System Programming and Compiler Construction

CSC 602

Subject Incharge

Varsha Shrivastava

Assistant Professor

email: varshashrivastava@sfit.ac.in

Room No: 407

CSC 602 System Programming and Compiler Construction

Module 5

Compilers : Analysis Phase

Contents as per 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- Operator precedence parser, SLR
  - Semantic Analysis
  - Syntax directed definitions.

Introduction

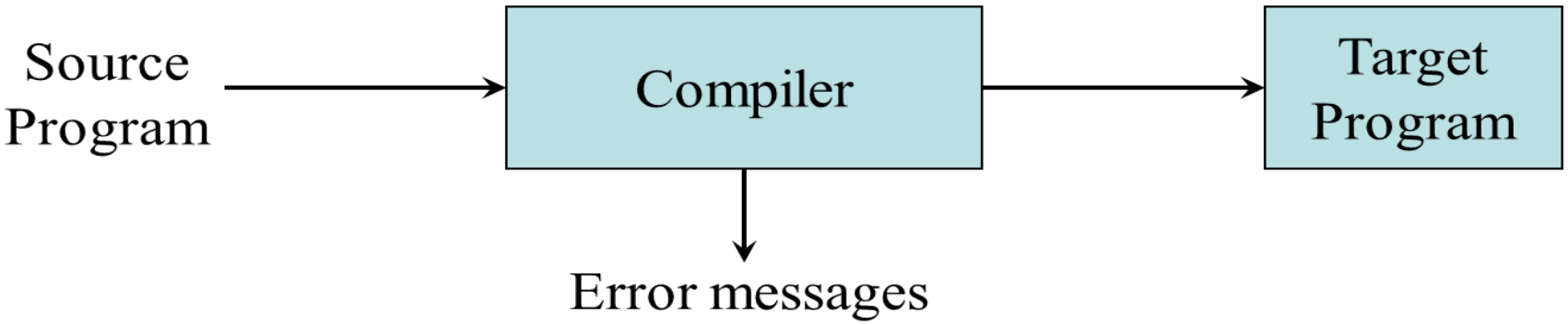
Translator and Compiler

- Translator
  - Program that takes input a program written in one programming language and produces output a program in another programming language.

- Compiler
  - If source language is a high level language (FORTRAN or COBOL) and output is a low level language (assembly or machine language), then such a translator is called a compiler.

Introduction

Compiler is a translator program that translates a program written in (HLL) the source program and translate it into an equivalent program in (MLL) the target program. As an important part of a compiler is error showing to the programmer.



Structure of Compiler

- Compiler
  - A compiler operates in ***phases.***
  - A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation.

  - There are two phases of compilation.

➢ Analysis (Machine Independent/Language Dependent)

➢ Synthesis(Machine Dependent/Language independent)

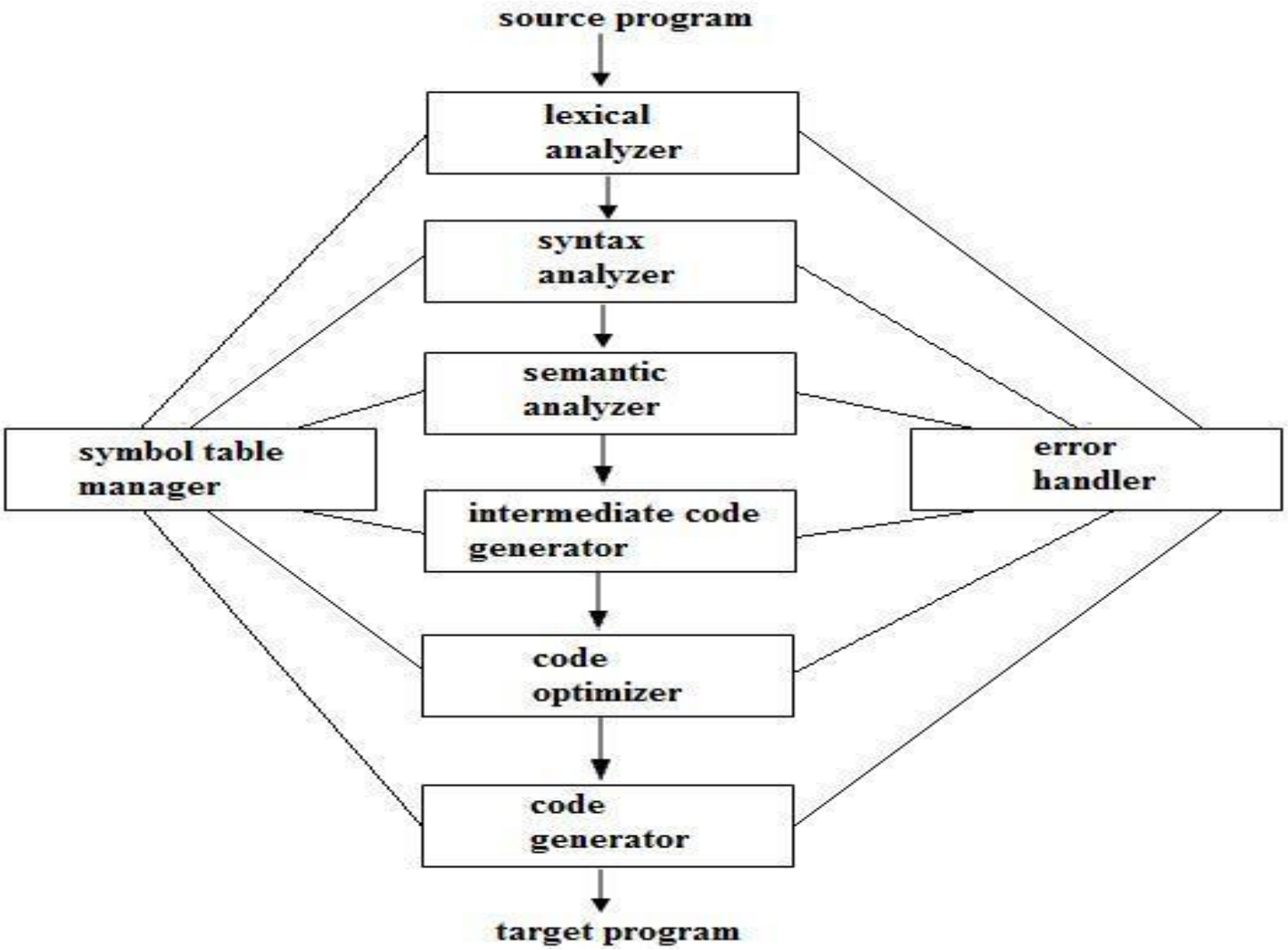
Structure of Compiler

- Input : source program in high level language
- Output : equivalent sequence of machine instructions

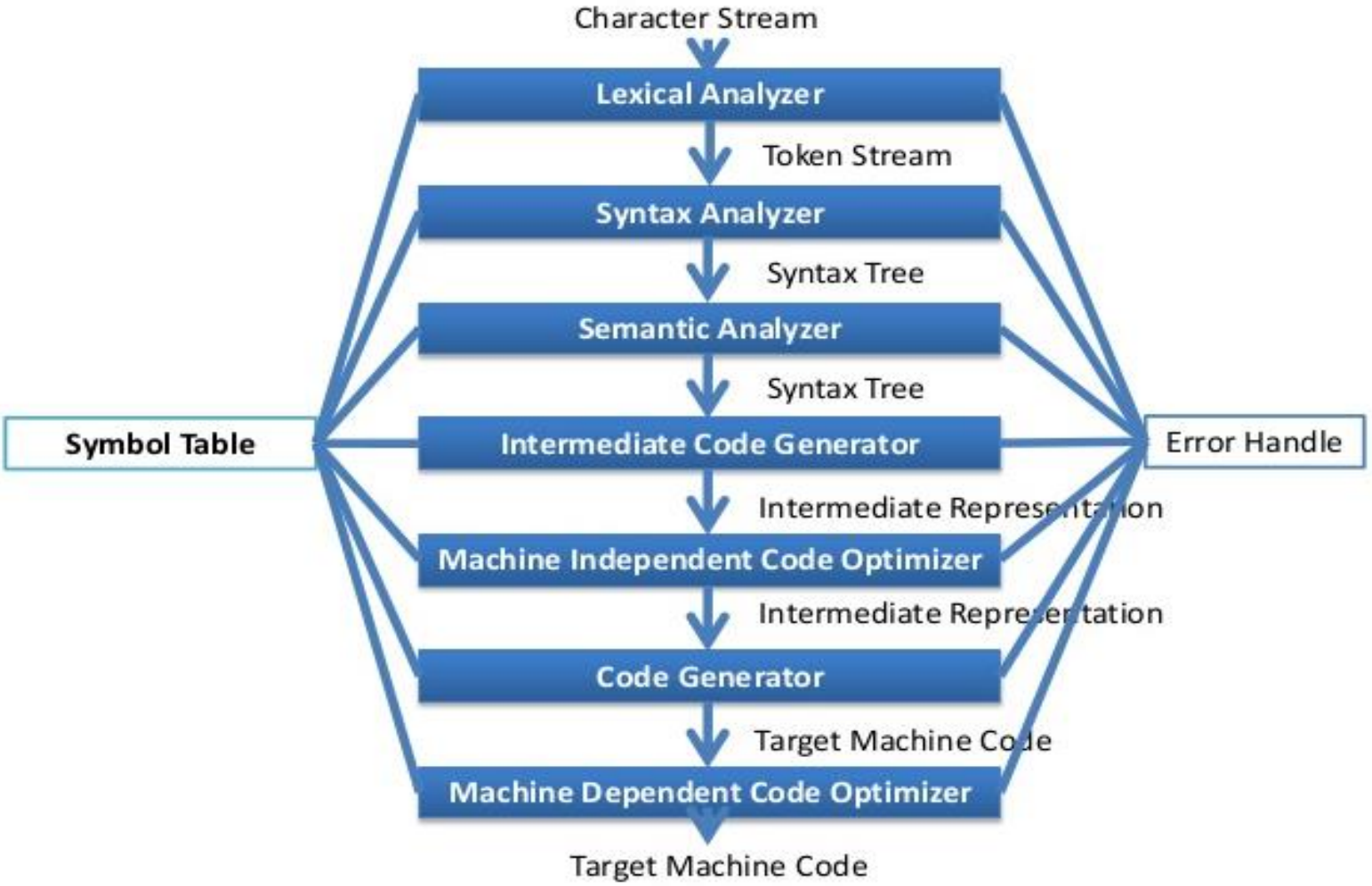
- Phases of compiler

  - Lexical Analysis
  - Syntax Analysis
  - Semantic Analysis
  - Intermediate code generation
  - Code optimization
  - Code generation

Phases of Compiler



Phases of Compiler (Detailed View)



Phases of Compiler: Lexical Analyzer

- First phase of compiler.
- Reads stream of characters making up source program .
- It groups characters into meaningful sequences called **lexemes**.
- For each lexeme, it produces as output token of the form - <token-name, attribute-value>
- Passes tokens to next phase (syntax analysis)

Phases of Compiler: Lexical Analyzer

- **Example**

position = initial + rate * 60

Lexeme	Token
position	<id,1>
=	<=>
Initial	<id,2>
+	<+>
rate	<id,3>
*	<*>
60	<60>

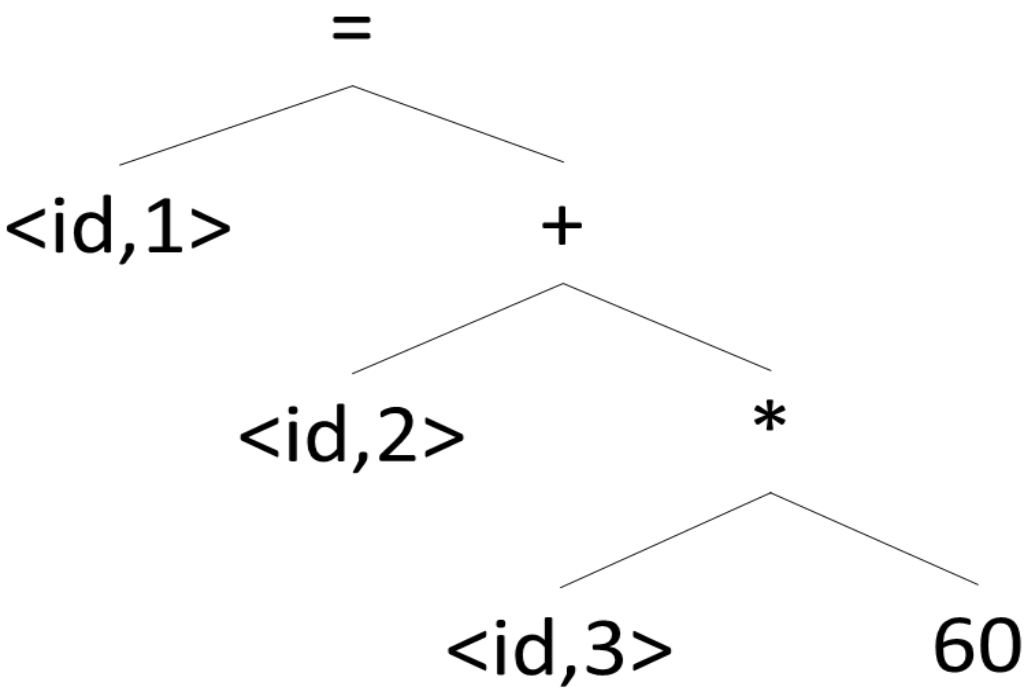
Phases of Compiler: Syntax Analyzer

- Second phase of compiler
- Uses components of tokens produced by lexical analyzer to create tree-like intermediate representation
- It depicts grammatical structure of token stream
- Typical representation is a syntax tree

Phases of Compiler: Syntax Analyzer

- **Example**

position = initial + rate * 60



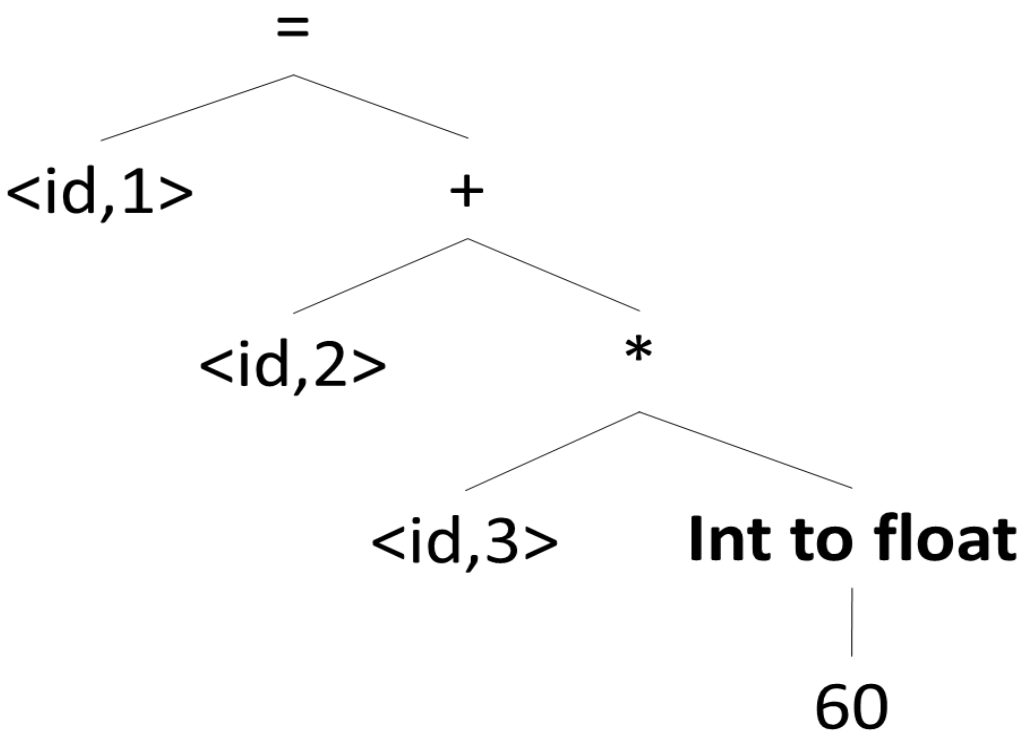
Phases of Compiler: Semantic Analyzer

- Third phase of compiler.
- It uses syntax tree and information in the symbol table to check source program for semantic consistency with language definition.
- It also gathers and saves information and saves it in either syntax tree or symbol table for use in intermediate code generation.
- Important part is **type checking**

Phases of Compiler: Semantic Analyzer

- **Example**

position = initial + rate * 60



Phases of Compiler: Intermediate Code Generator

- Fourth phase of compiler.
- Compiler may construct one or more intermediate representations which can have various forms.
- Syntax trees are a form of intermediate representation.
- Commonly used during syntax and semantic analysis

three address code:-

t1 = inttofloat(60)

t2 = id3 * t1

t3 = id2 + t2

id1 = t3

Phases of Compiler: Code Optimization

- Fifth phase of compiler.
- It is machine independent
- Attempts to improve intermediate code so that better target code will result
- Improves running time of target program

```
t1 = id3 * 60.0
id1 = id2 + t2
```

Phases of Compiler: Code Generation

- Sixth phase of compiler.
- Takes input intermediate representation of source program and maps it to target language.
- Crucial aspect is judicious assignment of registers to hold variables.

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

Phases of Compiler: Symbol Table Management

- Data structure for containing a record for each variable name, with fields for its attributes.
- Attributes may provide information about storage allocated for a name, its type, and its scope.
- In case of procedure names, attributes include number and types of arguments, method of passing each argument, and the type returned.

Phases of Compiler: Error Handler

- One of the most important functions of a compiler is the detection and reporting of errors in the source program.
- The error message should allow the programmer to determine exactly where the errors have occurred.
- Errors may occur in all or the phases of a compiler.
- Whenever a phase of the compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic msg.

Note :- Both of the table-management and error-Handling routines interact with all phases of the compiler.

Interpreter v/s Compiler

Figure: Compiler

Figure: Interpreter

Interpreter v/s Compiler

Interpreter	Compiler
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.
It takes less amount of time to analyse the source code but the overall execution time is slower.	It takes large amount of time to analyse the source code but the overall execution time is comparatively faster.
No intermediate object code is generated, hence are memory efficient.	Generates intermediate object code which further requires linking, hence requires more memory.
Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.	It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.
Programming languages like Python, Ruby use interpreters.	Programming language like C, C++ use compilers.

List of Compilers

Ada compilers
ALGOL compilers
BASIC compilers
C# compilers
C compilers
C++ compilers
COBOL compilers
Common Lisp compilers
Fortran compilers
Java compilers
Pascal compilers
PL/I compilers
Smalltalk compilers

Cross Compiler

Cross Compiler that runs on a machine **'A'** and produces a code for another machine **'B'**. It is capable of creating code for a platform other than the one on which the compiler is running.

In other words,
"A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running"

Example:-

A compiler that runs on a Windows 7 PC but generates code that runs on Android smartphone is a cross compiler.
GNU GCC is an example for cross compiler.

Compiler and Cross Compiler

COMPILER	CROSS COMPILER
A software that translates the computer code written in high-level programming language to machine language	A software that can create executable code for platforms other than the one on which the compiler is running
Helps to convert the high-level source code into machine understandable machine code	A type of compiler that can create executable code for different machines other than the machine it runs on

Compiler Applications

Machine Code Generation

- Convert source language program to machine understandable one.
- Takes care of semantics of varied constructs of source language
- Considers limitations and specific features of target machine
- Automata theory helps in syntactic checks – valid and invalid programs
- Compilation also generate code for syntactically correct programs

Compiler Applications

Format Converters

- Act as interfaces between two or more software packages
- Compatibility of input-output formats between tools coming from different vendors
- Also used to convert heavily used programs written in some older languages (like COBOL) to newer languages (like C/C++)

Compiler Applications

Text Formatting

- Accepts an ordinary text file as input having formatting commands embedded.
- Generates formatted text
- Example: troff, nroff, LaTex etc.

University Question

1. What are the different phases of a compiler? Illustrate compilers internal presentation of source program for the following statement after each phase

   Position=initial +rate * 60

   (or)

2. Explain various phases of a compiler with suitable example.
3. Compare the performance of Compilers and Interpreter

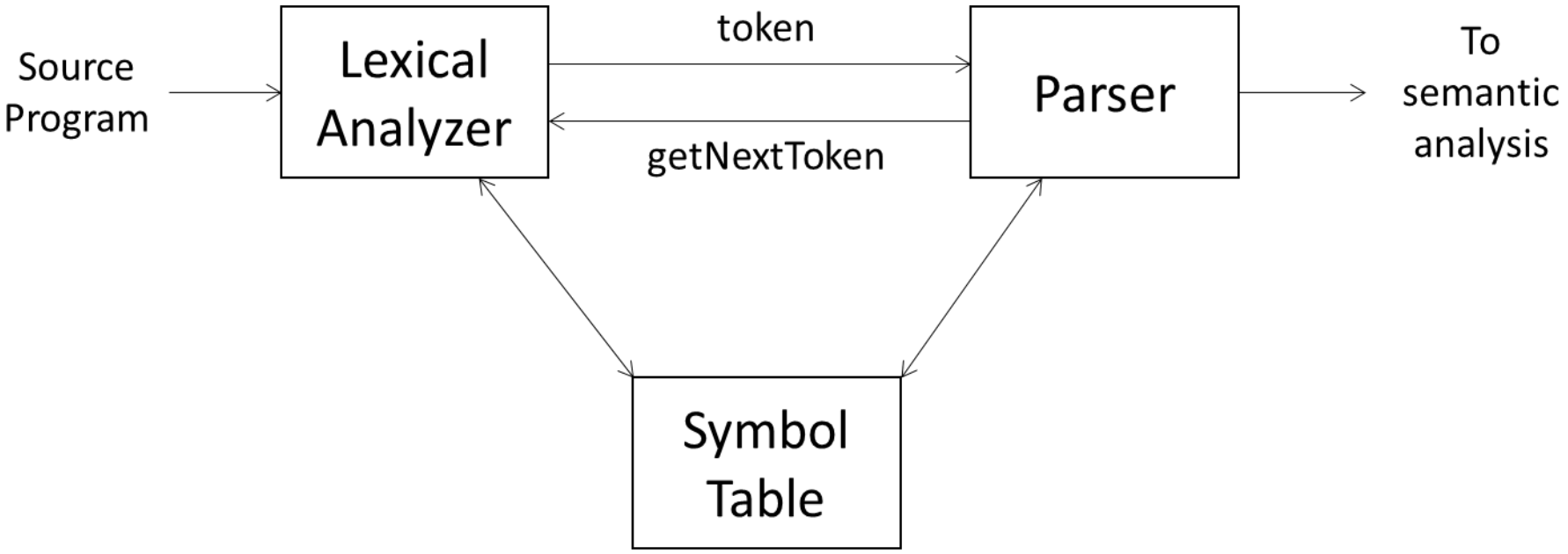
Phase 1 : Lexical Analysis

Role of lexical analyzer

- Lexical analyzer is also known as Scanner.
- Main task is to read input characters of the source program, group them into lexemes and produce output a sequence of tokens for each lexeme.
- When Lexical analyzer discovers a lexeme constituting an identifier, it is inserted into symbol table.
- It performs certain other tasks –
  - Stripping comments and whitespaces
  - Correlating error messages with source program

Phase 1 : Lexical Analysis

Role of lexical analyzer



Interactions between lexical analyzer and parser

Phase 1 : Lexical Analysis

Example:-

```
int add(int a, int b)
//addition of two numbers
{
    int c;
    c=a+b;
    return c;
}
```

S.No.	Lexeme	Token
1.	int	Keyword
2.	add	Identifier
3.	(	Punctuation
4.	int	Keyword
5.	a	Identifier
6.	,	Punctuation
7.	int	Keyword
8.	b	Identifier
9.	)	Punctuation
10.	{	Punctuation
11.	int	Keyword
12.	c	Identifier

S.No.	Lexeme	Token
13.	;	Punctuation
14.	c	Identifier
15.	=	Operator
16.	a	Identifier
17.	+	Operator
18.	b	Identifier
19.	;	Punctuation
20.	return	Keyword
21.	c	Identifier
22.	;	Punctuation
23.	}	Punctuation

Phase 1 : Lexical Analysis

Tokens, Patterns, Lexemes

- Tokens – pair consisting of token name and an optional attribute value. Token names are abstract symbol representing a kind of lexical unit.

- Pattern – sequence of characters that forms a keyword. It is a description of the form that lexemes of a token may take.

- Lexemes – sequence of characters that matches the pattern for a token. It is identified as an instance of that token.

Phase 1 : Lexical Analysis

Tokens, Patterns, Lexemes

Lexeme : Sequence of characters matched with pattern of token

Pattern : Set of rules describes the token

Token : Abstract symbol representing lexical unit.

Example :-

int a= 30;

Phase 1 : Lexical Analysis

Tokens, Patterns, Lexemes

Lexeme : Sequence of characters matched with pattern of token

Pattern : Set of rules describes the token

Token : Abstract symbol representing lexical unit.

Example :-

int a= 30;

position = initial + rate X 60;

Token	Sample Lexemes	Informal description of pattern
if	if	if
While	While	while
Relation	<, <=, = , <>, > >=	< or <= or = or <> or > or >=
Id	count, sun, i, j, pi, D2	Letter followed by letters and digits
Num	0, 12, 3.1416, 6.02E23	Any numeric constant

Phase 1 : Lexical Analysis

Tokens, Patterns, Lexemes

Token is a set of the strings over the source alphabet

eg:- identifier, number, operator, keyword

Pattern is a rule that describes that set.

eg:- Identifier → letter (letter/digit)*

Lexeme is a sequence of characters matching that pattern.

eg:- position,+,60

Phase 1 : Lexical Analysis

- **Scanner** is the only part of the compiler which reads a complete program.
- Lexical Analyzer is the only phase of the compiler that reads the source program character by character.(from the Disk by using getchar())
- It takes approx. 25-30% of the compiler's time.
- So, Lexical Analysis phase consumes considerable amount of time, due to which, compilation time goes low.
- Hence, speed of Lexical Analysis is major concern!!

Phase 1 : Lexical Analysis

Input Buffering

Solution:-

As large amount of time consumption takes place in moving characters, Specialized buffering techniques have been developed to reduce the amount of over head required to process an input character by lexical Analyzer

- A block of data is first read into a BUFFER and then it is read by Lexical Analyzer

Phase 1 : Lexical Analysis

Input Buffering

- Lexical Analyzer needs to look ahead several characters beyond the lexeme for a pattern before a match can be announced.
- Lexical Analyzer use a function ungetc() to push look ahead characters back into an input stream.

Input Buffering are of two types :-

1. One Buffer Scheme
2. Two Buffer Scheme

Lexical analyzer scans the input from left to right one character at a time.

Phase 1 : Lexical Analysis

Input Buffering

One Buffer Scheme:-

In this scheme, only one buffer is used to store the input string

Size of the Buffer = N Characters

Where N = Number of Character on one Disk block (1024 or 4096)

Phase 1 : Lexical Analysis

Input Buffering

It uses two Pointers :

Two pointers *lexemeBegin* and *forward* are maintained.

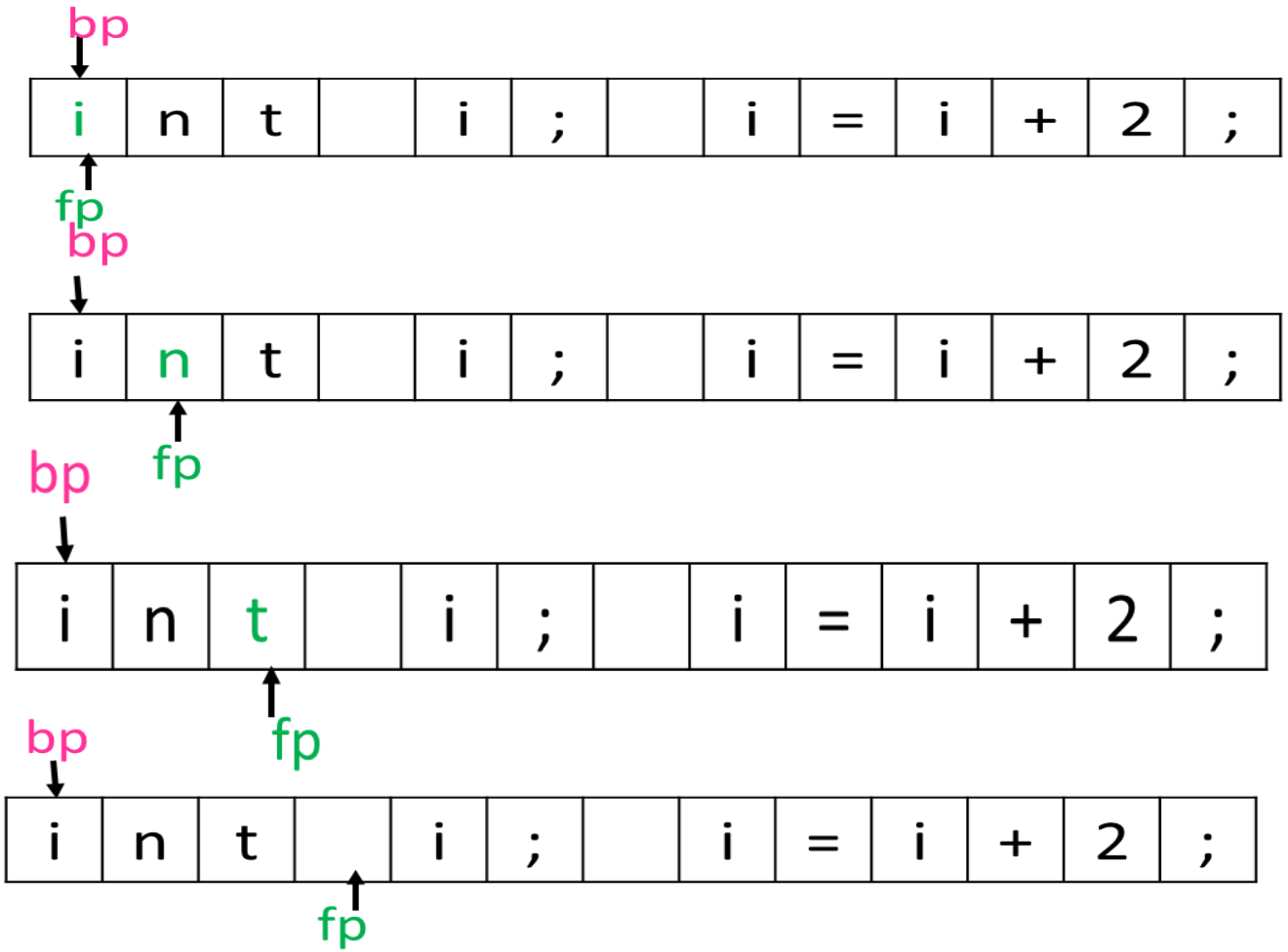
lexemeBegin (bp) points to the beginning of the current lexeme which is yet to be found.

forward (fp)scans ahead until a match for a pattern is found.

- Once a lexeme is found, *lexemebegin* is set to the character immediately after the lexeme which is just found and *forward* is set to the character at its right end.
- Current lexeme is the set of characters between two pointers.

Phase 1 : Lexical Analysis

One Buffer Scheme:- One system command reads N Characters ---

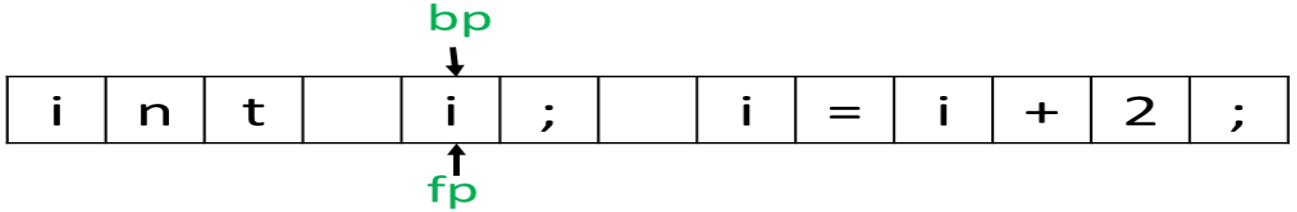


int→token (keyword)

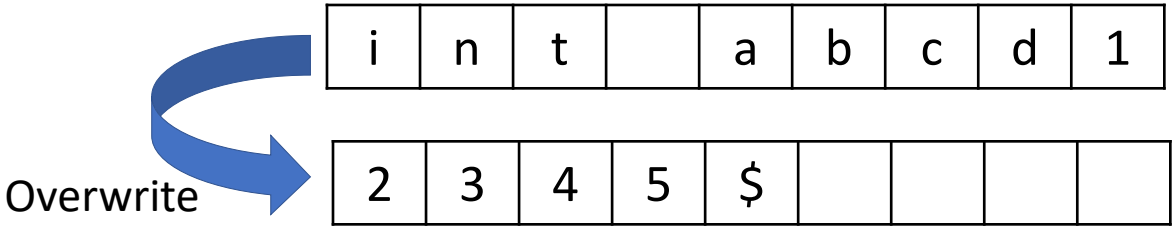
Phase 1 : Lexical Analysis

Input Buffering→ One Buffer Scheme

When fp points to blank space, it indicates end of lexeme and int is identified



Eg :- int abcd12345$;



Hence error will occur!!!

Drawback :- if a lexeme is very long then, it crosses the buffer boundary, to scan rest of the lexeme, the buffer has to be refilled, that makes overwriting the first of lexeme in input buffer

Phase 1 : Lexical Analysis

Input Buffering

- To overcome the drawback of One Buffer Scheme Two Buffer Scheme(Buffer Pairs) is introduced.
- Task of reading the source program is made difficult by the fact, that often one or more characters have to be read before recognizing the lexeme.
- Unsure of end of identifier until a character that doesn't belong to the identifier is encountered.
- Hence, Introduced Two buffer Scheme to handle large look ahead safely.

Phase 1 : Lexical Analysis

Input Buffering → Two Buffer Scheme

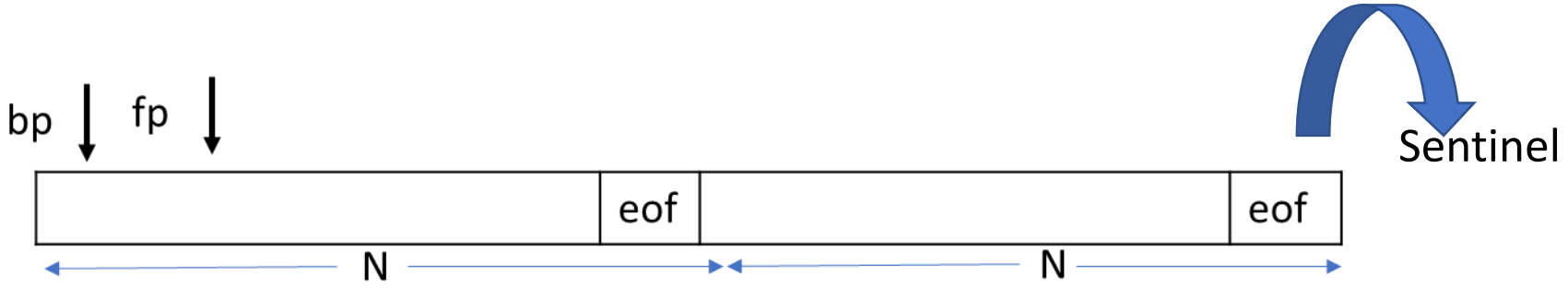
- Here two Buffers are used to store the input string.
- Both the buffers are N character array. Here, both buffers are scanned alternatively.
- When end of current buffer is reached, the other buffer is filled.

Phase 1 : Lexical Analysis

Input Buffering- *Buffer Pairs*

- The Buffer pairs are used to hold the input data.
- Buffers are divided into two N-Character Halves where, N is the number of characters e.g.- 1024 or 4096.

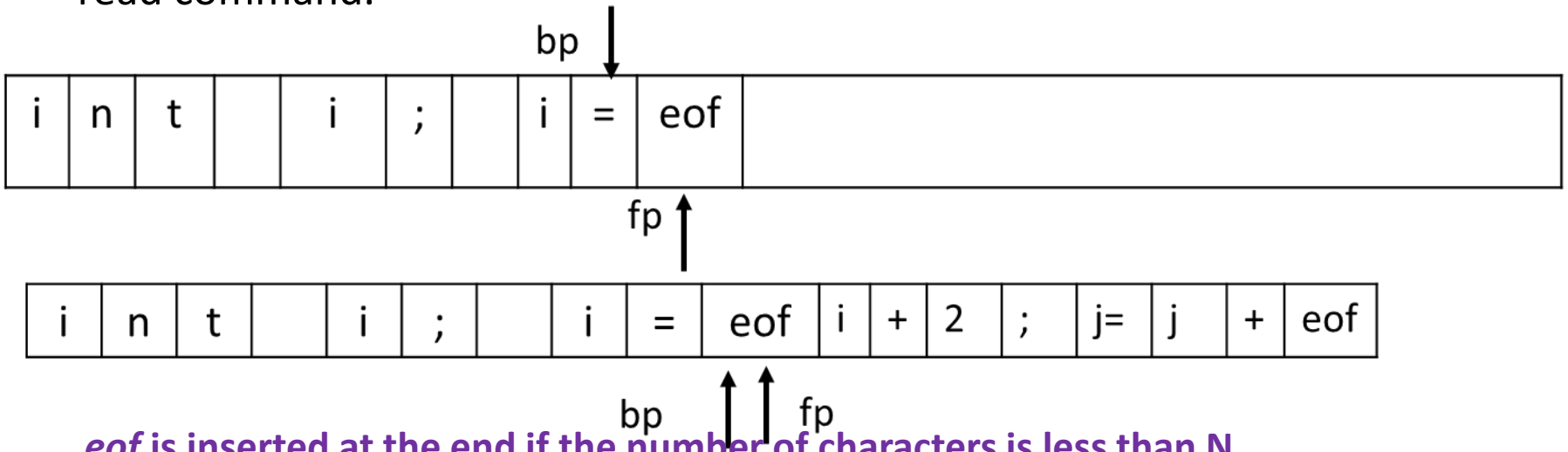
To identify end of buffer, eof character introduced at the end of both the buffer. Eof is called a Sentinel



Phase 1 : Lexical Analysis

Input Buffering- *Buffer Pairs*

- Consists of two buffers, each consists of N-character size which are reloaded alternatively.
- N-Number of characters on one disk block, e.g., 4096.
- N characters are read from the input file to the buffer using one system read command.

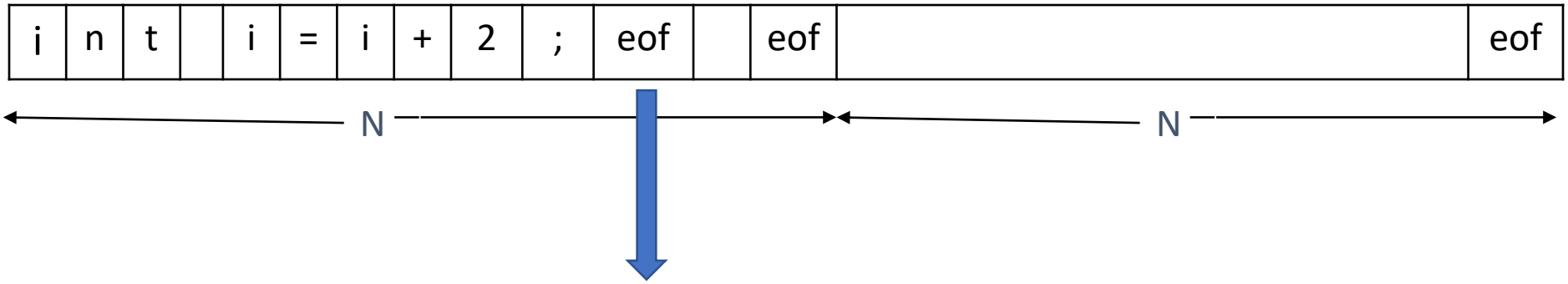


***eof* is inserted at the end if the number of characters is less than N.**

Phase 1 : Lexical Analysis

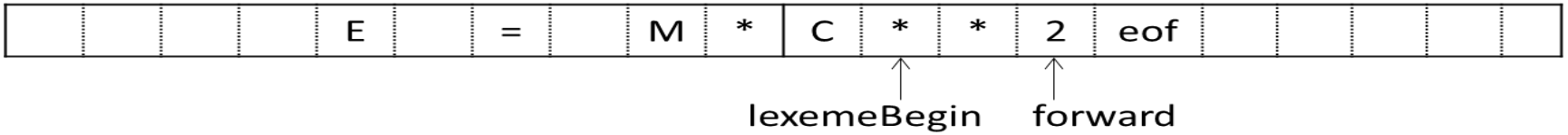
Input Buffering- *Buffer Pairs*

- **if the number of characters is less than N.**



eof within a buffer signifying end of input that means terminate Lexical Analysis

Buffer Pairs



Phase 1 : Lexical Analysis

Input Buffering- *Buffer Pairs*

Disadvantages of this scheme

- This scheme works well most of the time, but the amount of lookahead is limited.
- This limited lookahead may make it impossible to recognize tokens in situations where the distance that the forward pointer must travel is more than the length of the buffer.

(eg.) DECLARE (ARGl, ARG2, . . . , ARGn) in PL/1 program;

- It cannot determine whether the DECLARE is a keyword or an array name until the character that follows the right parenthesis.

Phase 1 : Lexical Analysis (Summary)

Input Buffering *Sentinels*

- In Two Buffer Scheme, each time when the forward pointer is moved, a check is done to ensure that one half of the buffer has not moved off. If it is done, then the other half must be reloaded.
- Therefore the ends of the buffer halves require two tests for each advance of the forward pointer.

Test 1: For end of buffer.

Test 2: To determine what character is read.

- The usage of sentinel reduces the two tests to one by extending each buffer half to hold a sentinel character at the end.
- The sentinel is a special character that cannot be part of the source program. ***(eof* character is used as sentinel).**

Phase 1 : Lexical Analysis (Summary)

Input Buffering- *Sentinels*

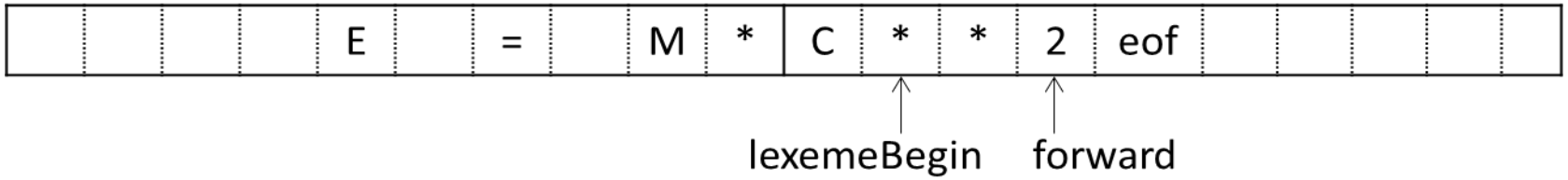
Advantages

- Most of the time, It performs only one test to see whether forward pointer points to an *eof.*
- Only when it reaches the end of the buffer half or *eof,* it performs more tests.
- Since N input characters are encountered between *eofs,* the average number of tests per input character is very close to 1.

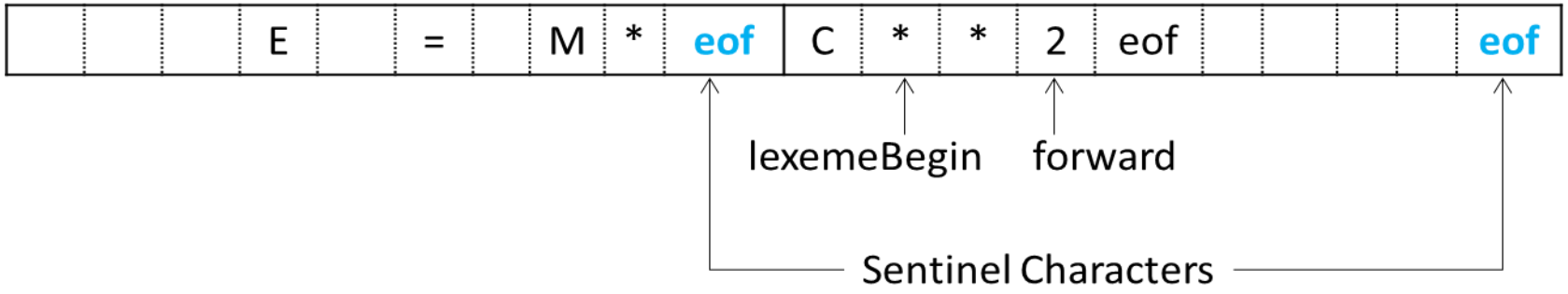
Phase 1 : Lexical Analysis

Input Buffering- *Sentinels*

Buffer Pairs



Sentinels



Phase 1 : Lexical Analysis

Specification of Tokens

- Regular expression are used to specify lexeme patterns.

letter → A|B|……|Z|a|b|……|z

digit → 0|1|2|……|9

id → letter (letter|digit)*

digits → digit* (.digits)? (E[+-]? digits)?

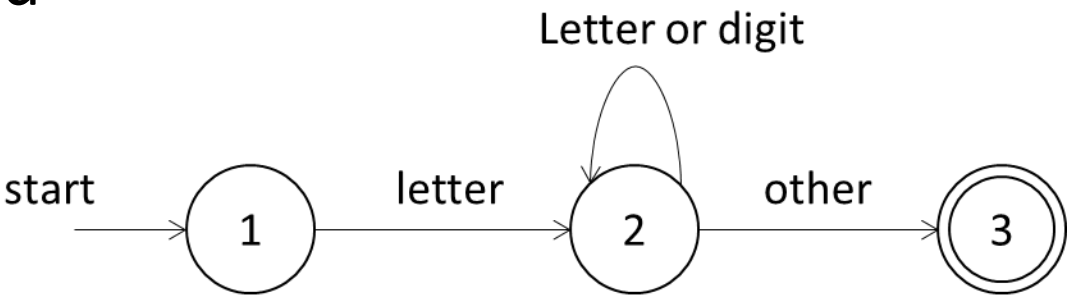
op → <|>|<=|>=|=|=|<>

ws → (blank|tab|newline)+

Phase 1 : Lexical Analysis

Recognition of Tokens

- Transition diagrams are used in recognition of tokens and keywords.
- Start and final states indicates lexeme has been found



Transition diagram for identifier

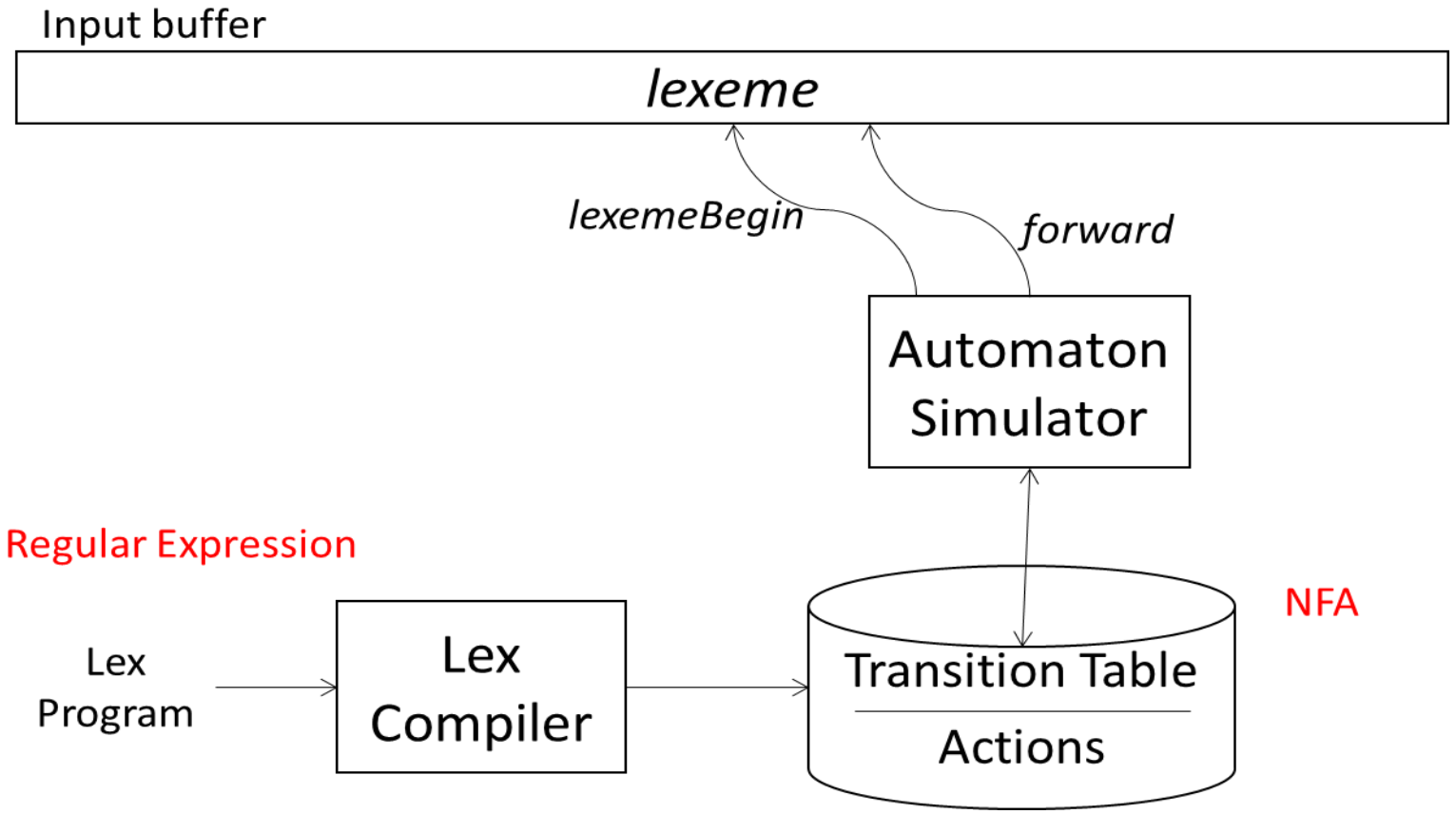
Phase 1 : Lexical Analysis

Design of Lexical Analyzer Generator

- Program that serves as Lexical Analyzer includes Fixed program simulates an automaton.
- Lexical analyzer consists of components that are created from the Lex program by Lex itself.
- Components are –
  - **Transition table** for automaton
  - **Functions passed** directly through Lex to output
  - **Actions** from input program

Phase 1 : Lexical Analysis

Design of Lexical Analyzer Generator



Structure of the Generated Analyzer

Phase 1 : Lexical Analysis

Design of Lexical Analyzer Generator

- To construct the automaton, take each regular expression pattern in the ***Lex*** program and convert it to an NFA

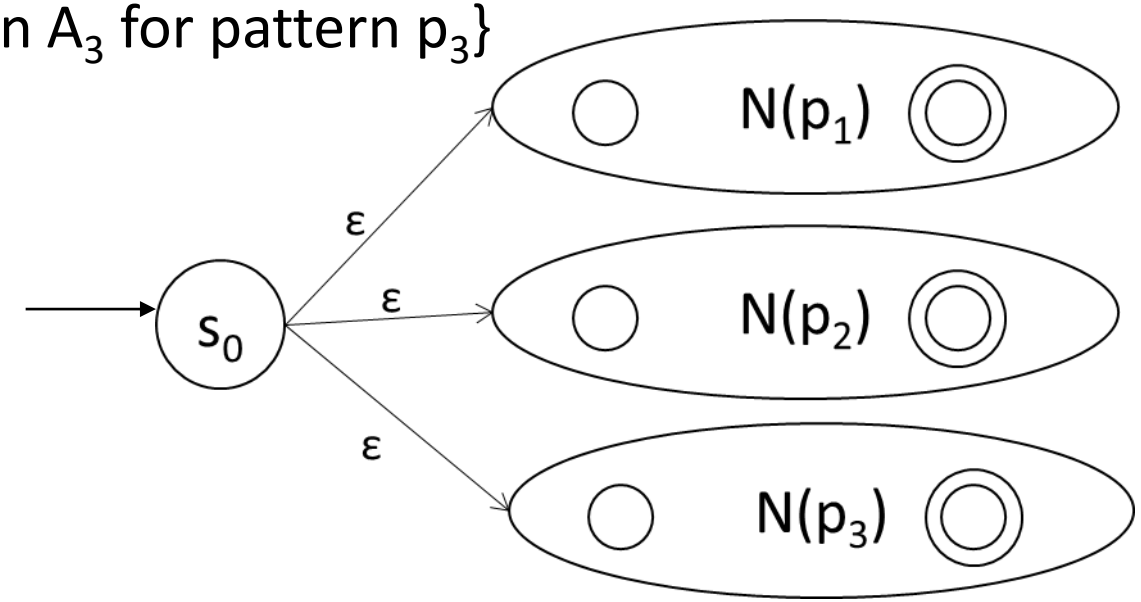
- Now, we need a single automaton that recognizes lexemes matching any pattern, then we combine all the NFA's into one by introducing a new start state with ε transitions to each start state of the NFAs

Phase 1 : Lexical Analysis

Design of Lexical Analyzer Generator

Example :-

- To identify the following patterns
  - a {action A_1 for pattern p_1}
  - abb {action A_2 for pattern p_2}
  - a^*b^+ {action A_3 for pattern p_3}



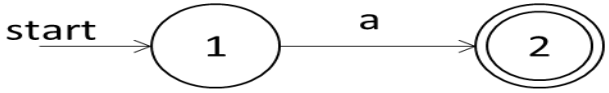
Phase 1 : Lexical Analysis

Design of Lexical Analyzer Generator

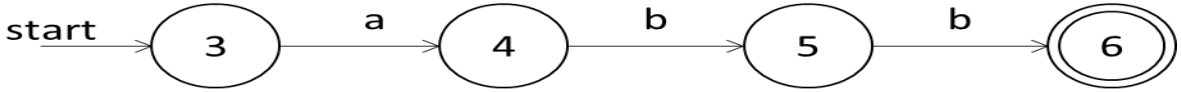
Example :-

- To identify the following patterns
  - a {action A_1 for pattern p_1}
  - abb {action A_2 for pattern p_2}
  - a^*b^+ {action A_3 for pattern p_3}

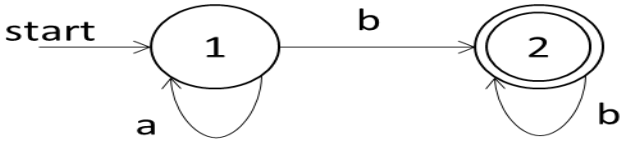
NFA for a



NFA for abb



NFA for a^*b^+

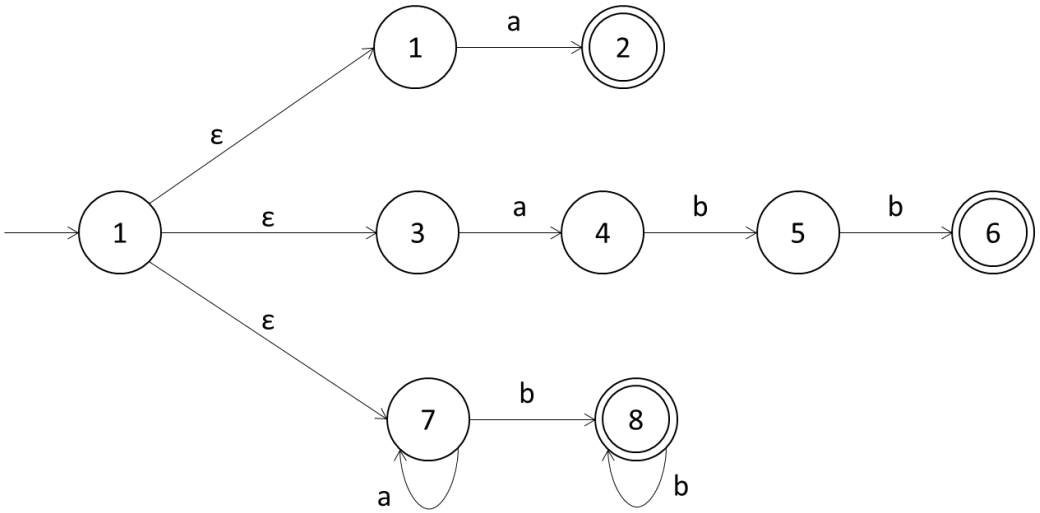


Phase 1 : Lexical Analysis

Design of Lexical Analyzer Generator

Example :-

- To identify the following patterns
  - a {action A_1 for pattern p_1}
  - abb {action A_2 for pattern p_2}
  - a^*b^+ {action A_3 for pattern p_3}



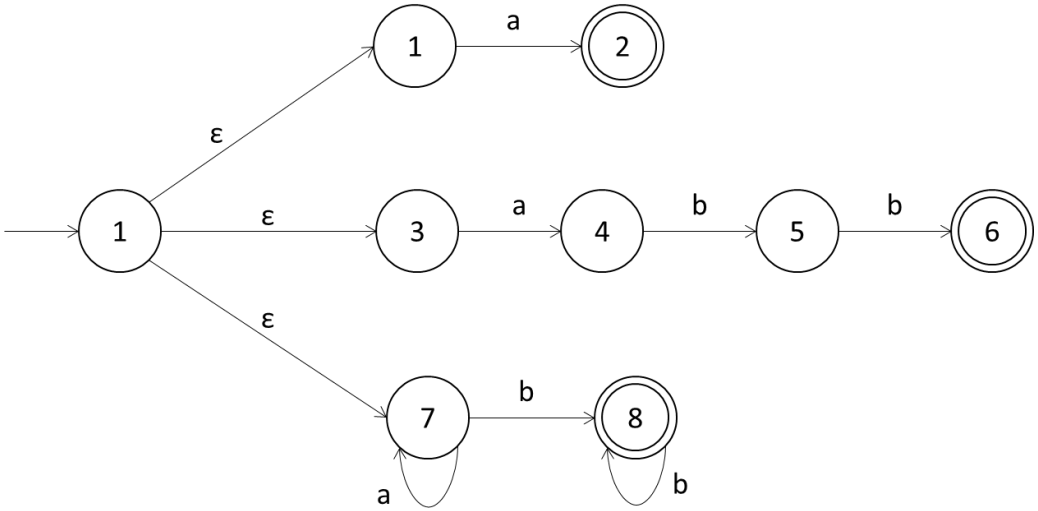
Combined NFA

Phase 1 : Lexical Analysis

Design of Lexical Analyzer Generator

Example :-

- To identify the following patterns
  - a {action A_1 for pattern p_1}
  - abb {action A_2 for pattern p_2}
  - a^*b^+ {action A_3 for pattern p_3}



Combined NFA

- Eventually, NFA simulation reaches a point where there are no next states i.e. no longer prefix of the input would ever get the NFA to an accepting state.
- Thus, We can say that a longer prefix is a lexeme matching some pattern

Phase 2 : Syntax Analyser

- Syntax Analysis-
  - Role of Context Free Grammar in Syntax Analysis
- Types of Parsers:
  - Top down parser- LL(1)
  - Bottom up parser- Operator precedence parser, SLR

Syntax Analyser

Syntax Definition

- Syntax of a language is specified using a notation called – Context Free Grammar (CFG)

- A grammar naturally describes the hierarchical structure of most programming language constructs.

- Example – ***if*** (expression) statement ***else*** statement

  (statement → ***if*** (expression) statement ***else*** statement)

Syntax Analyser

Context-Free Grammar

- Terminals – basic symbols from which strings are formed. Also called as “token name” or “token”.
- Nonterminal – syntactic variables that denote sets of strings. They help define the language generated by the grammar. They impose a hierarchical structure on the language.
- Start symbol – set of strings denoted by start symbol is the language generated by the grammar.

Syntax Analyser

Context-Free Grammar

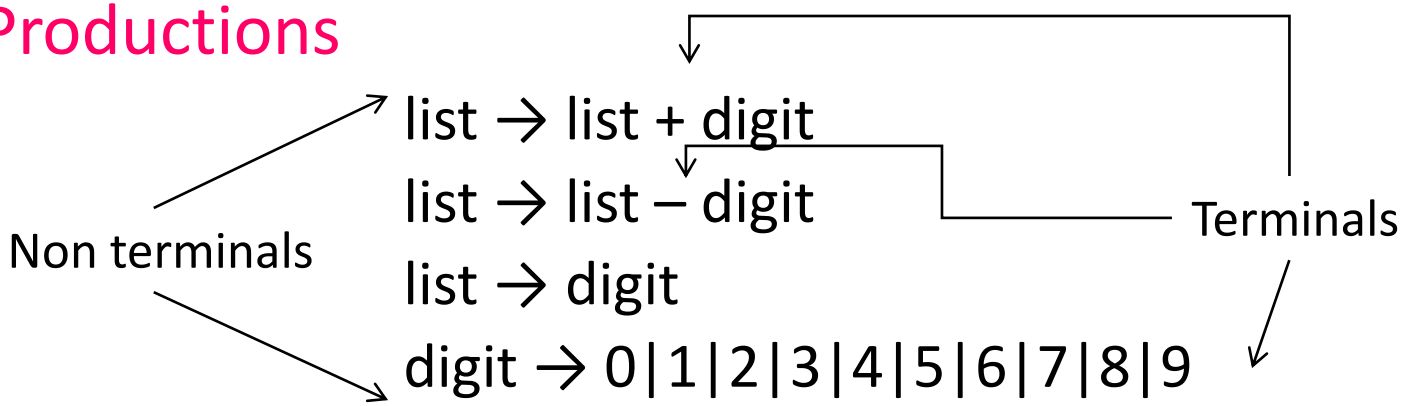
- Productions – specify manner in which terminals and non-terminals can be combined to form strings.

- Each production consists of –
  - Nonterminal : head of the production. Defines some strings denoted by the head
  - The symbol "→"
  - Body : consisting of 0 or more terminals and non-terminals. Describes one way in which strings of the nonterminal at the head can be constructed.

Syntax Analyser

Syntax Definition

- Productions



- Above grammar can also be written as

  list → list + digit | list – digit | digit

  digit → 0|1|2|3|4|5|6|7|8|9

Syntax Analyser

Derivations and Parse Tree

- A grammar derives a string by beginning with the start symbol and repeatedly replacing a nonterminal by the body of a production for that non terminal.

- Parsing is the problem of taking a string of terminals and figuring out how to derive it from the start symbol of the grammar, and if it cannot be derived, then reporting syntax errors within the string.

- A parse tree pictorially shows how the start symbol of a grammar derives a string in the language.

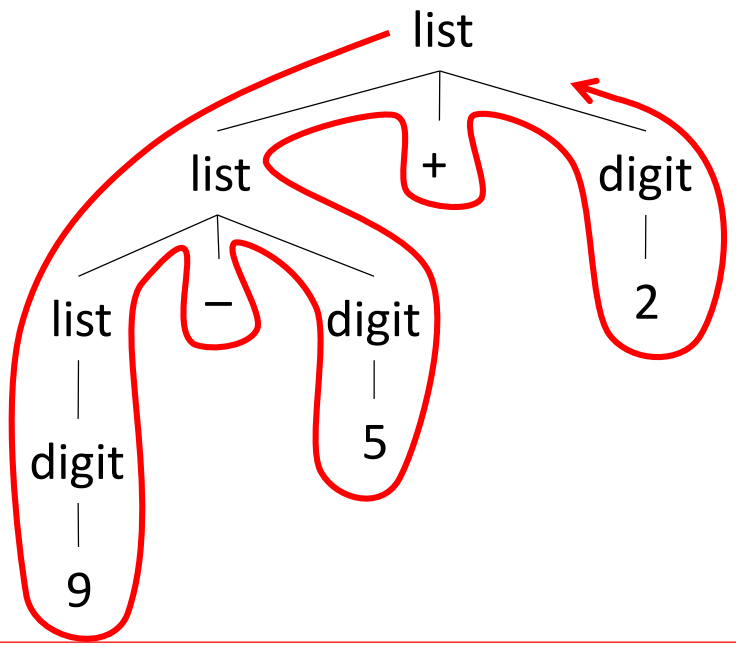
Syntax Analyser

Derivations and Parse Tree

- Example : list → list + digit |list-digit |digit

derive 9–5+2

Parse Tree



Leftmost derivation

list → list + digit

→ **list – digit** + digit

→ **digit** – digit + digit

→ **9** – digit + digit

→ 9 – **5** + digit

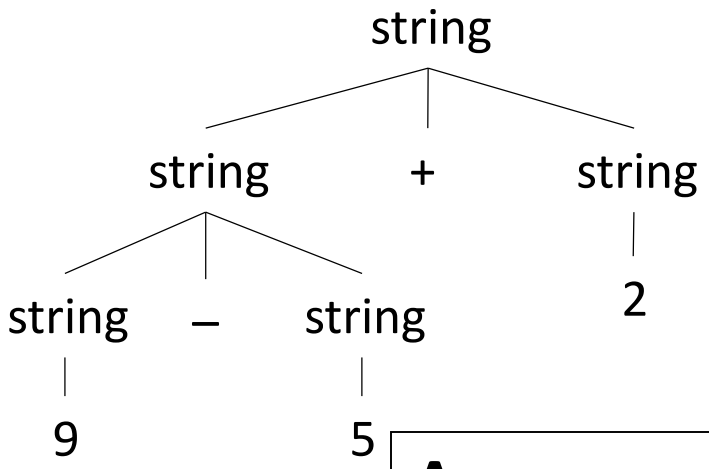
→ 9 – 5 + **2**

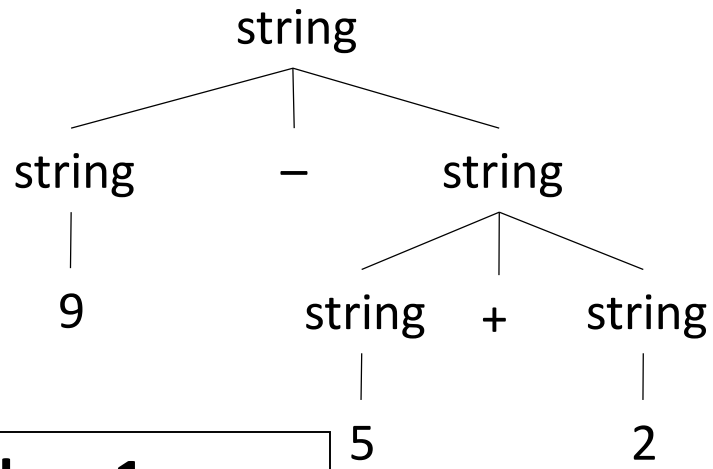
Syntax Analyser

Ambiguity

- Derive **9–5+2** for the grammar

string → string+string|string–string|0|1|2|3|4|5|6|7|8|9





A grammar with more than 1 parse tree/derivation for a sentence is said to be ambiguous

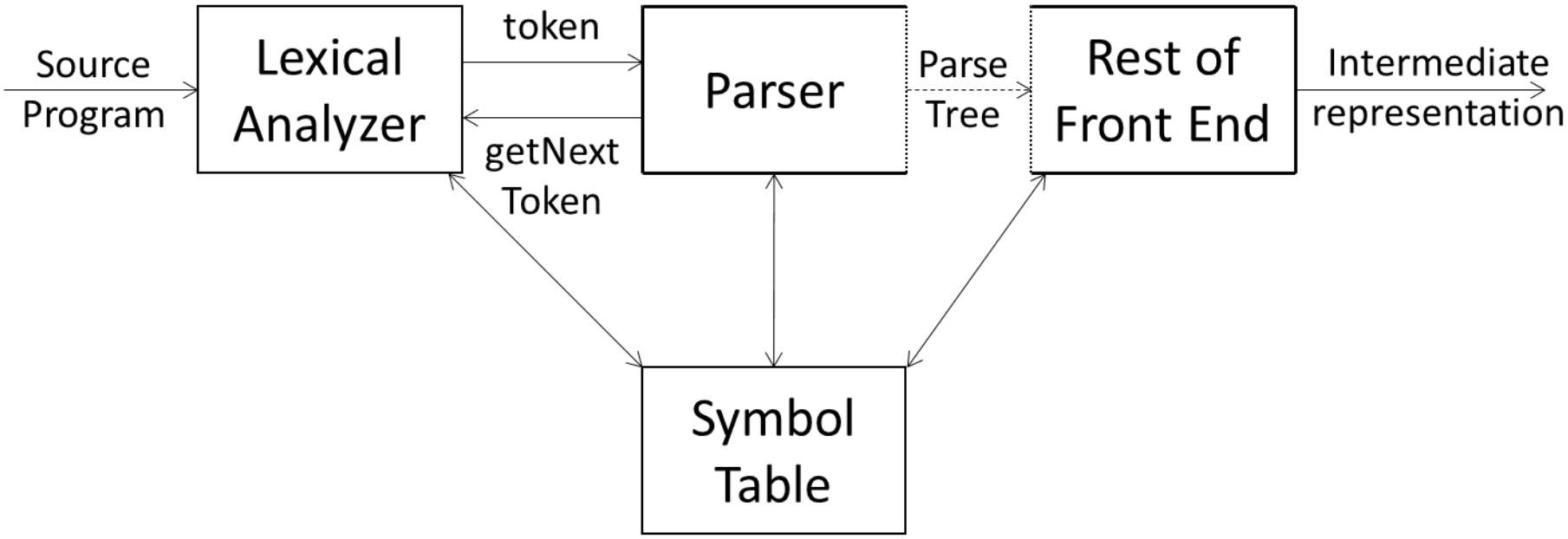
Syntax Analyser

Benefits of Grammar

- It gives precise, easy to understand syntactic specification of a programming language.
- Efficient parsers can be constructed from certain classes of grammars.
- Useful for detecting errors and translating source program into correct object code.
- Allows language to be evolved iteratively, by adding new constructs to perform new tasks.

Parser

Role of a Parser



Position of Parser in compiler model

Role of Parser/Syntax Analyser

Role of a Parser

- Takes input a string of tokens form lexical analyzer
- Verifies that string of token names can be generated by the grammar for the source language
- Report syntax errors in an intelligible fashion and recover from commonly occurring errors to continue processing remainder of the program
- Parsers commonly used in compilers –
  - Top-down Parser
  - Bottom-up Parser

Parsing Methods

- Top Down Parsing
  - Problem of constructing a parse tree for input string, starting from root node and creating nodes of the parse tree in preorder
  - Can also be viewed as finding leftmost derivation for an input string
- Bottom Up Parsing
  - Construction of parse tree for an input string beginning at the leaves and working up towards the root.
  - Can carry out translation without building an explicit tree

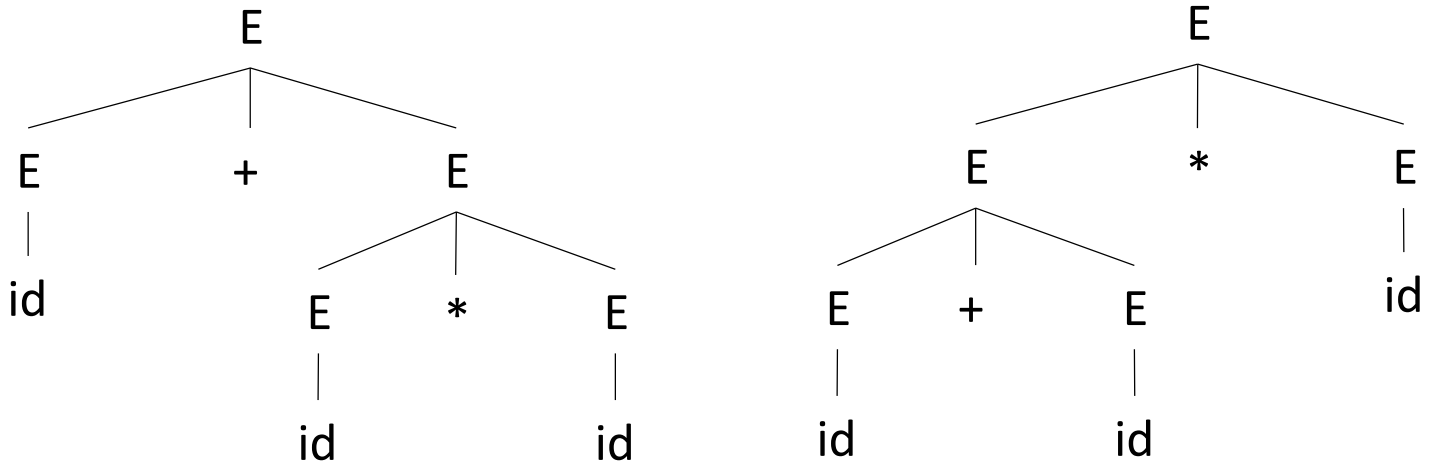
Parser

Example :-

- Derive the string **id+id*id** for the grammar

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

- Check if the grammar is ambiguous



Parsing Methods

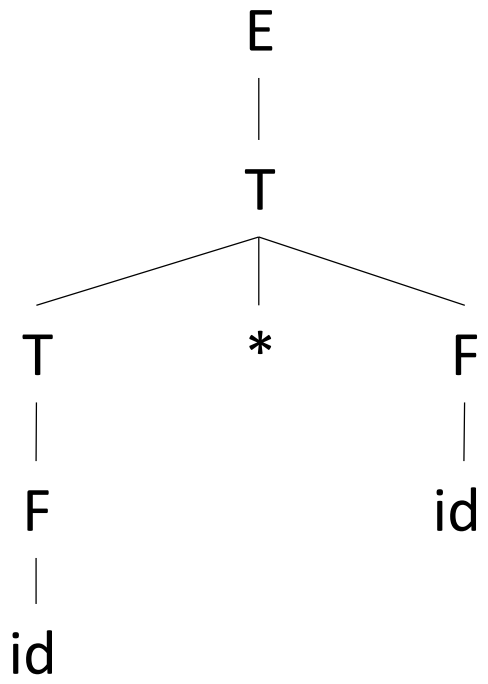
Example

- Top-Down parser for **id*id**

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

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

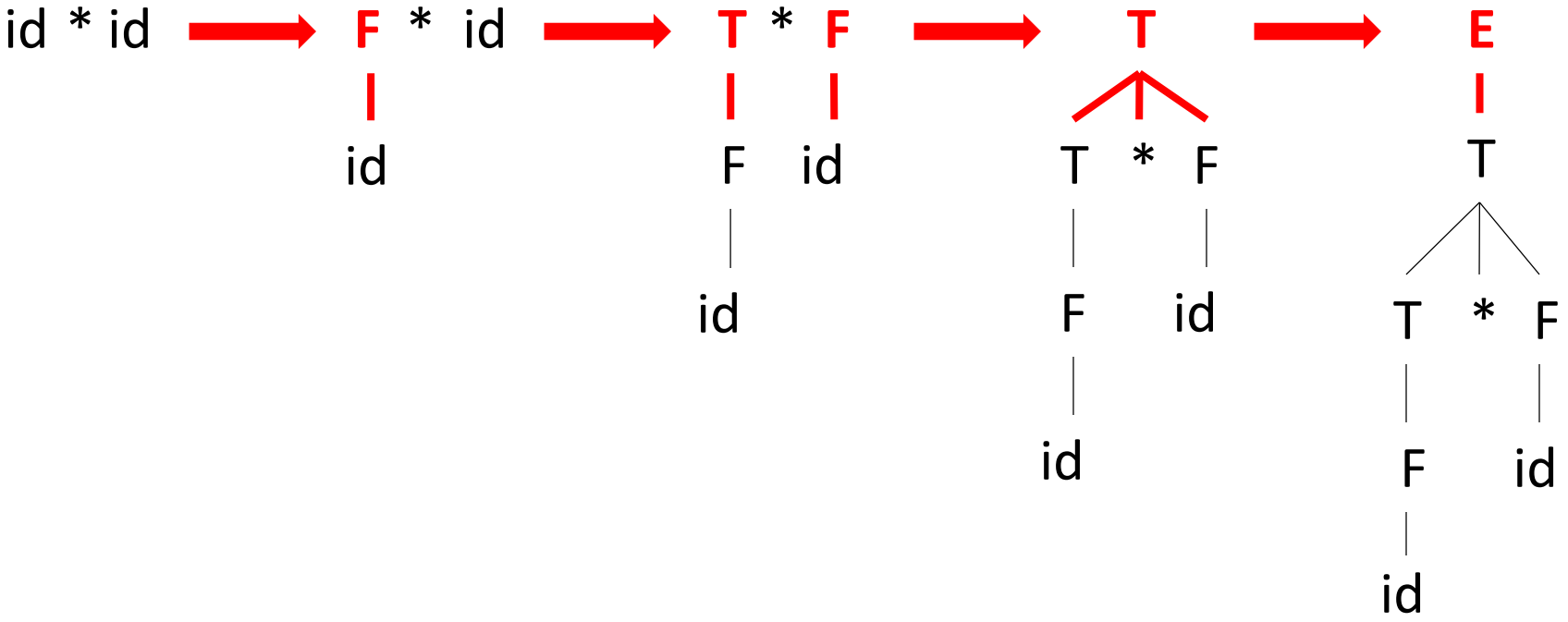
$$F \rightarrow (E) \mid id$$



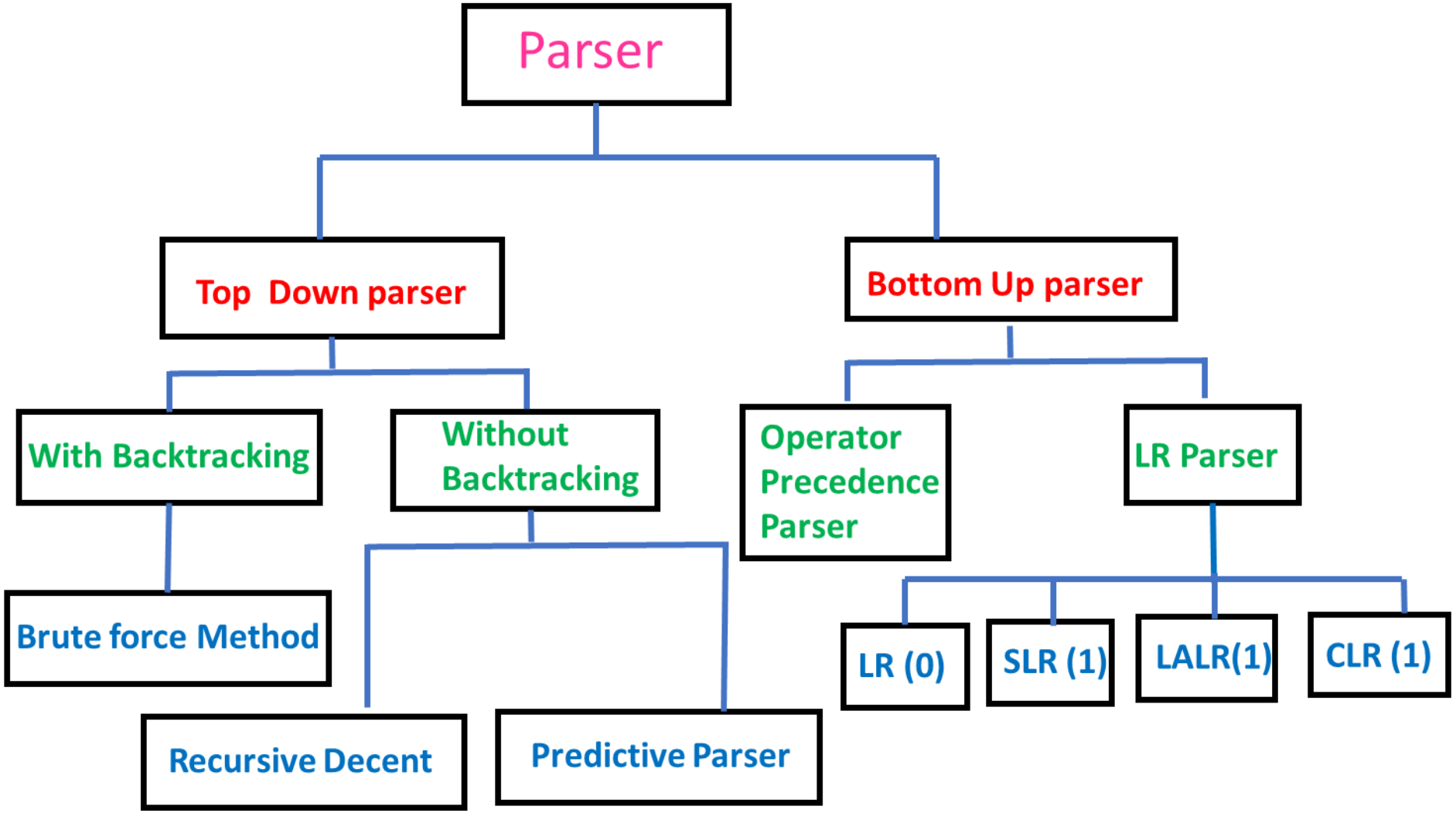
Parsing Methods

Example

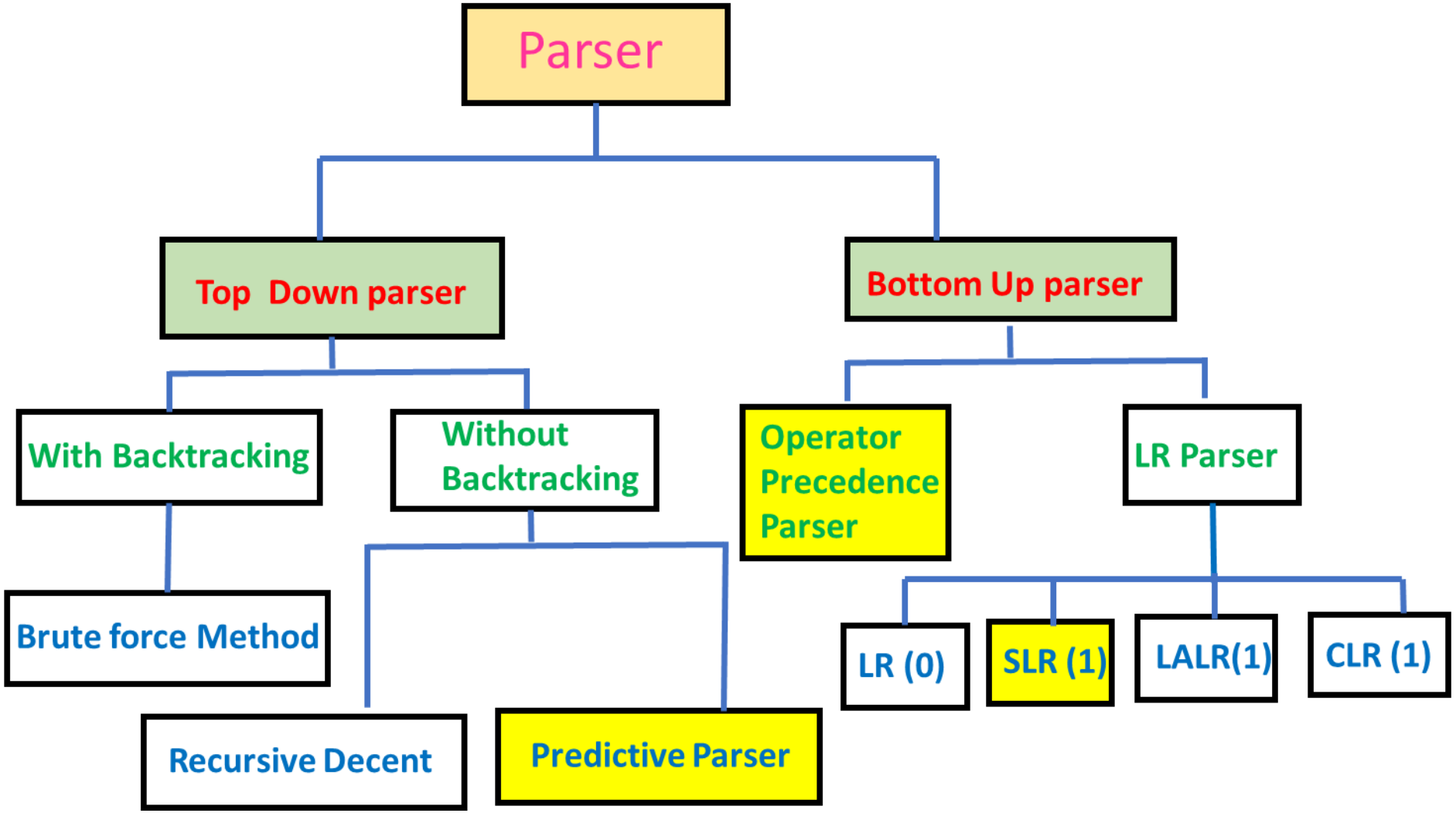
- Bottom-Up Parser



Classification of Parser Techniques

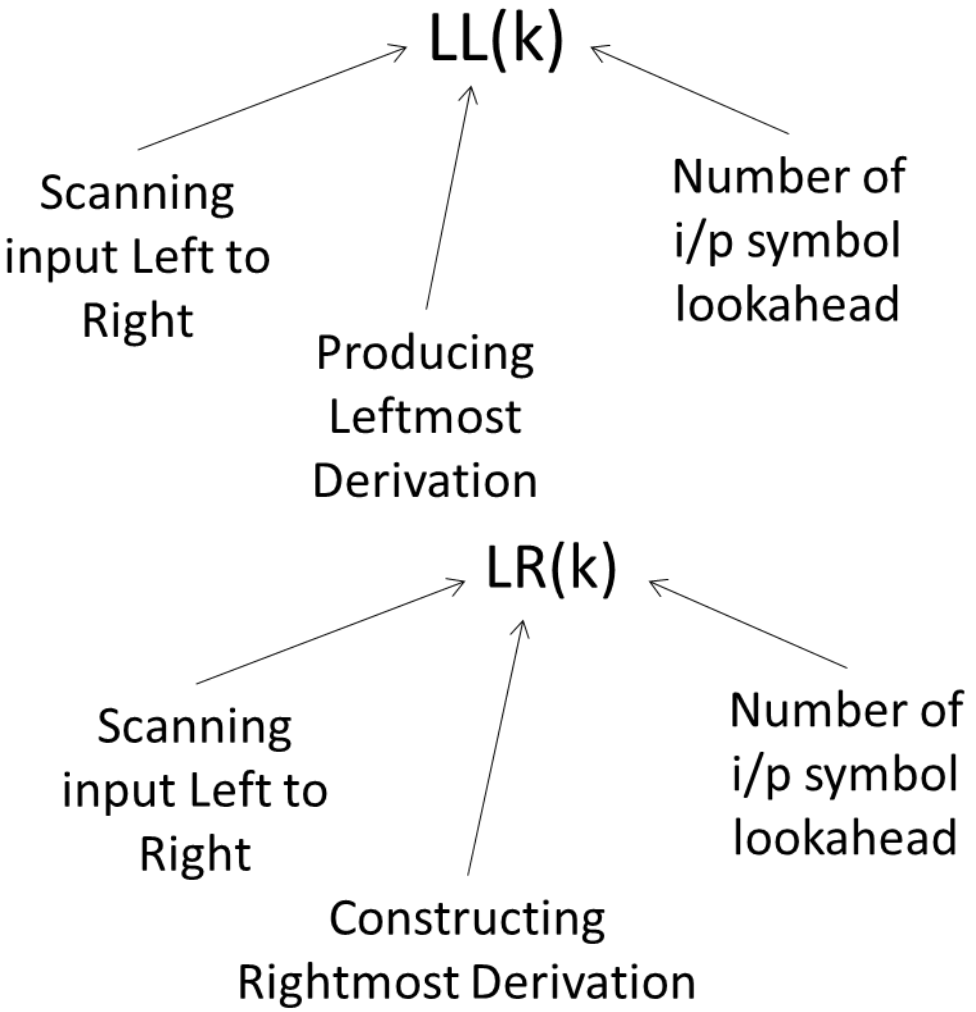


Classification of Parser Techniques



Types of Parsers

- Top-Down Parser
  - Recursive Descent Parser
  - Predictive or LL(1) Parser
- Bottom-Up Parser
  - Shift Reduce Parser
  - SLR or LR(0) Parsers
  - Canonical LR or LR
  - Look Ahead LR or LALR



Top Down Parsing

- In Top Down Parsing, the Parse tree is created top to bottom, i.e. start from the root and work down to the leaves.
- Top down parsing can be viewed as an attempt to find left most derivation for an input string to check the validity of a string.
- IN TDP, a string is derived using LMD from a Non-Terminal (Start Symbol) of a given Grammar.
- TDP expands a parse tree from the start symbol to the leaves by consuming tokens generated by Lexical Analyzer

Top Down Parsing

- TDP uses the concept of **Backtracking**
- At each step of a top-down parser, the key problem is that of determining the production to be applied for a nonterminal
- Once production is chosen, rest of parsing consists of matching terminal symbols in production body with the input string
- Cannot handle left-recursive and left-factored grammars.

Top Down Parsing

- Brute force Method
  - Uses the concept of Backtracking.
- Recursive Descent Parser
  - General form of Top-Down Parser
  - May use backtracking to find correct production
- Predictive Parser (also called LL(1) parser)
  - Special case of Recursive Descent parser which doesn't require backtracking
  - Uses one symbol lookahead at each step to make parsing action decisions

Top Down Parsing with Backtracking

Brute force Method

- Here, Full Backtracking is used to create a parse tree until the correct/ given string is generated.

- In case,
- if the given string does not belong to the grammar, then also all possible combinations are checked before declaring that parse tree cannot be generated. Hence does not belong to the grammar.

Top Down Parsing with Backtracking

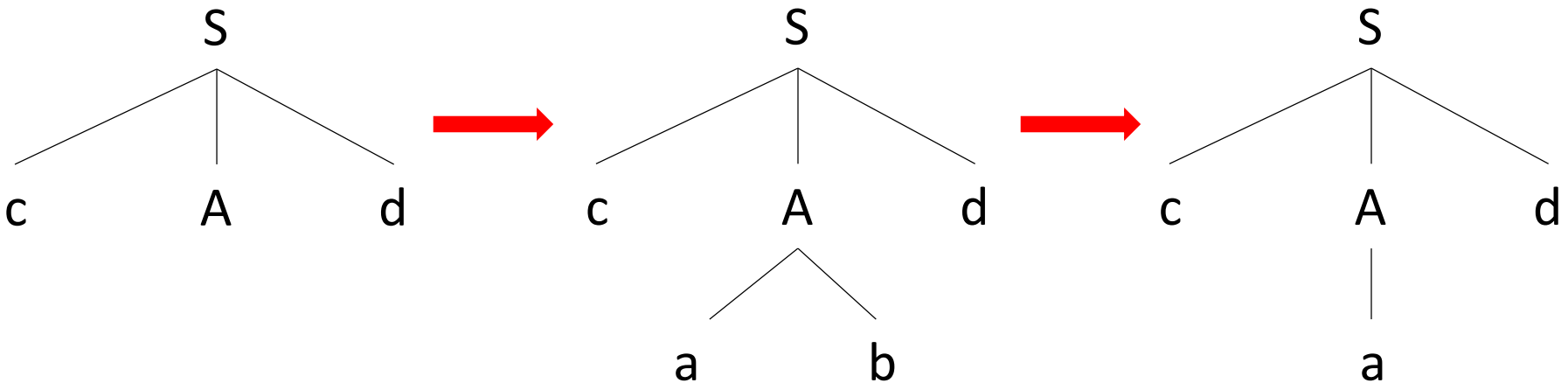
Brute force Method

Consider a CFG

S → c A d

A → a b | a

Derive the string ***cad*** from the given grammar



Top Down Parsing

Left Recursion and Left Factoring

- Left Recursion Elimination

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

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

- Left Factoring

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n$

$A \rightarrow \alpha A'$

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

Top Down Parsing-Recursive Descent Parser

- General form of Top-down Parsing
- Consists of a set of procedures (one for each non-terminal)
- Execution begins with start symbol
- If procedure body scans entire input string, it halts and announces success
- May require backtracking to find correct production
- Backtracking is not very efficient as compared to tabular methods
- Left recursion can cause infinite loop

Top Down Parsing-Recursive Descent Parser

```
void A()
 {
        Choose an A-Production, A→X1X2……Xk;
        for (i=1 to k)
        {
                if (Xi is a non-terminal )
                        call procedure Xi();
                else if (Xi equals current i/p symbol a)
                        advance i/p to next symbol;
                else /*an error has occurred*/;
        }
}
```

Top Down Parsing-Recursive Descent Parser

Consider the Grammar

E→iE’

E’→+iE’/ ε

Recursive Functions:-

```
E()
 {
        if (l==‘i’)
        {       match (‘i’);
                E’();
        }
}
```

```
E’()
 {
        if (l==‘+’)
        {       match (‘+’);
                match (‘i’);
                E’();
        }
        else
        return;
}
```

Top Down Parsing-Recursive Descent Parser

Consider the Grammar

E→iE'

E'→+iE'/ ε

Recursive Functions:-

```
match (char t)
 {
   if (l==t)
        l=getchar();
    else
        printf("error");
}
```

```
main()
{
        E();
        if (l=='$')
        Printf("Parsing Success");
}
```

Top Down Parsing-Recursive Descent Parser

Advantages:-

- Easy to Construct.
- Overhead associated with backtracking is eliminated.

Disadvantage:-

- We cannot write Recursive Decent Parser for all types of CFG.
- It cannot be implemented only for those languages which supports recursive procedure calls

Top Down Parsing- Predictive Parser

- It is of the class of LL(1) grammar
- It is also known as Table Driven Parser/ Non-recursive Decent Parser
- No left-recursion or left-factoring
- Uses FIRST() and FOLLOW() functions
- They allow to choose which production to apply based on next input symbol

Top Down Parsing- Predictive Parser

- A grammar G is LL(1) if and only if A→α|ß are 2 distinct production of G, the following conditions hold:
  - For no terminal ***a*** does both α and ß derive strings beginning with ***a***
  - At most one of α and ß can derive the empty string

    (First(α) and First(ß) must be disjoint)
  - If ß→ε, then α doesn't derive any string beginning with a terminal in FOLLOW(A) and likewise if α→ε

    (if ε is in First(ß), First(α) and Follow(α) are disjoint, and likewise if ε is in First(α))

Top Down Parsing- Predictive Parser

To Compute FIRST

- FIRST(α) : set of terminals that begin strings derived from α (where α is a symbol or nonterminal)

- FIRST(α)
  - If α is a terminal, then FIRST(α) = α
  - If α is a nonterminal then check production $\alpha \rightarrow \beta\delta$
    - If β is a terminal, then FIRST(α) = β
    - If β is a nonterminal, then FIRST(α) = FIRST(β)

Top Down Parsing- Predictive Parser

To Compute Follow

- FOLLOW(A) : for a non-terminal A, set of terminals ***α*** that can appear immediately to the right of A in some sentential form ($ is a special "end marker" symbol that is placed in FOLLOW set of start symbol)

- FOLLOW(A)
  - If A is a start symbol then its FOLLOW contains **$**
  - Check production S→*α*A*ß*
    - If *ß* is a terminal, then FOLLOW(A) = *ß*
    - If *ß* is a non-terminal, then FOLLOW(A) = FIRST(*ß)*
      - If FIRST(*ß*) contains *ε*, then FOLLOW(A)=FIRST(*ß*) U FOLLOW(*ß*)
    - If *ß* is empty i.e.S→*α*A, then FOLLOW(A) = FOLLOW(S)

Top Down Parsing- Predictive Parser

Step 1 : Eliminate Left Recursion

Step 2 : Eliminate Left Factoring

Step 3 : find FIRST() and FOLLOW() for all symbols

Step 4 : Create parsing table (M)

- For each terminal ***a*** in First(A), add A→α to M[A,a]
- If ε is in First(α), then for each terminal b in Follow(A), add A→a to M[A,b]. If ε is in First(α) and $ is in Follow(A), add A→a to M[A,$] as well

Top Down Parsing- Predictive Parser

Example :-

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

Step 1 : Eliminate Left Recursion

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Step 2 : Eliminate Left Factoring

No left Factoring

Top Down Parsing- Predictive Parser

Step 3 : Find FIRST() and FOLLOW()

FIRST(E) = FIRST(T) = FIRST(F) = (,id

FIRST(E') = +,ε

FIRST(T) = FIRST(F) = (,id

FIRST(T') = *,ε

FIRST(F) = (,id

FOLLOW(E) =$,)

FOLLOW(E') = FOLLOW(E) = $,)

FOLLOW(T) = FIRST(E') U FOLLOW(E') = +,$,)

FOLLOW(T') = FOLLOW(T) = +,$,)

FOLLOW(F) = FIRST(T') U FOLLOW(T') = *,+,$,)

Top Down Parsing- Predictive Parser

Step 4 : Create Parsing Table M

Non Terminal	Terminals					
	id	+	*	(	)	$
E						
E’						
T						
T’						
F						

Top Down Parsing- Predictive Parser

Step 4 : Create Parsing Table M

Non Terminal	Terminals					
	id	+	*	(	)	$
E	E→TE'					
E'						
T						
T'						
F						

Top Down Parsing- Predictive Parser

Step 4 : Create Parsing Table M

Non Terminal	Terminals					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'						
T						
T'						
F						

Top Down Parsing- Predictive Parser

Step 4 : Create Parsing Table M

Non Terminal	Terminals					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'				
T						
T'						
F						

Top Down Parsing- Predictive Parser

Step 4 : Create Parsing Table M

Non Terminal	Terminals					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T						
T'						
F						

Top Down Parsing- Predictive Parser

Step 4 : Create Parsing Table M

Non Terminal	Terminals					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'					
T'						
F						

Top Down Parsing- Predictive Parser

Step 4 : Create Parsing Table M

Non Terminal	Terminals					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'						
F						

Top Down Parsing- Predictive Parser

Step 4 : Create Parsing Table M

Non Terminal	Terminals					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'			T'→*FT'			
F						

Top Down Parsing- Predictive Parser

Step 4 : Create Parsing Table M

Non Terminal	Terminals					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F						

Top Down Parsing- Predictive Parser

Step 4 : Create Parsing Table M

Non Terminal	Terminals					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id					

Top Down Parsing- Predictive Parser

Step 4 : Create Parsing Table M

Non Terminal	Terminals					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Bottom Up Parsing

- Construction of parse tree for an input string, beginning from leaves (terminals) and working up towards the root node (start symbol)

- General style of bottom-up parsers are **shift-reduce** parsers

- Largest class of grammars for which shift-reduce parsers can be built is **LR** class of grammars

Bottom Up Parsing

Types :-

- Shift Reduce
  - General form of bottom-up parsing
  - Uses stack to hold grammar symbols and input buffer to hold string to be parsed
  - $ is used to mark bottom of stack and also right end of input

- LR parsers
  - Scans input left to right
  - Constructs rightmost derivation in reverse
  - Can describe more languages than LL(k) grammars

Bottom Up Parsing

Reductions

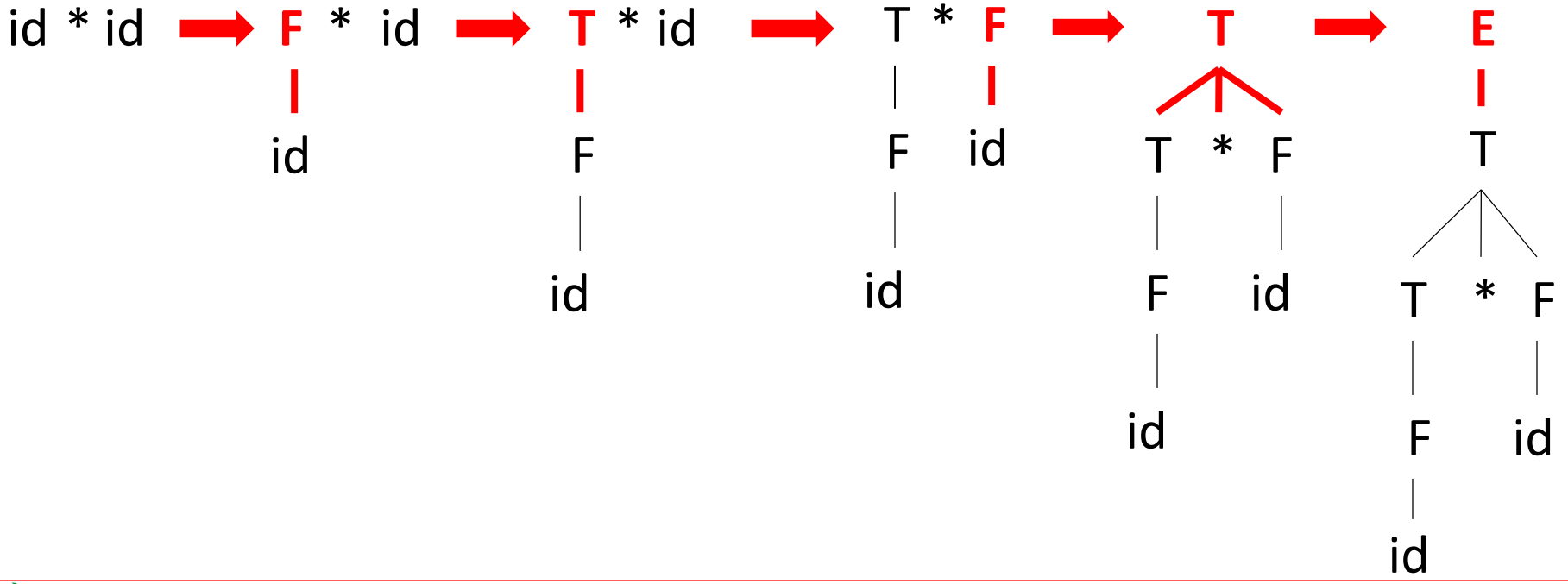
- Process of "reducing" a string to the start symbol of the grammar.
- At each reduction step, a specific substring matching the body of a production is replaced by the non-terminal at the head of that production.
- The key decisions during bottom-up parsing are about when to reduce and about what production to apply, as the parse proceeds.

Bottom Up Parsing

Example

- Shift Reduce is reverse process of rightmost derivation

$E \rightarrow T \rightarrow T * F \rightarrow T * id \rightarrow F * id \rightarrow id * id$



Bottom Up Parsing

Handle Pruning

- A **Handle** is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.
- The leftmost substring that matches the body of some production need not be a handle.

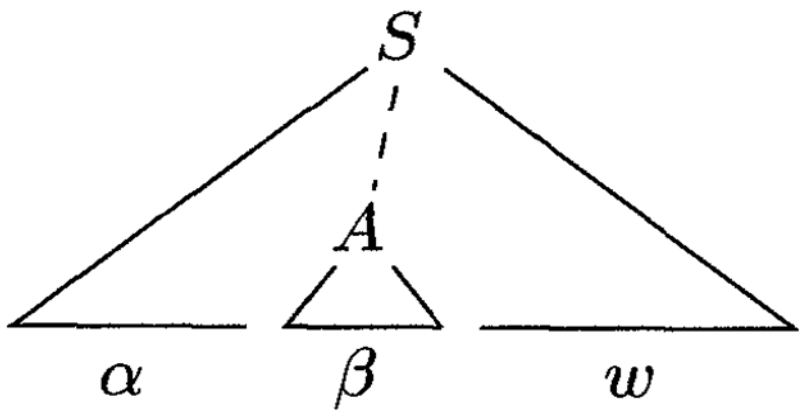
Right Sentential Form		Handle	Reducing Production
$\mathbf{id}_1 * \mathbf{id}_2$		$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$		F	$T \to F$
$T * \mathbf{id}_2$	$\longrightarrow$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$		$T * F$	$E \to T * F$

Cannot replace T with E→T as it becomes E*id which cannot be derived

Bottom Up Parsing

Example

- Formally, if $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta w$, then production $A \rightarrow \beta$ in the position following α, is a handle of $\alpha\beta w$



- The string w to the right of the handle must contain only terminal symbols

Bottom Up Parsing

Example

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$$

- Let w= $\gamma_n$where γ_n the nth right sentential form of some yet unknown rightmost derivation.

- To reconstruct this derivation in reverse order, we locate the handle β_nn γ_nn and replace $\beta_n$the head of the relevant production $A_n \rightarrow \beta_n$ to obtain the previous right-sentential form γ_{n-1}

Bottom Up Parsing

Shift Reduce Parser

- General form of bottom-up parsing
- Uses stack and input buffer
- Handle always appears at the top of the stack just before it is identified as the handle
- During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string β of grammar symbols on top of the stack.
- It then reduces β to the head 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

Bottom Up Parsing

Example

STACK	INPUT	ACTION

Bottom Up Parsing

Example

STACK	INPUT	ACTION
$	$\mathbf{id}_1 * \mathbf{id}_2$ $	shift

Bottom Up Parsing

Example

STACK	INPUT	ACTION
$	$\mathbf{id}_1 * \mathbf{id}_2$ $	shift
$ $\mathbf{id}_1$	$* \mathbf{id}_2$ $	reduce by $F \rightarrow \mathbf{id}$

Bottom Up Parsing

Example

STACK	INPUT	ACTION
$	$\mathbf{id}_1 * \mathbf{id}_2$ $	shift
$ $\mathbf{id}_1$	$* \mathbf{id}_2$ $	reduce by $F \rightarrow \mathbf{id}$
$ F	$* \mathbf{id}_2$ $	reduce by $T \rightarrow F$

Bottom Up Parsing

Example

STACK	INPUT	ACTION
$	$\mathbf{id}_1 * \mathbf{id}_2$ $	shift
$ $\mathbf{id}_1$	$* \mathbf{id}_2$ $	reduce by $F \rightarrow \mathbf{id}$
$ F	$* \mathbf{id}_2$ $	reduce by $T \rightarrow F$
$ T	$* \mathbf{id}_2$ $	shift

Bottom Up Parsing

Example

STACK	INPUT	ACTION
$	$\mathbf{id}_1 * \mathbf{id}_2$ $	shift
$ $\mathbf{id}_1$	$* \mathbf{id}_2$ $	reduce by $F \rightarrow \mathbf{id}$
$ F	$* \mathbf{id}_2$ $	reduce by $T \rightarrow F$
$ T	$* \mathbf{id}_2$ $	shift
$ $T *$	$\mathbf{id}_2$ $	shift

Bottom Up Parsing

Example

STACK	INPUT	ACTION
$	$\mathbf{id}_1 * \mathbf{id}_2$ $	shift
$ $\mathbf{id}_1$	$* \mathbf{id}_2$ $	reduce by $F \rightarrow \mathbf{id}$
$ F	$* \mathbf{id}_2$ $	reduce by $T \rightarrow F$
$ T	$* \mathbf{id}_2$ $	shift
$ $T *$	$\mathbf{id}_2$ $	shift
$ $T * \mathbf{id}_2$	$	reduce by $F \rightarrow \mathbf{id}$

Bottom Up Parsing

Example

STACK	INPUT	ACTION
$	$\mathbf{id}_1 * \mathbf{id}_2$ $	shift
$ $\mathbf{id}_1$	$* \mathbf{id}_2$ $	reduce by $F \rightarrow \mathbf{id}$
$ F	$* \mathbf{id}_2$ $	reduce by $T \rightarrow F$
$ T	$* \mathbf{id}_2$ $	shift
$ $T *$	$\mathbf{id}_2$ $	shift
$ $T * \mathbf{id}_2$	$	reduce by $F \rightarrow \mathbf{id}$
$ $T * F$	$	reduce by $T \rightarrow T * F$

Bottom Up Parsing

Example

STACK	INPUT	ACTION
$	$\mathbf{id}_1 * \mathbf{id}_2$ $	shift
$ $\mathbf{id}_1$	$* \mathbf{id}_2$ $	reduce by $F \rightarrow \mathbf{id}$
$ F	$* \mathbf{id}_2$ $	reduce by $T \rightarrow F$
$ T	$* \mathbf{id}_2$ $	shift
$ $T *$	$\mathbf{id}_2$ $	shift
$ $T * \mathbf{id}_2$	$	reduce by $F \rightarrow \mathbf{id}$
$ $T * F$	$	reduce by $T \rightarrow T * F$
$ T	$	reduce by $E \rightarrow T$

Bottom Up Parsing

Example

STACK	INPUT	ACTION
$	$\mathbf{id}_1 * \mathbf{id}_2$ $	shift
$ $\mathbf{id}_1$	$* \mathbf{id}_2$ $	reduce by $F \to \mathbf{id}$
$ F	$* \mathbf{id}_2$ $	reduce by $T \to F$
$ T	$* \mathbf{id}_2$ $	shift
$ $T *$	$\mathbf{id}_2$ $	shift
$ $T * \mathbf{id}_2$	$	reduce by $F \to \mathbf{id}$
$ $T * F$	$	reduce by $T \to T * F$
$ T	$	reduce by $E \to T$
$ E	$	accept

Bottom Up Parsing

Example

- For the given grammar

  $E \rightarrow E + T \mid T$

  $T \rightarrow T * F \mid F$

  $F \rightarrow (E) \mid id$

- Derive the string – **id * id + id**

Bottom Up Parsing

STACK	INPUT	ACTION
$	id * id + id$	SHIFT

Bottom Up Parsing

STACK	INPUT	ACTION
$	id * id + id$	SHIFT
$id	* id + id$	REDUCE by ***F → id***

Bottom Up Parsing

STACK	INPUT	ACTION
$	id * id + id$	SHIFT
$id	* id + id$	REDUCE by ***F → id***
$F	* id + id$	REDUCE by ***T → F***

Bottom Up Parsing

STACK	INPUT	ACTION
$	id * id + id$	SHIFT
$id	* id + id$	REDUCE by ***F → id***
$F	* id + id$	REDUCE by ***T → F***
$T	* id + id$	SHIFT

Bottom Up Parsing

STACK	INPUT	ACTION
$	id * id + id$	SHIFT
$id	* id + id$	REDUCE by ***F → id***
$F	* id + id$	REDUCE by ***T → F***
$T	* id + id$	SHIFT
$T *	id + id$	SHIFT

Bottom Up Parsing

STACK	INPUT	ACTION
$	id * id + id$	SHIFT
$id	* id + id$	REDUCE by ***F → id***
$F	* id + id$	REDUCE by ***T → F***
$T	* id + id$	SHIFT
$T *	id + id$	SHIFT
$T * id	+ id$	REDUCE by ***F → id***

Bottom Up Parsing

STACK	INPUT	ACTION
$	id * id + id$	SHIFT
$id	* id + id$	REDUCE by ***F → id***
$F	* id + id$	REDUCE by ***T → F***
$T	* id + id$	SHIFT
$T *	id + id$	SHIFT
$T * id	+ id$	REDUCE by ***F → id***
$T * F	+ id$	REDUCE by ***T → T * F***

Bottom Up Parsing

STACK	INPUT	ACTION
$	id * id + id$	SHIFT
$id	* id + id$	REDUCE by ***F → id***
$F	* id + id$	REDUCE by ***T → F***
$T	* id + id$	SHIFT
$T *	id + id$	SHIFT
$T * id	+ id$	REDUCE by ***F → id***
$T * F	+ id$	REDUCE by ***T → T * F***
$T	+ id$	REDUCE by ***E → T***

Bottom Up Parsing

STACK	INPUT	ACTION
$	id * id + id$	SHIFT
$id	* id + id$	REDUCE by ***F → id***
$F	* id + id$	REDUCE by ***T → F***
$T	* id + id$	SHIFT
$T *	id + id$	SHIFT
$T * id	+ id$	REDUCE by ***F → id***
$T * F	+ id$	REDUCE by ***T → T * F***
$T	+ id$	REDUCE by ***E → T***
$E	+ id$	SHIFT

Bottom Up Parsing

STACK	INPUT	ACTION
$	id * id + id$	SHIFT
$id	* id + id$	REDUCE by ***F → id***
$F	* id + id$	REDUCE by ***T → F***
$T	* id + id$	SHIFT
$T *	id + id$	SHIFT
$T * id	+ id$	REDUCE by ***F → id***
$T * F	+ id$	REDUCE by ***T → T * F***
$T	+ id$	REDUCE by ***E → T***
$E	+ id$	SHIFT
$E +	id$	SHIFT

Bottom Up Parsing

STACK	INPUT	ACTION
$	id * id + id$	SHIFT
$id	* id + id$	REDUCE by ***F → id***
$F	* id + id$	REDUCE by ***T → F***
$T	* id + id$	SHIFT
$T *	id + id$	SHIFT
$T * id	+ id$	REDUCE by ***F → id***
$T * F	+ id$	REDUCE by ***T → T * F***
$T	+ id$	REDUCE by ***E → T***
$E	+ id$	SHIFT
$E +	id$	SHIFT
$E + id	$	REDUCE by ***F → id***

Bottom Up Parsing

STACK	INPUT	ACTION
$	id * id + id$	SHIFT
$id	* id + id$	REDUCE by ***F → id***
$F	* id + id$	REDUCE by ***T → F***
$T	* id + id$	SHIFT
$T *	id + id$	SHIFT
$T * id	+ id$	REDUCE by ***F → id***
$T * F	+ id$	REDUCE by ***T → T * F***
$T	+ id$	REDUCE by ***E → T***
$E	+ id$	SHIFT
$E +	id$	SHIFT
$E + id	$	REDUCE by ***F → id***
$E + F	$	REDUCE by ***T → F***

Bottom Up Parsing

STACK	INPUT	ACTION
$	id * id + id$	SHIFT
$id	* id + id$	REDUCE by ***F → id***
$F	* id + id$	REDUCE by ***T → F***
$T	* id + id$	SHIFT
$T *	id + id$	SHIFT
$T * id	+ id$	REDUCE by ***F → id***
$T * F	+ id$	REDUCE by ***T → T * F***
$T	+ id$	REDUCE by ***E → T***
$E	+ id$	SHIFT
$E +	id$	SHIFT
$E + id	$	REDUCE by ***F → id***
$E + F	$	REDUCE by ***T → F***
$E + T	$	REDUCE by ***E → E + T***

Bottom Up Parsing

STACK	INPUT	ACTION
$	id * id + id$	SHIFT
$id	* id + id$	REDUCE by ***F → id***
$F	* id + id$	REDUCE by ***T → F***
$T	* id + id$	SHIFT
$T *	id + id$	SHIFT
$T * id	+ id$	REDUCE by ***F → id***
$T * F	+ id$	REDUCE by ***T → T * F***
$T	+ id$	REDUCE by ***E → T***
$E	+ id$	SHIFT
$E +	id$	SHIFT
$E + id	$	REDUCE by ***F → id***
$E + F	$	REDUCE by ***T → F***
$E + T	$	REDUCE by ***E → E + T***
$E	$	ACCEPT

Bottom Up Parsing - LR Parsers

- Table driven grammars
- Can be constructed to recognize virtually all programming languages for which CFG(context free grammar) can be written
- Most general non-backtracking shift-reduce parsing method
- Types
  - Simple LR (SLR)
  - Look-Ahead LR (LALR)
  - Canonical LR

Bottom Up Parsing - LR Parsers

LR(0)

- It is a collection of sets of LR(0) items
- It provides basis for constructing a Deterministic Finite Automaton that is used to make parsing decisions
- To define LR(0) collection
  - Define augmented grammar
  - Define functions – CLOSURE and GOTO

Bottom Up Parsing - LR Parsers

LR(0)

- Defining Augmented Grammar
  - If G is a grammar with start symbol S, then G' is the augmented grammar of G
  - For eg. G' has a new start symbol S' and production S'→S
  - The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input
  - Acceptance occurs when and only when the parser is about to reduce by S'→S

Bottom Up Parsing - LR Parsers

Example

- Given Grammar

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

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

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

- Step 1 : Augment Grammar

$$\mathbf{E' \rightarrow E}$$

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

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

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

Bottom Up Parsing - LR Parsers

LR(0)

- CLOSURE
  - If I is a set of items for a grammar G, then CLOSURE(I) is the set of items constructed from I by the rules –

1. Initially add every item in I to CLOSURE(I)
2. If A→α.Bβ is in CLOSURE(I) and B→ is a production, then add item B→. to CLOSURE(I). Apply this rule till no more new items can be added to CLOSURE(I).

Bottom Up Parsing - LR Parsers

Example

I_0= E'→ .E
E → .E + T
E → .T
T → .T * F
T → .F
F → .(E)
F → .id

I_1= E'→ E.
E → E. + T

I_2= E → T.
T → T. * F

I_3= T → F.

I_4= F → (. E)
E → .E + T
E → .T
T → .T * F
T → .F
F → .(E)
F → .id

I_5= F → id.

I_6= E → E +. T
T → .T * F
T → .F
F → .(E)
F → .id

I_7= T → T *. F
F → .(E)
F → .id

I_8= F → (E.)
E → E. + T

I_9= E → E + T.
T → T. * F

I_{10}= T → T * F.

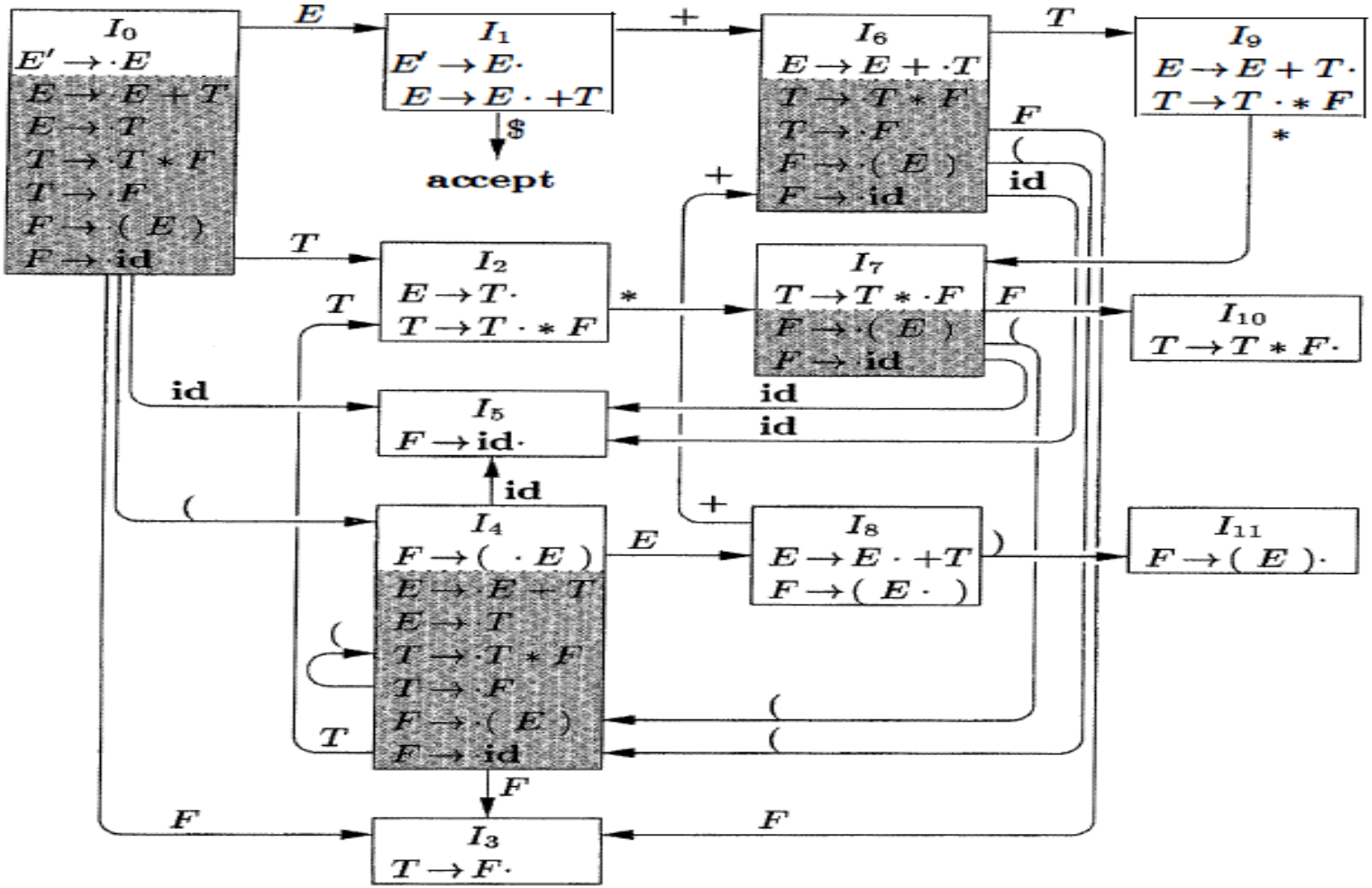
I_{11}= F → (E).

Bottom Up Parsing - LR Parsers

LR(0)

- GOTO
  - Written as GOTO(I,X) where I is a set of items and X is a grammar symbol
  - GOTO(I,X) is defined to be closure of the set of all items [A→αX.β] such that [A→α.Xβ] is in I.
  - It is used to define the transitions in LR(0) automaton for a grammar

Bottom Up Parsing - LR Parsers



Bottom Up Parsing - LR Parsers

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		**id** $*$ **id** $+$ **id** \$	shift
(2)	0 5	**id**	$*$ **id** $+$ **id** \$	reduce by $F \rightarrow$ **id**
(3)	0 3	F	$*$ **id** $+$ **id** \$	reduce by $T \rightarrow F$
(4)	0 2	T	$*$ **id** $+$ **id** \$	shift
(5)	0 2 7	$T*$	**id** $+$ **id** \$	shift
(6)	0 2 7 5	$T*$ **id**	$+$ **id** \$	reduce by $F \rightarrow$ **id**
(7)	0 2 7 10	$T*F$	$+$ **id** \$	reduce by $T \rightarrow T*F$
(8)	0 2	T	$+$ **id** \$	reduce by $E \rightarrow T$
(9)	0 1	E	$+$ **id** \$	shift
(10)	0 1 6	$E+$	**id** \$	shift
(11)	0 1 6 5	$E+$ **id**	\$	reduce by $F \rightarrow$ **id**
(12)	0 1 6 3	$E+F$	\$	reduce by $T \rightarrow F$
(13)	0 1 6 9	$E+T$	\$	reduce by $E \rightarrow E+T$
(14)	0 1	E	\$	accept

Bottom Up Parsing - LR Parsers

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		**id** $*$ **id** $+$ **id** \$	shift
(2)	0 5	**id**	$*$ **id** $+$ **id** \$	reduce by $F \rightarrow$ **id**
(3)	0 3	F	$*$ **id** $+$ **id** \$	reduce by $T \rightarrow F$
(4)	0 2	T	$*$ **id** $+$ **id** \$	shift
(5)	0 2 7	$T*$	**id** $+$ **id** \$	shift
(6)	0 2 7 5	$T*$ **id**	$+$ **id** \$	reduce by $F \rightarrow$ **id**
(7)	0 2 7 10	$T*F$	$+$ **id** \$	reduce by $T \rightarrow T*F$
(8)	0 2	T	$+$ **id** \$	reduce by $E \rightarrow T$
(9)	0 1	E	$+$ **id** \$	shift
(10)	0 1 6	$E+$	**id** \$	shift
(11)	0 1 6 5	$E+$ **id**	\$	reduce by $F \rightarrow$ **id**
(12)	0 1 6 3	$E+F$	\$	reduce by $T \rightarrow F$
(13)	0 1 6 9	$E+T$	\$	reduce by $E \rightarrow E+T$
(14)	0 1	E	\$	accept

Bottom Up Parsing - SLR(1)

- Usually (1) is omitted and grammar having SLR(1) parsing table is said to be SLR
- Table consisting of ACTION and GOTO is called SLR(1) parsing table for grammar G
- SLR cannot accept ambiguous grammars

Bottom Up Parsing - SLR(1)

Example

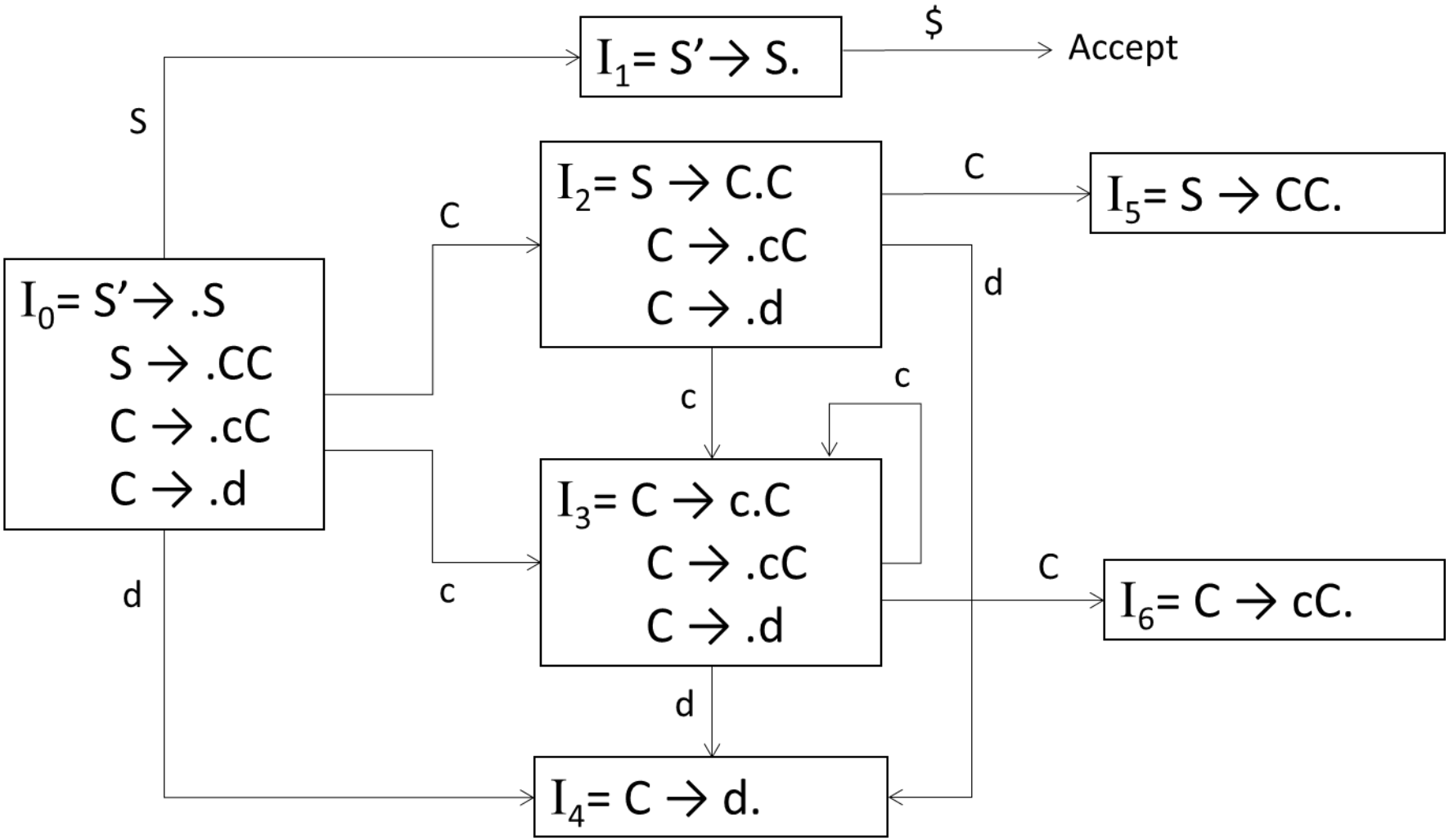
- Generate SLR parsing table for the following grammar –

$$S \rightarrow CC$$

$$C \rightarrow cC \mid d$$

- Derive the string **“ccdcd”**

Bottom Up Parsing - SLR(1)



Bottom Up Parsing - SLR(1)

SLR Parsing Table

States	c	d	$	S	C
0	s3	s4		1	2
1			Accept		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5			r2		
6	r1	r1	r1		

Bottom Up Parsing - SLR(1)

Stack	Symbol	Input	Action
0	$	ccdcd$	Shift
0 3	$ c	cdcd$	Shift
0 3 3	$ cc	dcd$	Shift
0 3 3 4	$ ccd	cd$	Reduce by C→d
0 3 3 6	$ ccC	cd$	Reduce by C→cC
0 3 6	$ cC	cd$	Reduce by C→cC
0 2	$ C	cd$	Shift
0 2 3	$ Cc	d$	Shift
0 2 3 4	$ Ccd	$	Reduce by C→d
0 2 3 6	$ CcC	$	Reduce by C→cC
0 2 5	$ CC	$	Reduce by S→CC
0 1	$ S	$	ACCEPT

More Powerful LR Parsers

- Canonical LR method –
  - Makes use of lookahead symbols.
  - Uses a large set of items called LR(1) items.

- Lookahead LR (LALR) –
  - Based on LR(0) sets of items.
  - Has fewer states than typical parsers based on LR(1) items.

Bottom Up Parsing-Operator Precedence Parser

- Used between operators to resolve ambiguity
- Used in numerous parsers because of simplicity
- Can be used for manipulation of tokens without any reference to a grammar
- Cannot handle tokens with multiple meaning.
  - Eg: (–) minus sign

Bottom Up Parsing-Operator Precedence Parser

- 3 disjoint relations
  - $<\cdot$
  - $\cdot>$
  - $\doteq$

- Example:

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

Bottom Up Parsing-Operator Precedence Parser

1. If operator $\theta 1$ has higher precedence than $\theta 2$, make $\theta 1 \cdot > \theta 2$

2. If operator $\theta 1$ and $\theta 2$ have equal precedence, make $\theta 1 \cdot > \theta 2$ and $\theta 2 > \cdot \theta 1$ if they are left associative (+ and –). OR

   make $\theta 1 < \cdot \theta 2$ and $\theta 2 < \cdot \theta 1$ if they are right associative ($\uparrow$ exponential sign)

Bottom Up Parsing-Operator Precedence Parser

1. ↑ is of highest precedence and is right associative

2. * and / are of next highest precedence and left associative

3. + and – are of lowest precedence and left associative

Bottom Up Parsing-Operator Precedence Parser

	+	-	*	/	^	id	(	)	$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	>	>	<	<	<	>	>
/	>	>	>	>	<	<	<	>	>
^	>	>	>	>	<	<	<	>	>
id	>	>	>	>	>			>	>
(	<	<	<	<	<	<	<	=	
)	>	>	>	>	>			>	>
$	<	<	<	<	<	<	<		

Practice Question

- Design Operator Precedence Parsing table for the following grammar

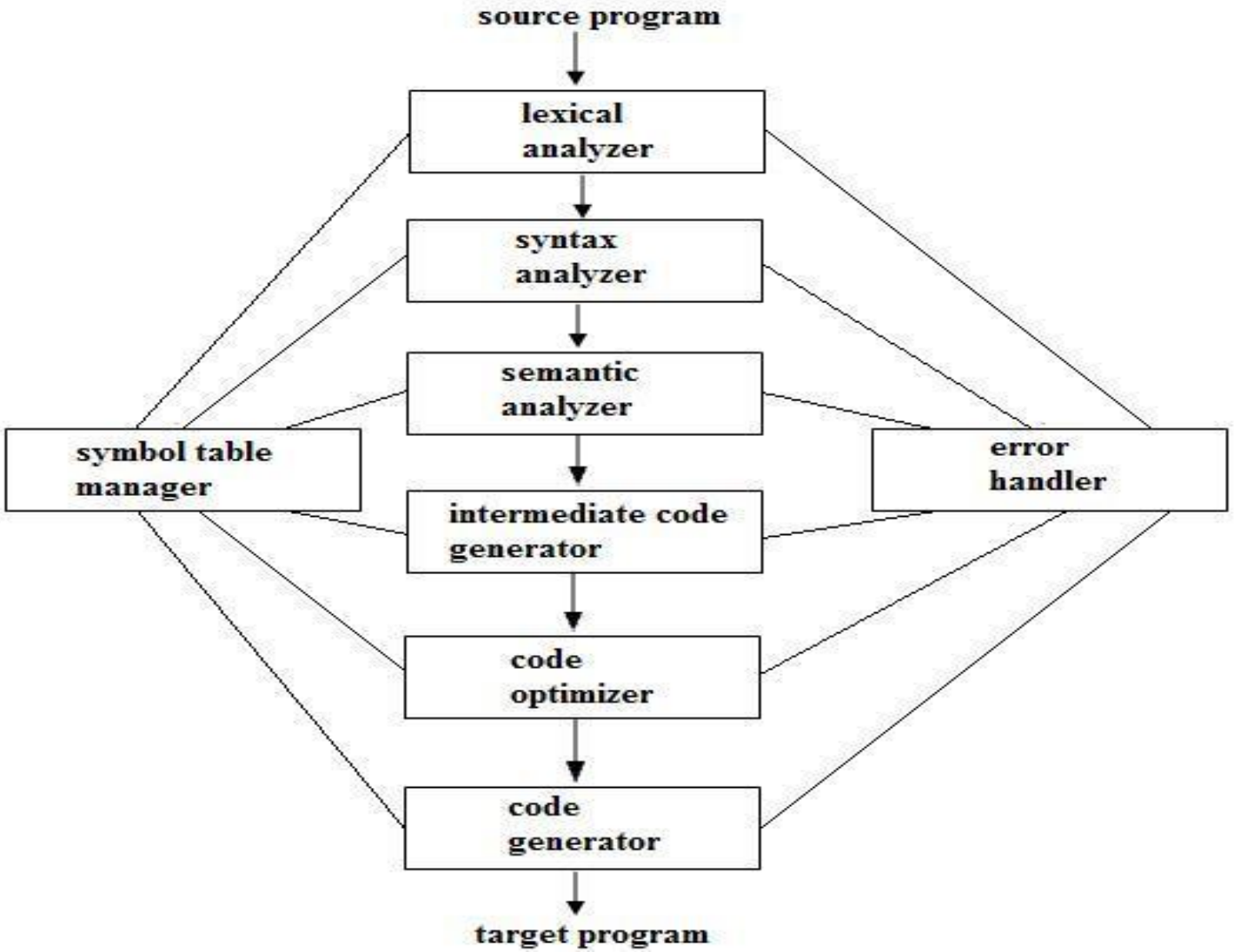
$$E \rightarrow E + T \mid T$$

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

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

Syntax Directed Translation

Syntax Directed Translation



Syntax Directed Translation

- Translation of languages is guided by CFGs
- The translated code will be used in Intermediate Code generation.
- SDT = Grammar + Semantic Rules
- Semantic rules are written in curly braces { }

Syntax Directed Translation

- 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.
- 2 types of attributes –
  - Synthesized attribute
  - Inherited attribute

Syntax Directed Translation

Types of Attributes

- Synthesized Attribute
  - For a non-terminal A, at a parse tree node N, synthesized attribute is defined by a semantic rule associated with production at N

- Inherited Attribute
  - For a non-terminal B at a parse tree node N, inherited attribute is defined by a semantic rule associated with the production at the parent node of N

Syntax Directed Translation

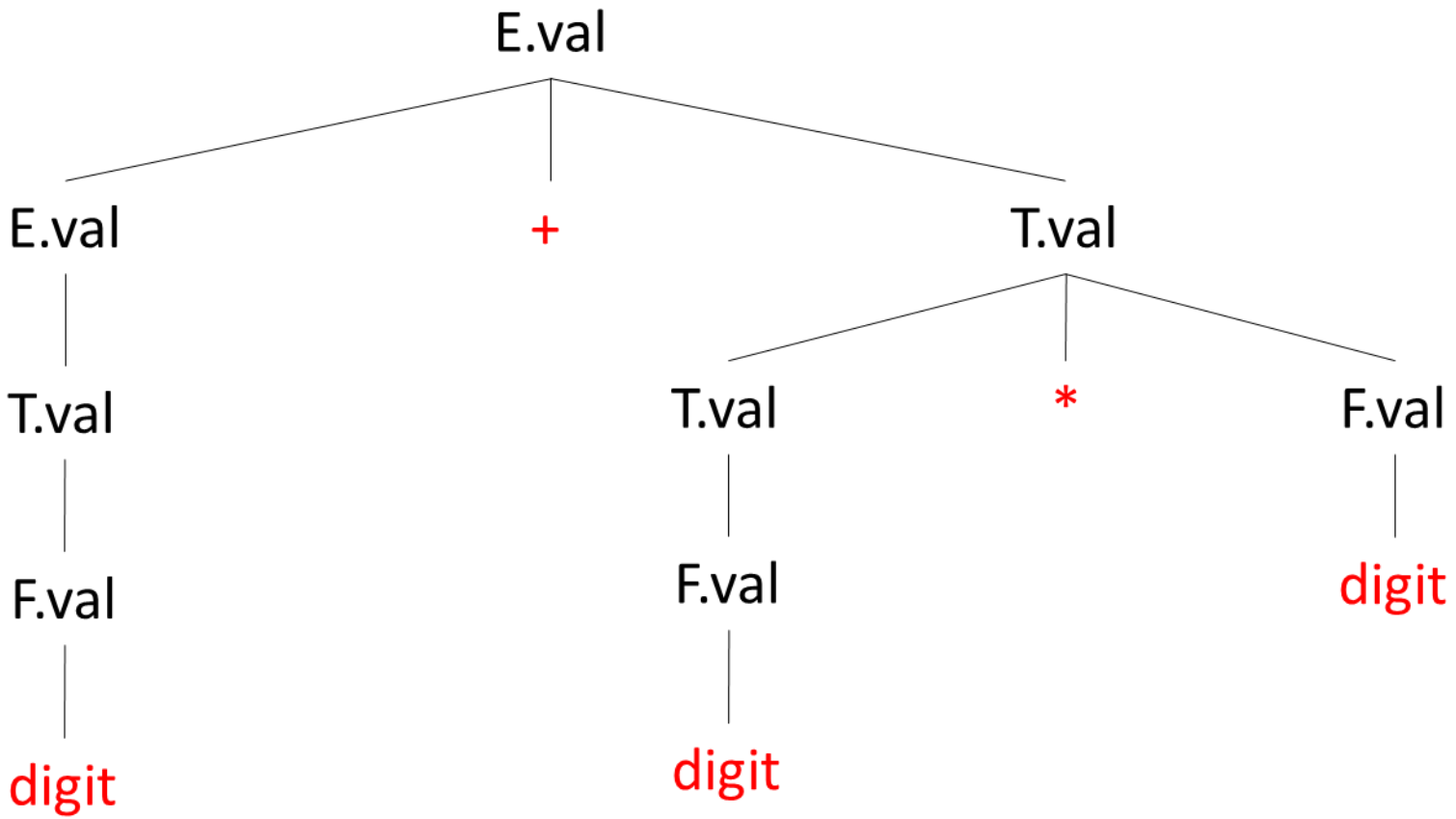
Example 1

$E \rightarrow E + T$	{E.val = E.val + T.val}
$E \rightarrow T$	{E.val = T.val}
$T \rightarrow T * F$	{T.val = T.val x F.val}
$T \rightarrow F$	{T.val = F.val}
$F \rightarrow (E)$	{F.val = E.val}
$F \rightarrow digit$	{F.val = digit.lexval}

- Evaluate **3+5*4**

Syntax Directed Translation

Example 1



Syntax Directed Translation

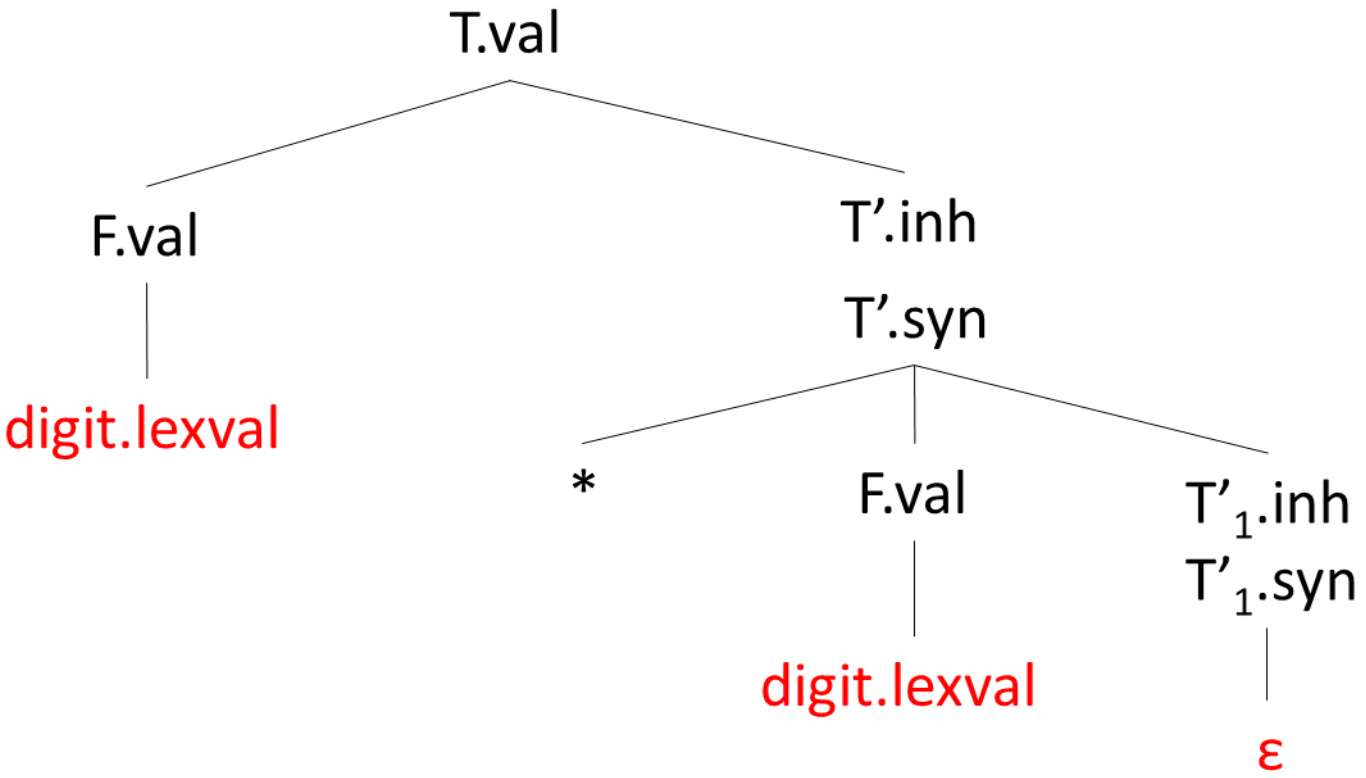
Example 2

$T \rightarrow FT'$	{T'.inh = F.val}
	{T.val = T'.syn}
$T' \rightarrow *FT'_1$	{T'_1.inh = T'.inh x F.val}
	{T'.syn = T'_1.syn}
$T' \rightarrow \varepsilon$	{T'.syn = T'.inh}
$F \rightarrow$ digit	{F.val = digit.lexval}

- Evaluate **3*5**

Syntax Directed Translation

Example 2

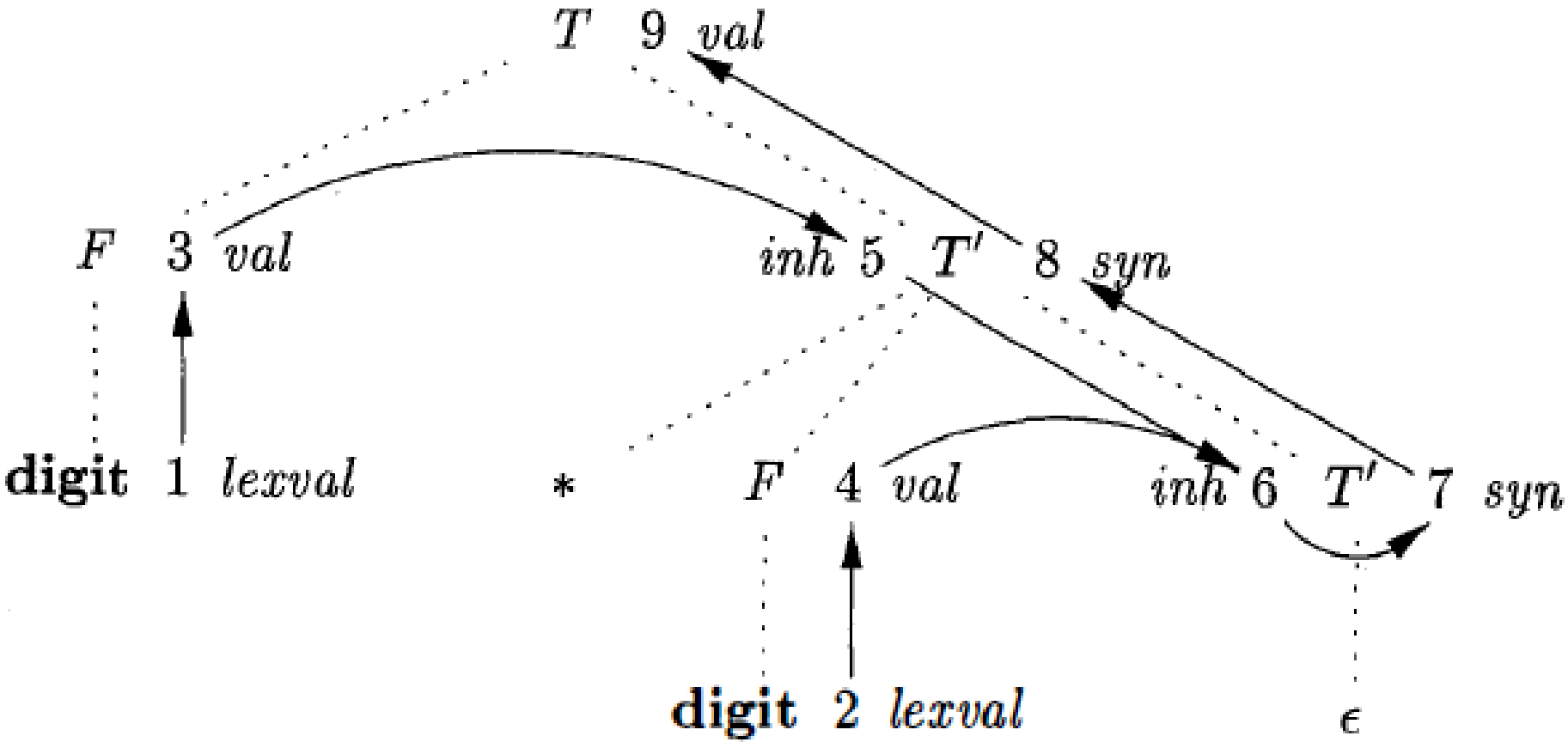


Dependency Graphs

- A dependency graph depicts 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.

Syntax Directed Translation

Dependency Graphs



Ordering Evaluation of Attributes

- If the dependency graph has an edge from node M to node N, then the attribute corresponding to M must be evaluated before the attribute of N.

- For a sequence of nodes $N_1, N_2, \ldots N_k$,If there is an edge of the dependency graph from N_i to N_j ($i < j$), then such an ordering 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.

S-attributed Definitions

- An SDD is S-attributed if every attribute is synthesized.
- Each rule computes an attribute for the nonterminal at the head of a production from attributes taken from body of the production.
- An S-attributed SDD can be implemented with an LR parser.

S-attributed Definitions

- When an SDD is S-attributed, its attributes can be evaluated in any bottom-up order of the nodes of the parse tree.
- When traversing leaves of a node N for the last time, evaluate attributes in postorder traversal.
- Postorder traversal corresponds exactly to the order in which an LR parser reduces a production.

Syntax Directed Translation

L-attributed SDD

- Each attribute must be
  - Synthesized

    OR
  - Inherited with limited rules

    Suppose there is a production $A \rightarrow X_1 X_2 \ldots X_n$

    and there is an inherited attribute $X_i.a$ computed by a rule associated with this production

L-attributed Definition

- Then the rule may only use
  a) Inherited attributes associated with A
  b) Either inherited or synthesized attributes associated with the occurrences of symbols $X_1X_2…X_{i-1}$ located to left of X_i
  c) Inherited or synthesized attributes associated with occurrence of X_i itself but without cycles in dependency graph formed by attributes of X_i.

THE END!

Have a nice day!