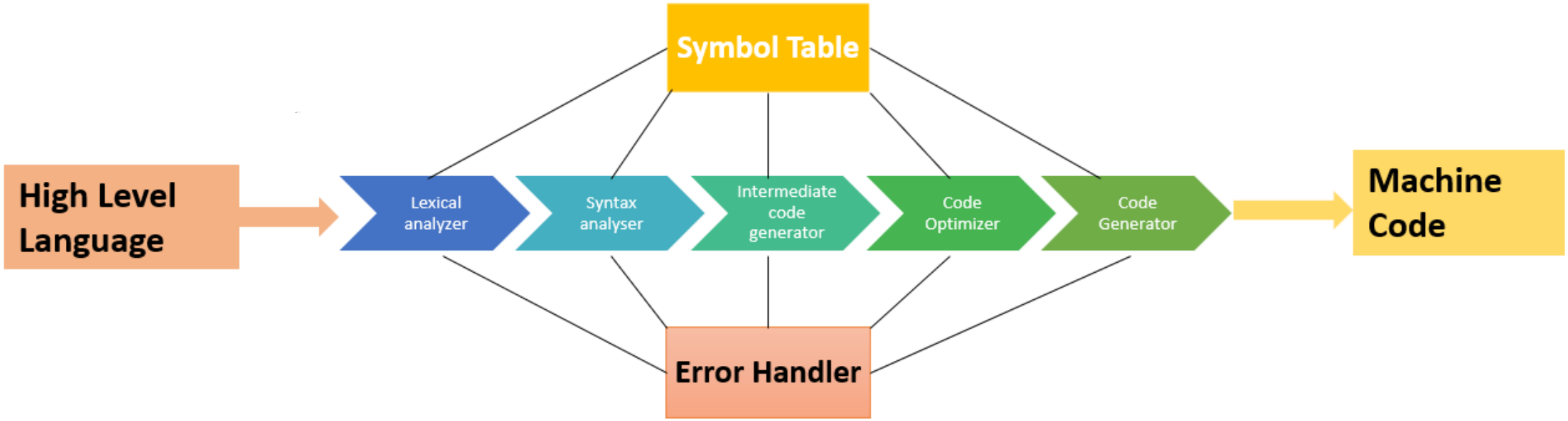

Lexical Analysis
Group 15

Compiler: An introduction

- A Compiler is a software that typically takes a high-level language (Like C++ and Java) code as input and converts the input to a lower-level language at once. It lists all the errors if the input code does not follow the rules of its language.

Phases of a Compiler



Lexical Phase

- It is the first phase of the compiler.
- It has 2 parts: Scanning and Lexical Analysis.
- **Scanning:** Deletion of comments, and compaction of consecutive white space characters into one
- **Lexical Analysis:** Complex portion, to produce tokens from the output of the scanner.

Overview of Lexical Analysis

- Input: Text (Code)
- Output: Sequence of Tokens

- In this phase, include files and macros are handled, line numbers are counted, white spaces are removed, illegal symbols are reported and the symbol table is created

Lexical Phase

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

Types of Tokens

- Keywords
- Operators
- Strings
- Constants
- Special Characters
- Identifiers

Steps in Lexical Analysis

- **Input Preprocessing:** This stage involves cleaning up the input text and preparing it for lexical analysis. This may include **removing comments, whitespace, and other non-essential characters** from the input text.

- **Tokenization:** This is the process of breaking the input text into a sequence of tokens. This is usually done by matching the characters in the input text against a set of **patterns** or **regular expressions** that define the different types of tokens.

Steps in Lexical Analysis

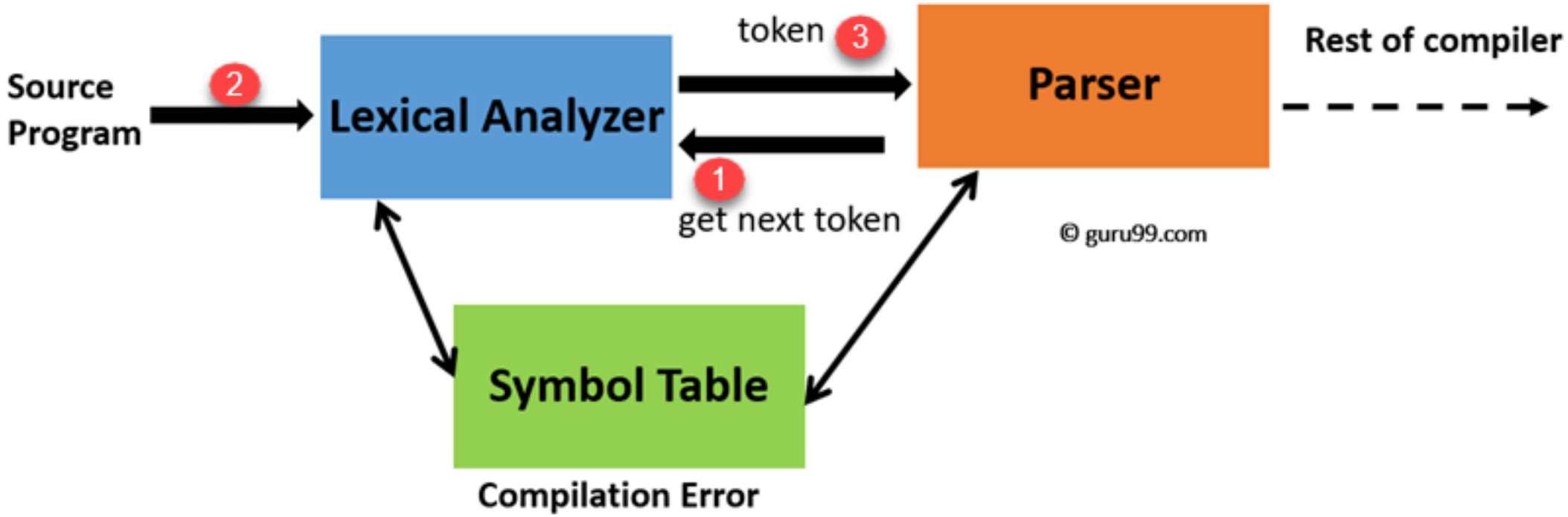
- **Token classification**: In this stage, the **lexer determines the type of each token**. For example, in a programming language, the lexer might classify keywords, identifiers, operators, and punctuation symbols as separate token types.

- **Token validation**: In this stage, the **lexer checks that each token is valid** according to the rules of the programming language. For example, it might check that a variable name is a valid identifier, or that an operator has the correct syntax.

Steps in Lexical Analysis

- **Output generation**: In this final stage, the lexer generates the output of the lexical analysis process, which is typically a **list of tokens**. This list of tokens can then be passed to the next stage of compilation or interpretation.

Interaction of Lexical analyzer with parser



GATE Question

The number of tokens in the following C statement is:

```
printf("i = %d, &i = %x", i, &i);
```

A) 3

B) 26

C) 10

D) 21

Answer

C) 10 tokens

```
printf
(
"i = %d, &i = %x"
,
i
,
&
i
)
;
```

Issues faced during Lexical Analysis

- As we know, in Lexical Analysis, the lexer splits the text and generates tokens but there might be cases where just considering the current character might produce ambiguous results.
- For Example, a > b and a >= b. The lexer might see the first character '>' and classify it as greater than instead of greater than equal to.
- This can be resolved by selecting the longest matching token.

Lexical Errors

- It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error
- Ex: fi (a == f(x)) . . .
- Simplest recovery strategy is “panic mode" recovery
- Other possible error-recovery actions are:
  - Delete one character from the remaining input.
  - Insert a missing character into the remaining input.
  - Replace a character by another character.
  - Transpose two adjacent characters

GATE Question

- Which one of the following languages over the alphabet {0,1} is described by the regular expression?
  (0+1)*0(0+1)*0(0+1)*

(A) The set of all strings containing the substring 00.
(B) The set of all strings containing at most two 0's.
(C) The set of all strings containing at least two 0's.
(D) The set of all strings that begin and end with either 0 or 1.

Answer

- C) is the correct answer

Explanation:

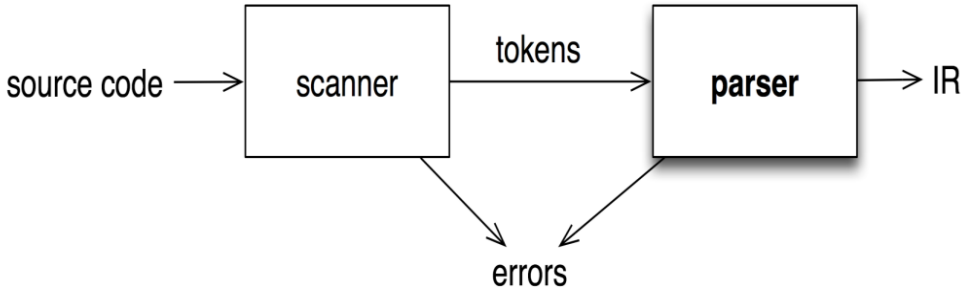
Option A says that it must have substring 00. But 10101 is also a part of language but it does not contain 00 as substring. So it is not correct option.
Option B says that it can have maximum two 0's but 00000 is also a part of language. So it is not correct option.

Option C says that it must contain atleast two 0. In regular expression, two 0 are present. So this is correct option.
Option D says that it contains all strings that begin and end with either 0 or 1. But it can generate strings which start with 0 and end with 1 or vice versa as well. So it is not correct.


Parsers

Functions of the Parser

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



Genaral Types of Parsers

- Universal Parser(Can Parse any grammar but inefficient. Used in NLP)
  - Cocke- Younger-Kasami
  - Earley's Algorithm
- Top-Down Parsers (Parse Tree built from top to bottom)
- Bottom Up Parsers

Example

- Which one of the following is a top-down parser

(A) Recursive descent parser.

(B) Operator precedence parser.

(C) An LR(k) parser.

(D) An LALR(k) parser

Answer: (A)

Recursive Descent parsing is LL(1) parsing which top down parsing.

Bottom Up Parser

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

Shift Reduce Parser

- Simplest of the Bottom up Parsers
- Shift input symbols until a handle is found.
- Reduce the substring to the non-terminal on the LHS of the corresponding production.
- A shift-reduce parser has 4 actions:

1) Shift the next input symbol is shifted onto the stack
2) Reduce the handle that is at top of stack a) pop handle b) push appropriate LHS symbol
3) Accept and stop parsing & report success
4) Error recovery routine is called

Gate Previous Year Example

- A shift reduce parser carries out the actions specified within braces immediately after reducing with the corresponding rule of grammar.

$S \rightarrow xxW \ \{print\,'1'\}$

$S \rightarrow y \ \{print\,'2'\}$

$W \rightarrow Sz \ \{print\,'3'\}$

What is the translation of **xxxxyzz** using the syntax directed translation scheme described by the above rules?

a) 23131 b) 11233 c) 11231 d) 33211

Answer: a) 23131

Context Free Grammars - CFG

- Programming language constructs are defined using context free grammar
- For example E→ E + E | E * E | (E) | id Expression grammar involving the operators, +, *, ( )
- Defined formally as (V, T, P, S)

V – Variables / Non-terminals

T – Terminals that constitute the string

P – Set of Productions that has a LHS and RHS

S – Special Symbol, subset of V

Previous year GATE question

S -> XY

X -> aX | a

Y -> aYb | null

Identify language generated by following grammar

A) $\{a^m b^n \mid m \geq n, n > 0\}$

B) $\{a^m b^n \mid m \geq n, n \geq 0\}$

C) $\{a^m b^n \mid m > n, n \geq 0\}$

D) $\{a^m b^n \mid m > n, n > 0\}$

Prerequisites for Top Down Parsing

- **Eliminate Left Recursion**

  - Rewrite every left-recursive production

    $A \rightarrow A\,\alpha \;/\; \beta \mid \gamma \mid A\,\delta$

  - into a right-recursive production:

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

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

Prerequisites for Top Down Parsing

- Replace productions

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

with

$$A \rightarrow \alpha\,A_R \mid \gamma$$

$$A_R \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

First and Follow

- FIRST function is computed for all terminals and non-terminals
- FIRST(a) = the set of terminals that begin all strings derived from a

- FOLLOW(A) = the set of terminals that can immediately follow non-terminal A

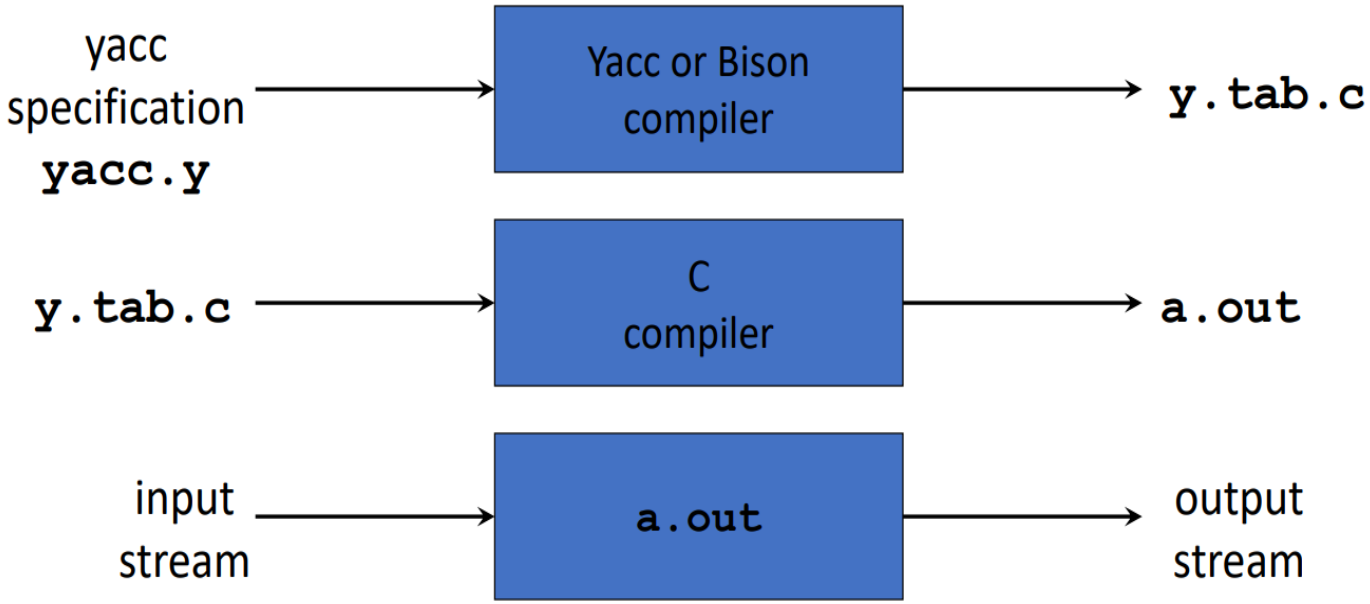
Previous year GATE question

- Consider the following grammar G:
- S → tABCD
- A → qt |t B → r | ε
- C → q | ε
- D → p
- What is the FOLLOW(A)?

1. {r, q, p, t } **2. {r, q, p }** 3. {ε, r, q, p } 4. {$, r, q, p }

Parser Generator

- There is enough integration between the lexer and the parser
- Lexer has already been implemented using a tool LEX
- Parser could also be done with a tool so as to help speed up



Simple LR parsing

- Easy to implement, but not powerful
- Uses LR(0) items

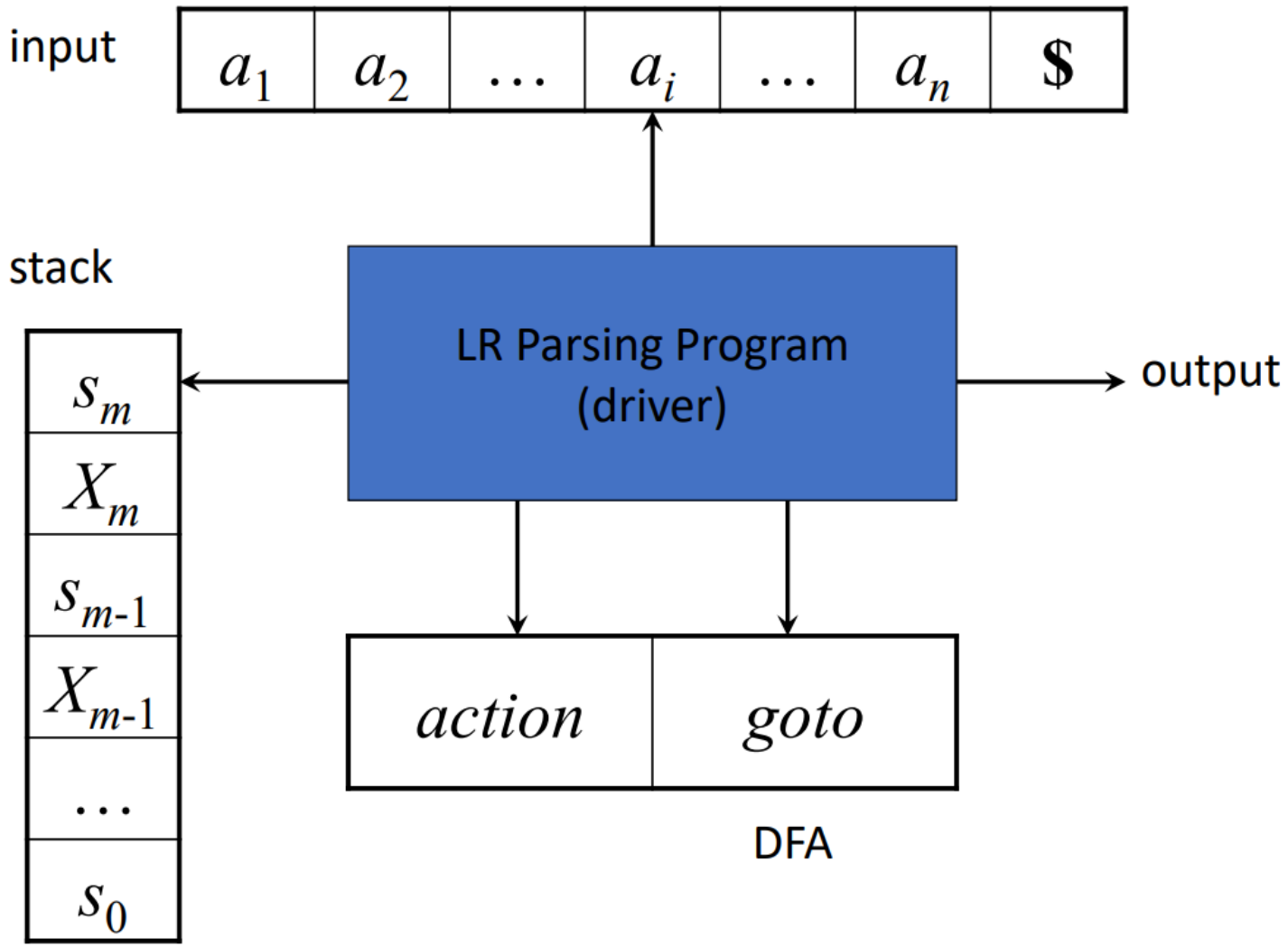
In **E-> E+E**

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

Our ultimate aim is to finally reach state E+E•, which corresponds to an actual handle yielding to the reduction E→E+E

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

Items that has the "." at the end leads to the reduction by that production


input
a_1 a_2 … a_i … a_n $
stack
s_m
X_m
s_{m-1}
X_{m-1}
…
s_0
LR Parsing Program
(driver)
output
action
goto
DFA

Steps

- Form the augmented grammar
- Construction of LR(0) items
- Construct the follow() for all the non-terminals which requires construction of first() for all the terminals and nonterminals
- Using this and the follow( ) of the grammar, construct the parsing table
- Using the parsing table, a stack and an input parse the input

LR (0) items

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

Thus, a production A → X Y Z has four items:

[A → • X Y Z] [A → X • Y Z] [A → X Y • Z] [A → X Y Z •]

Production A → ε has one item [A → •]

Example

Augmented Grammar

E’ → E

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

LR(0) Items

I_0

- E’ → .E
- E → . E +T
- E → .T
- T → .T * F
- T → .F
- F → . (E)
- F → . id

I1 = Goto (I0 , E)

- E’ → E.
- E → E . + T

I2 = Goto (I0 , T), Goto (I3 , T)

- E → T.
- T → T.*F

I3 = Goto (I0 , **(**), Goto (I3 , **(**))

- F → (.E)
- E → .E + T
- E → .T
- T → .T * F
- T → .F
- F → .(E)
- F → .id

Table creation

Input: Augmented Grammar G’

Output: SLR parsing table with functions, shift, reduce and accept

Parsing table is between items and Terminals and nonterminals

The non-terminals correspond to the goto() of the items set

The terminals have the parsing table corresponding to the action – shift / reduce/accept

- si means shift state i
- rj means reduce by production numbered j
- Blank means error

Follow

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

Augmented Grammar

E’ → E

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

Shift, Accept, Reduce

State	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	s5			s3			1	2	4
1		s6				accept			
2		r2	s10		r2	r2			
3	s5			s3			7	2	4
4		r4	r4		r4	r4			
5		r6	r6		r6	r6			
6	s5			s3				8	4

State	Action						Goto		
	id	+	*	(	)	$	E	T	F
7		s6			s9				
8		r1	s10		r1	r1			
9		r5	r5		r5	r5			
10	s5			s3					11
11		r3	r3		r3	r3			

Parsing Example

Stack	Input	Action
0	id * id $	[0, id] → s5 , shift
0 id 5	* id $	[5, *], r6, pop 2 symbols, Goto[0, F] = 4
0 F 4	* id $	[4, *], r4, pop 2 symbols, Goto[0, T] = 2
0 T 2	* id $	[2, *], → s10, shift

Drawbacks

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

Short Comes

In SLR, if there is a production of the form A → α▪ , then a reduce action takes place based on follow(A)

There would be situations, where when state i appears on the top of stack, the viable prefix βα on the stack is such that βA cannot be followed by terminal ‘a’ in a right sentential form

Hence, the reduction A → α would be invalid on input ‘a’

CALR parsing

- Construct LR(1) items
- Use these items to construct the CALR parsing table involving action and goto
- Use this table, along with input string and stack to parse the string
- Extra symbol is incorporated in the items to include a terminal symbol as a second component
- 1 – refers to the length of the second component – lookahead of the item

Example

- S → CC
- C → cC
- C → d

Augmented

- S’ → S
- S → CC
- C → cC
- C → d

LR(1) items

- I_0 :

S’ → .S, $

S → .CC, $

C → .cC, c/d (first(C$)

C → .d, c/d

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

LALR Parsing

- LR(1) parsing tables have many states
- LALR(1) parsing (Look-Ahead LR) combines LR(1) states to reduce table size
- Less powerful than LR(1)
- Does not have shift-reduce conflicts, because shifts do not use lookaheads
- May introduce reduce-reduce conflicts, but not much of a problem for grammars of programming languages

Merging

- I_3 : goto(I_0 , c), goto(I_3 , c),
  C → c.C, c/d
  C → .cC, c/d
  C → .d, c/d
- I_6 : goto(I_2 , c) goto(I_6 ,c)
  C → c.C, $
  C → .cC, $
  C → .d, $

- I_{36} : goto(I_0 , c), goto(I_{36} , c),
  C → c.C, c/d/$
  C → .cC, c/d/$
  C → .d, c/d/$

Previous year GATE question

Q. Consider the grammar shown below.

S → C C

C → c C | d

The grammar is
(A) LL(1)
(B) SLR(1) but not LL(1)
(C) LALR(1) but not SLR(1)
(D) LR(1) but not LALR(1)

Answer: **(A)**

Q. Consider the following two sets of LR(1) items of an LR(1) grammar.

```
X -> c.X, c/d
   X -> .cX, c/d
   X -> .d, c/d
   X -> c.X, $
   X -> .cX, $
   X -> .d, $
```

Which of the following statements related to merging of the two sets in the corresponding LALR parser is/are FALSE?

Cannot be merged since look aheads are different.

Can be merged but will result in S-R conflict.

Can be merged but will result in R-R conflict.

Cannot be merged since goto on c will lead to two different sets.

(A) 1 only
(B) 2 only
(C) 1 and 4 only
(D) 1, 2, 3, and 4

Answer: **(D)**

Q. Which of the following statements about the parser is/are correct?

I. Canonical LR is more powerful than SLR.

II. SLR is more powerful than LALR.

III. SLR is more powerful than canonical LR.

(A) I only
(B) II only
(C) III only
(D) II and III only

Answer: **(A)**


Syntax Directed
Translation

Syntax Directed Translation

Parser uses a CFG(Context-free-Grammar) to validate the input string and produce output for the next phase of the compiler. Output could be either a parse tree or an abstract syntax tree. Now to interleave semantic analysis with the syntax analysis phase of the compiler, we use Syntax Directed Translation.

The Need for SDT

Challenges and Solutions

- In traditional compiler design, the syntax analysis and code generation were done in separate passes. This resulted in a number of challenges, such as the need for multiple passes, the inability to handle context-sensitive grammar, and the difficulty in producing optimized code.

SDT Techniques

Inherited and Synthesized Attributes

- SDT uses two types of attributes: inherited and synthesized. Inherited attributes are passed up the parse tree from child nodes to parent nodes, while synthesized attributes are calculated from the attributes of the node itself and its children.

Inherited Attributes

Pass Up the Parse Tree

- Inherited attributes are used to propagate information up the parse tree. They are defined at the grammar level and are used to communicate information from child nodes to their parent nodes.

Synthesized Attributes in SDT

Calculate Values

- Synthesized attributes are used to calculate values based on the attributes of the node itself and its children. They are also defined at the grammar level and are used to communicate information from parent nodes to their child nodes.

SDT Applications

Semantic Analysis and Code Generation

- SDT is used in compiler design for two main purposes: semantic analysis and code generation. Semantic analysis involves the checking of the program's meaning and is performed during parsing. Code generation involves the translation of the parse tree into executable code.

Semantic Analysis

SDT and Type Checking

- SDT is used in semantic analysis to perform type checking, which involves checking that the types of operands in an expression match the expected types. This is done by attaching type information to the parse tree nodes using synthesized attributes.

Code Generation in Compiler Design

SDT and Intermediate Code Generation

- SDT is also used in code generation to generate intermediate code, which is an intermediate representation of the program that can be further optimized before being translated into machine code.

SDT and Intermediate Code Generation

- A typical example of SDT and intermediate code generation involves the translation of arithmetic expressions. The parse tree for an arithmetic expression is annotated with synthesized attributes that indicate the type of the expression and the intermediate code for the expression.

Implementation of SDT in Compiler Design

SDT Algorithms and Techniques

- The implementation of SDT in compiler design involves the use of algorithms and techniques to calculate the attributes of the parse tree nodes. Some of the algorithms and techniques used include bottom-up parsing, top-down parsing, and LR parsing.

Tools for Implementing SDT in Compiler Design

Yacc, Bison, and Other Tools

- Yet Another Compiler Compiler
- Bison, which is a parser generator that is compatible with Yacc but offers additional features such as support for C++ and Java.
- ANTLR (ANother Tool for Language Recognition), which is a powerful parser generator that supports both LL and LR parsing

Advantages and Disadvantages of SDT in Compiler Design

Pros and Cons

- SDT offers several advantages in compiler design, including the ability to handle context-sensitive grammar, the ability to generate optimized code, and the simplification of the code generation process.
- However, there are also some disadvantages to using SDT, including the potential for increased memory usage and the increased complexity of the compiler design process.
- Overall, SDT is a powerful technique that can greatly improve the efficiency and effectiveness of compiler design. Its advantages make it a valuable tool for implementing compilers that can generate optimized and correct code.

GATE Question

Consider the following context-free grammar:

S → aSb | ε

where ε denotes the empty string.

The following SDT for evaluating an S-expression is given:

S → aSb { S.val = 2*S.val + 1; }

S → ε { S.val = 0; }

What is the value of S.val for the S-expression ababba?

(A) 0

(B) 25

(C) 31

(D) 63

GATE Question

Consider the following Syntax Directed Translation Scheme (SDTS), with non-terminals {S, A} and terminals {a, b}}.

$$
\begin{aligned}
S &\longrightarrow \mathbf{a}A \quad \{ \text{print } 1 \} \\
S &\longrightarrow \mathbf{a} \quad \{ \text{print } 2 \} \\
A &\longrightarrow S\mathbf{b} \quad \{ \text{print } 3 \}
\end{aligned}
$$

Using the above SDTS, the output printed by a bottom-up parser, for the input **aab** is

(A) 1 3 2
(B) 2 2 3
(C) 2 3 1
(D) Syntax Error

GATE Question

Correct Answer is C (2 3 1)

Bottom up parser does the parsing by RMD in reverse.

RMD is as follows:

=>S

=> aA (hence, print 1)

=> aSb (hence, print 3)

=> aab (hence, print 2)

If we take in Reverse it will print : 231