id	\$	Reduce by F → id Pop '5' and check GOTO (6,F)	F → id

SLR Parsing Algorithm

Stack	Symbols	Input	Action	Output
0		id * id + id \$	Shift	
0 5	id	* id + id \$	Reduce by F → id Pop '5' and check GOTO (0,F)	F → id
0 3	F	* id + id \$	Reduce by T → F Pop '3' and check GOTO (0,T)	T → F
0 2	T	* id + id \$	Shift	
0 2 7	T *	id + id \$	Shift	
0 2 7 5	T * id	+ id \$	Reduce by F → id Pop '5' and check GOTO (7,F)	F → id
0 2 7 10	T * F	+ id \$	Reduce by T → T * F Pop '10','7','2' and check GOTO (0,T)	T → T * F
0 2	T	+ id \$	Reduce by E → T Pop '2' and check GOTO (0,E)	E → T
0 1	E	+ id \$	Shift	
0 1 6	E +	id \$	Shift	
0 1 6 5	E + id	\$	Reduce by F → id Pop '5' and check GOTO (6,F)	F → id
0 1 6 3	E + F	\$	Reduce by T → F Pop '3' and check GOTO (6,T)	T → F

SLR Parsing Algorithm

Stack	Symbols	Input	Action	Output
0		id * id + id \$	Shift	
0 5	id	* id + id \$	Reduce by F → id Pop '5' and check GOTO (0,F)	F → id
0 3	F	* id + id \$	Reduce by T → F Pop '3' and check GOTO (0,T)	T → F
0 2	T	* id + id \$	Shift	
0 2 7	T *	id + id \$	Shift	
0 2 7 5	T * id	+ id \$	Reduce by F → id Pop '5' and check GOTO (7,F)	F → id
0 2 7 10	T * F	+ id \$	Reduce by T → T * F Pop '10','7','2' and check GOTO (0,T)	T → T * F
0 2	T	+ id \$	Reduce by E → T Pop '2' and check GOTO (0,E)	E → T
0 1	E	+ id \$	Shift	
0 1 6	E +	id \$	Shift	
0 1 6 5	E + id	\$	Reduce by F → id Pop '5' and check GOTO (6,F)	F → id
0 1 6 3	E + F	\$	Reduce by T → F Pop '3' and check GOTO (6,T)	T → F
0 1 6 9	E + T	\$	Reduce by E → E + T Pop '9', '6', '1' and check GOTO (0,E)	E → E + T

Stack	Symbols	Input	Action	Output
0		id * id + id \$	Shift	
0 5	id	* id + id \$	Reduce by F → id Pop '5' and check GOTO (0,F)	F → id
0 3	F	* id + id \$	Reduce by T → F Pop '3' and check GOTO (0,T)	T → F
0 2	T	* id + id \$	Shift	
0 2 7	T *	id + id \$	Shift	
0 2 7 5	T * id	+ id \$	Reduce by F → id Pop '5' and check GOTO (7,F)	F → id
0 2 7 10	T * F	+ id \$	Reduce by T → T * F Pop '10','7','2' and check GOTO (0,T)	T → T * F
0 2	T	+ id \$	Reduce by E → T Pop '2' and check GOTO (0,E)	E → T
0 1	E	+ id \$	Shift	
0 1 6	E +	id \$	Shift	
0 1 6 5	E + id	\$	Reduce by F → id Pop '5' and check GOTO (6,F)	F → id
0 1 6 3	E + F	\$	Reduce by T → F Pop '3' and check GOTO (6,T)	T → F
0 1 6 9	E + T	\$	Reduce by E → E + T Pop '9', '6', '1' and check GOTO (0,E)	E → E + T
0 1	E	\$	Accept	



**Thank
You**



Syntax Directed Translation

Content

- Syntax Directed Definition
- Synthesized Attributes
- Inherited Attributes
- Evaluation order of SDD's
- S attributed definition
- L attributed definition

Syntax Directed Definition

- A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules.
- Attributes are associated with grammar symbols and rules are associated with productions.
- If X is a symbol and a is one of its attributes, then we write $X.a$ to denote the value of a at a particular parse-tree node labeled X .
- If we implement the nodes of the parse tree by records or objects, then the attributes of X can be implemented by data fields in the records that represent the nodes for X .
- Attributes may be of any kind: numbers, types, table references, or strings, for instance.
- The strings may even be long sequences of code, say code in the intermediate language used by a compiler.

Synthesized Attributes

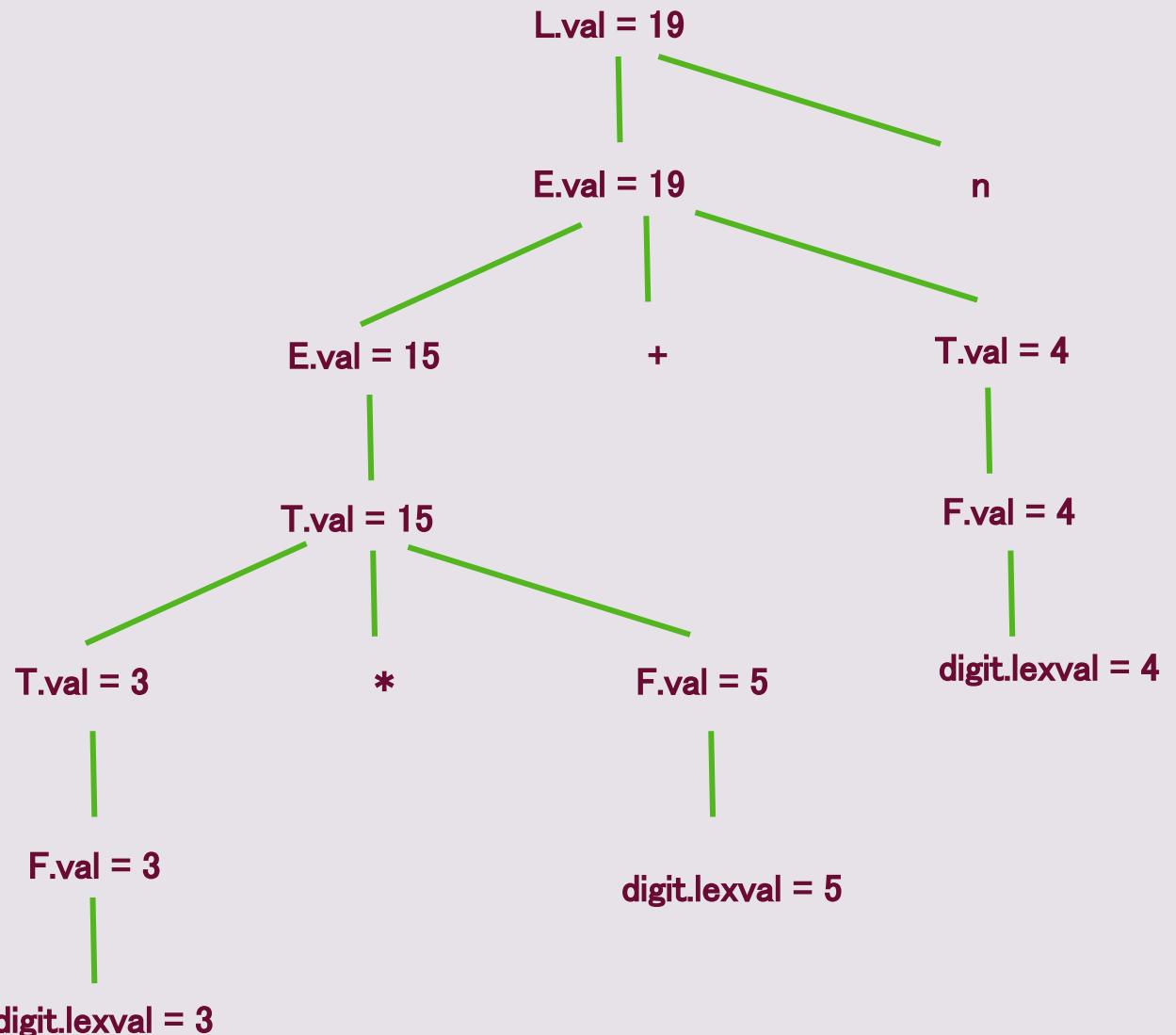
- A synthesized attribute for a nonterminal A at a parse-tree node N is defined by a semantic rule associated with the production at N.
- The production must have A as its head.
- A synthesized attribute at node N is defined only in terms of **attribute values at the children of N and at N itself.**
- Syntax directed definition that uses Synthesized attributes exclusively is said to be an S-attributed definition.
- A parse tree for an S-attributed definition can be annotated by evaluating semantic rules for attributes at each node
- To implement S-attributed definition LR parser can be used i.e. bottom-up parsing is used

Synthesized Attributes

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

Syntax Directed Definition of simple desk calculator

Synthesized Attributes



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

Prepared By: Vaibhav Ambhire

Inherited Attributes

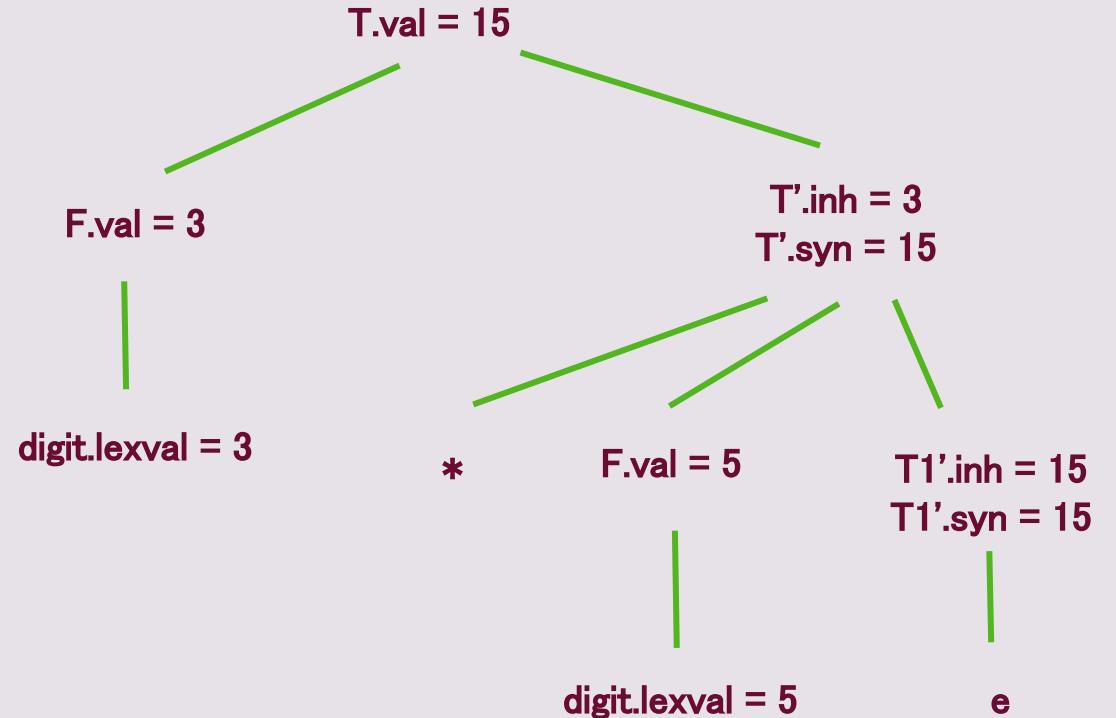
- An inherited attribute for a non - terminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N.
- The production must have B as a symbol in its body.
- An inherited attribute at node N is defined only in terms of **attribute values at N's parent, N itself and N's siblings.**
- Inherited attributes are convenient for expressing dependencies of a program language construct on the context in which appears

Inherited Attributes

Production	Semantic Rules
$T \rightarrow F T'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \rightarrow * F T_1'$	$T_1'.inh = T'.inh * F.val$ $T'.syn = T_1'.syn$
$T' \rightarrow e$	$T'.syn = T'.inh$
$F \rightarrow digit$	$F.val = digit.lexval$

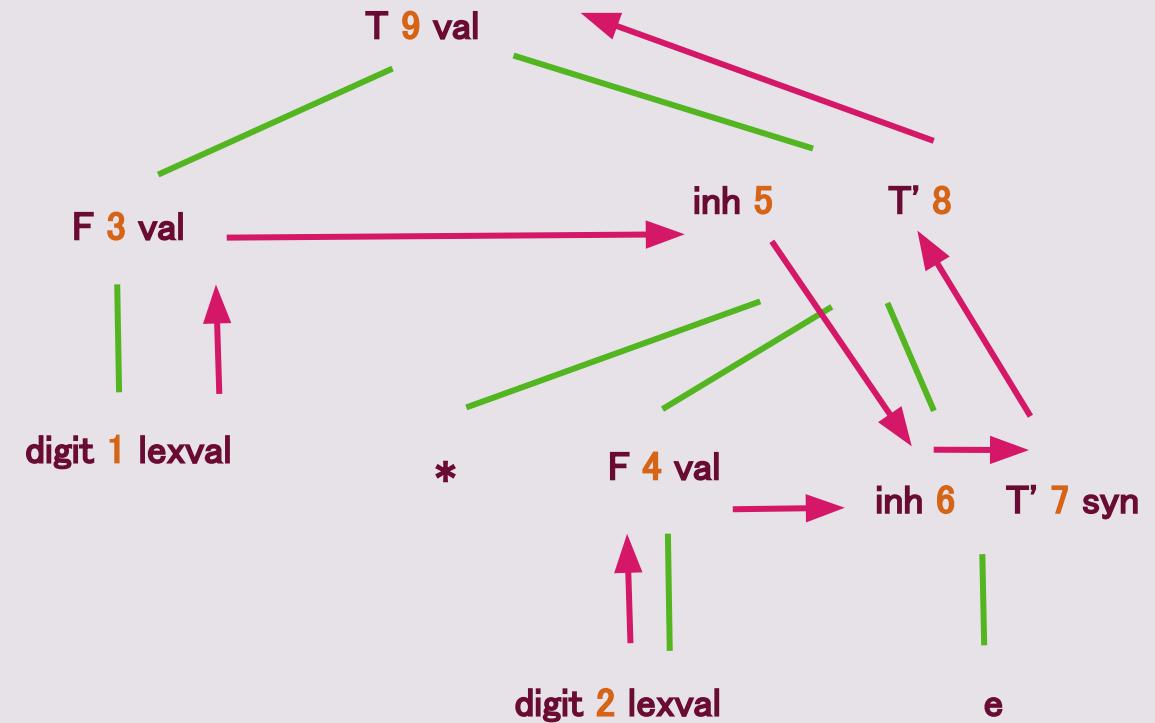
Inherited Attributes

Production	Semantic Rules
$T \rightarrow F T'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \rightarrow * F T1'$	$T1'.inh = T'.inh * F.val$ $T'.syn = T1'.syn$
$T' \rightarrow e$	$T'.syn = T'.inh$
$F \rightarrow digit$	$F.val = digit.lexval$



Annotated Parse Tree for $3 * 5$

Dependency Graph for Annotated Tree



Annotated Parse Tree for $3 * 5$

Prepared By: Vaibhav Ambhire

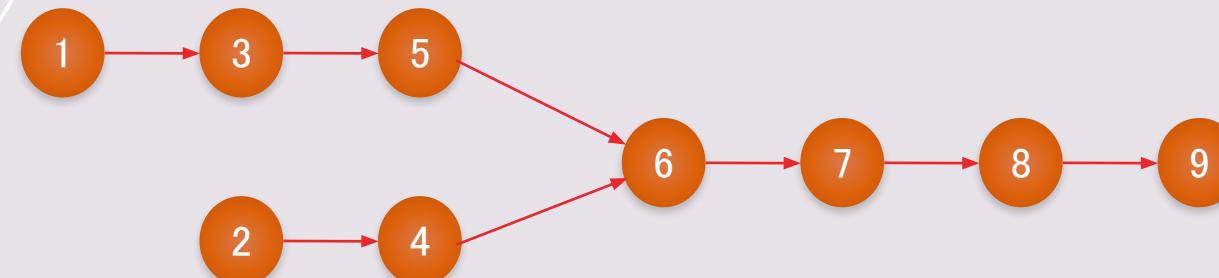
Topological Sort of Dependency Graph

- A dependency graph characterizes the possible order in which we can evaluate the attributes at various nodes of a parse tree.
- If there is an edge from node M to N, then attribute corresponding to M first be evaluated before evaluating N.
- The only allowable orders of evaluation are N_1, N_2, \dots, N_k such that if there is an edge from N_i to N_j then $i < j$.
- Such an ordering embeds a directed graph into a linear order, and is called a topological sort of the graph.
- If there is **any cycle** in the graph, then there are no topological sorts; that is, there is **no way to evaluate the SDD** on this parse tree.
- If there are no cycles, however, then there is always at least one topological sort.

Topological Sort of DependencyGraph

- Base sequenies are {1 3 5} and {2 4} with the suffx {6 7 8 9} being constant as the last nodes of the topological sorting need to remain fixed.

- 2 1 3 5 4 6 7 8 9,
- 2 1 3 4 5 6 7 8 9,
- 2 1 4 3 5 6 7 8 9,
- 2 4 1 3 5 6 7 8 9
- 1 3 5 2 4 6 7 8 9,
- 1 3 2 5 4 6 7 8 9,
- 1 2 3 5 4 6 7 8 9,
- 1 3 2 4 5 6 7 8 9,
- 1 2 3 4 5 6 7 8 9,
- 1 2 4 3 5 6 7 8 9,



S – Attributed

Definition

- Uses only synthesized attributes
- Semantic actions are placed at right end of the production
- Also called as postfix SDT
- Attributes are evaluated during bottom-up parsing

L- Attributed Definition

- Uses both inherited and synthesized attributes.
- Each inherited attribute is restricted to inherit either from parent or left sibling
- Semantic actions are placed anywhere on the RHS
- Attributes are evaluated by traversing parse tree depth first and left to right

THANK YOU