Chapter 1

Lexical Analysis and Parsing

LEARNING OBJECTIVES

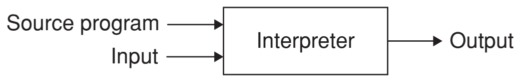
- *Language processing system*
- *Lexical analysis*
- *Syntax analysis*
- *Context free grammars and ambiguity*
- *Types of parsing*
- *Top down parsing*
- *Bottom up parsing*
- *Conflicts*
- *Operator precedence grammar*
- *LR parser*
- *Canonical LR parser(CLR)*

Language Processing System

Language Processors

Interpreter

It is a computer program that executes instructions written in a programming language. It either executes the source code directly or translates source code into some efficient intermediate representation and immediately executes this.



Example: Early versions of Lisp programming language, BASIC.

Translator

A software system that converts the source code from one form of the language to another form of language is called as translator. There are 2 types of translators namely (1) Compiler (2) Assembler.

Compiler converts source code of high level language into low level language.

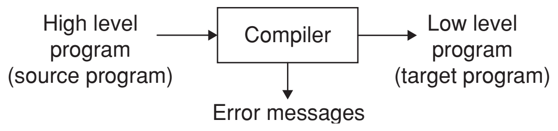
Assembler converts assembly language code into binary code.

Compilers

A compiler is a software that translates code written in high-level language (i.e., source language) into target language.

Example: source languages like C, Java,… etc. Compilers are user friendly.

The target language is like machine language, which is efficient for hardware.


High level program (source program)
Compiler
Low level program (target program)
Error messages

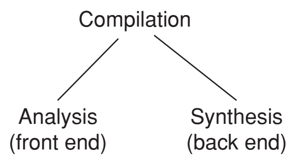
Passes

The number of iterations to scan the source code, till to get the executable code is called as a pass.

Compiler is two pass. Single pass requires more memory and multipass require less memory.

Analysis–synthesis model of compilation

There are two parts of compilation:

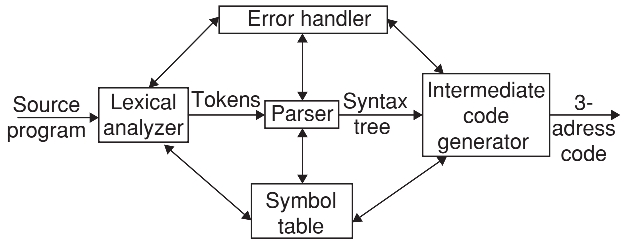


Analysis It breaks up the source program into pieces and creates an intermediate representation of the source program. This is more language specific.

Synthesis It constructs the desired target program from the intermediate representation. The target program will be more machine specific, dealing with registers and memory locations.

Front end vs back end of a compiler

The front end includes all analysis phases and intermediate code generator with part of code optimization.



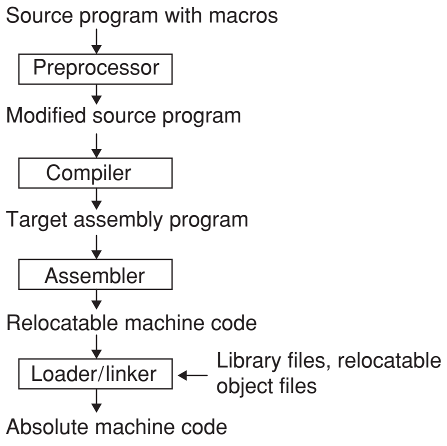
The back end includes code optimization and code generation phases. The back end synthesizes the target program from intermediate code.

Context of a compiler

In addition to a compiler, several other programs may be required to create an executable target program, like preprocessor to expand macros.

The target program created by a compiler may require further processing before it can be run.

The language processing system will be like this:



Phases

Compilation process is partitioned into some subproceses called phases.

In order to translate a high level code to a machine code, we need to go phase by phase, with each phase doing a particular task and parsing out its output for the next phase.

Lexical analysis or scanning

It is the first phase of a compiler. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes.

Example: Consider the statement: if $(a < b)$

In this sentence the tokens are if, $(a, <, b,)$.

Number of tokens = 6

Identifiers: a, b

Keywords: if

Operators: <, (,)

Syntax analyzer or Parser

- Tokens are grouped hierarchically into nested collections with collective meaning.
- A context free grammar (CFG) specifies the rules or productions for identifying constructs that are valid in a programming language. The output is a parse/syntax/derivation tree.

Example: Parse tree for –(id + id) using the following grammar:

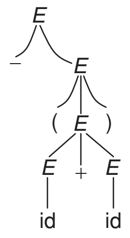
$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow -E \qquad (G_1)$

$E \rightarrow (E)$

$E \rightarrow id$



Semantic analysis

- It checks the source program for semantic errors.
- Type checking is done in this phase, where the compiler checks that each operator has matching operands for semantic consistency with the language definition.
- Gathers the type information for the next phases.

Example 1: The bicycle rides the boy.

This statement has no meaning, but it is syntactically correct.

Example 2:

```
int a;
bool b;
char c;
c = a + b;
```

We cannot add integer with a Boolean variable and assign it to a character variable.

Intermediate code generation

The intermediate representation should have two important properties:

(i) It should be easy to produce.

(ii) Easy to translate into the target program

'Three address code' is one of the common forms of Intermediate code.

Three address code consists of a sequence of instructions, each of which has at most three operands.

Example:

$id_1 = id_2 + id_3 \times 10;$

$t_1 := \text{inttoreal}(10)$

t_2:= id_3 × t_1

t_3:= id_2 + t_2

id_1 = t_3

Code optimization

The output of this phase will result in faster running machine code.

Example: For the above intermediate code the optimized code will be

t_1:= id_3 × 10.0

id_1: = id_2 + t1

In this we eliminated t_2 and t_3 registers.

Code generation

- In this phase, the target code is generated.
- Generally the target code can be either a relocatable machine code or an assembly code.
- Intermediate instructions are each translated into a sequence of machine instructions.
- Assignment of registers will also be done.

Example:

```
MOVF    id3, R2
MULF    ≠ 60.0, R2
MOVF    id2, R1
ADDF    R2, R1
MOVF    R1, id1
```

Symbol table management

A symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name.

What is the use of a symbol table?

1. To record the identifiers used in the source program.
2. Its type and scope
3. If it is a procedure name then the number of arguments, types of arguments, the method of parsing (by reference) and the type returned.

Error detection and reporting

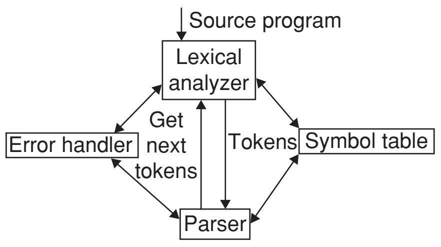
(i) Lexical phase can detect errors where the characters remaining in the input 'do not form any token'.
(ii) Errors of the type, 'violation of syntax' of the language are detected by syntax analysis.
(iii) Semantic phase tries to detect constructs that have the right syntactic structure but no meaning.

Example: adding two array names etc.

Lexical Analysis

Lexical Analysis is the first phase in compiler design. The main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis.

There will be interaction with the symbol table as well.



Lexeme: Sequence of characters in the source program that matches the pattern for a token. It is the smallest logical unit of a program.

Example: 10, *x*, *y*, <, >, =

Tokens: These are the classes of similar lexemes.

Example: Operators: <, >, =
Identifiers: *x*, *y*
Constants: 10
Keywords: if, else, int

Operations performed by lexical analyzer

1. Identification of lexemes and spelling check
2. Stripping out comments and white space (blank, new line, tab etc).
3. Correlating error messages generated by the compiler with the source program.
4. If the source program uses a macro-preprocessor, the expansion of macros may also be performed by lexical analyzer.

Example 1: Take the following example from Fortran
DO 5 I = 1.25
Number of tokens = 5
The 1st lexeme is the keyword DO
Tokens are DO, 5, I, =, 1.25.

Example 2: An example from C program
for (int *i* = 1; *i* < = 10; i + +)
Here tokens are for, (, int, *i*, =, 1,;, *i*, < =, 10,;, i, ++,)
Number of tokens = 13

LEX compiler

Lexical analyzer divides the source code into tokens. To implement lexical analyzer we have two techniques namely hand code and the other one is LEX tool.

LEX is an automated tool which specifies lexical analyzer, from the rules given by the regular expression.

These rules are also called as pattern recognizing rules.

Syntax Analysis

This is the 2nd phase of the compiler, checks the syntax and constructs the syntax/parse tree.

Input of parser is token and output is a parse/ syntax tree.

Constructing parse tree

Construction of derivation tree for a given input string by using the production of grammar is called parse tree.

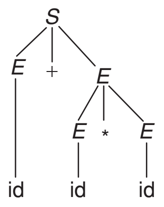
Consider the grammar

$S \rightarrow E + E / E * E$

$E \rightarrow id$

The parse tree for the string

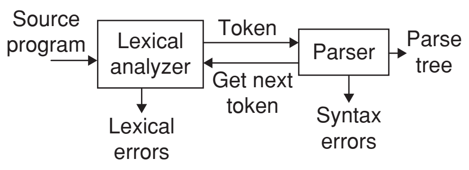
$\omega = id + id * id$ is



$\omega = id + id * id$

Role of the parser

1. Construct a parse tree.
2. Error reporting and correcting (or) recovery. A parser can be modeled by using CFG (Context Free Grammar) recognized by using pushdown automata/table driven parser.
3. CFG will only check the correctness of sentence with respect to syntax not the meaning.



How to construct a parse tree?

Parse tree's can be constructed in two ways.

(i) Top-down parser: It builds parse trees from the top (root) to the bottom (leaves).
(ii) Bottom-up parser: It starts from the leaves and works up to the root.

In both cases, the input to the parser is scanned from left to right, one symbol at a time.

Parser generator

Parser generator is a tool which creates a parser.

Example: compiler – compiler, YACC

The input of these parser generator is grammar we use and the output will be the parser code.

The parser generator is used for construction of the compilers front end.

Scope of declarations

Declaration scope refers to the certain program text portion, in which rules are defined by the language.

Within the defined scope, entity can access legally to declared entities.

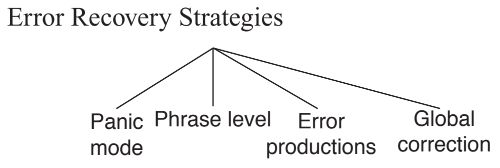
The scope of declaration contains immediate scope always. Immediate scope is a region of declarative portion with enclosure of declaration immediately.

Scope starts at the beginning of declaration and scope continues till the end of declaration. Whereas in the over loadable declaration, the immediate scope will begin, when the callable entity profile was determined.

The visible part refers text portion of declaration, which is visible from outside.

Syntax Error Handling

1. Reports the presence of errors clearly and accurately.
2. Recovers from each error quickly.
3. It should not slow down the processing of correct programs.



Panic mode On discovering an error, the parser discards input symbols one at a time until one of the synchronizing tokens is found.

Phrase level A parser may perform local correction on the remaining input. It may replace the prefix of the remaining input.

Error productions Parser can generate appropriate error messages to indicate the erroneous construct that has been recognized in the input.

Global corrections There are algorithms for choosing a minimal sequence of changes to obtain a globally least cost correction.

Context Free Grammars and Ambiguity

A grammar is a set of rules or productions which generates a collection of finite/infinite strings.

It is a 4-tuple defined as $G = (V, T, P, S)$

Where

V = set of variables
T = set of terminals
P = set of production rules
S = start symbol

Example: $S \to (S)/e$

$S \to (S)$ (1)

$S \to e$ (2)

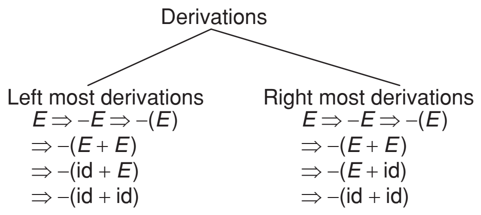
Here S is start symbol and the only variable.
(,), e is terminals.
(1) and (2) are production rules.

Sentential forms

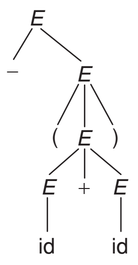
$s \overset{*}{\Rightarrow} \alpha$, Where α may contain non-terminals, then we say that α is a sentential form of G.

Sentence: A sentence is a sentential form with no non-terminals.

Example: $-(id + id)$ is a sentence of the grammar (G_1).


Derivations
Left most derivations
$E \Rightarrow -E \Rightarrow -(E)$
$\Rightarrow -(E + E)$
$\Rightarrow -(\text{id} + E)$
$\Rightarrow -(\text{id} + \text{id})$
Right most derivations
$E \Rightarrow -E \Rightarrow -(E)$
$\Rightarrow -(E + E)$
$\Rightarrow -(E + \text{id})$
$\Rightarrow -(\text{id} + \text{id})$

Right most derivations are also known as canonical derivations.



Ambiguity

A grammar that produces more than one parse tree for some sentence is said to be ambiguous.

Or

A grammar that produces more than one left most or more than one right most derivations is ambiguous.

For example consider the following grammar:

String $\to$ String + String/String – String /0/1/2/…/9

9 – 5 + 2 has two parse trees as shown below

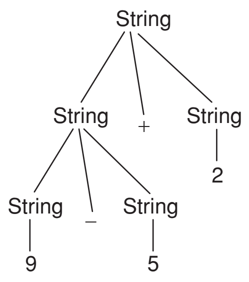


Figure 1 Leftmost derivation

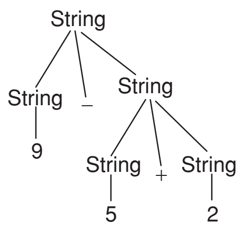


Figure 2 Rightmost derivation

- Ambiguity is problematic because the meaning of the program can be incorrect.
- Ambiguity can be handled in several ways
  1. Enforce associativity and precedence
  2. Rewrite the grammar by eliminating left recursion and left factoring.

Removal of ambiguity

The grammar is said to be ambiguous if there exists more than one derivation tree for the given input string.

The ambiguity of grammar is undecidable; ambiguity of a grammar can be eliminated by rewriting the grammar.

Example:

$E \to E + E/id\} \to$ ambiguous grammar

$\left.\begin{array}{l} E \to E + T/T \\ T \to id \end{array}\right\}$ rewritten grammar (unambiguous grammar)

Left recursion

Left recursion can take the parser into infinite loop so we need to remove left recursion.

Elimination of left recursion

$A \to A\alpha/\beta$ is a left recursive.

It can be replaced by a non-recursive grammar:

$$A \to \beta A'$$
$$A' \to \alpha A'/\varepsilon$$

In general

$$A \to A\alpha_1/A\alpha_2/\ldots/A\alpha_m/\beta_1/\beta_2/\ldots/\beta_n$$

We can replace A productions by

$$A \to \beta_1 A'/\beta_2 A'/\!-\!\beta_n A'$$
$$A' \to \alpha_1 A'/\alpha_2 A'/\!-\!\alpha_m A'$$

Example 3: Eliminate left recursion from

$E \to E + T/T$

$T \to T * F/F$

$F \to (E)/\text{id}$

Solution $E \to E + T/T$ it is in the form

$A \to A\alpha/\beta$

So, we can write it as $E \to TE'$

$E' \to +TE'/\varepsilon$

Similarly other productions are written as

$$T \to FT'$$
$$T^1 \to \times FT'/\in$$
$$F \to (E)/\text{id}$$

Example 4 Eliminate left recursion from the grammar

$$S \to (L)/a$$
$$L \to L, S/b$$

Solution: $S \to (L)/a$

$L \to bL'$

$L' \to SL'/\in$

Left factoring

A grammar with common prefixes is called non-deterministic grammar. To make it deterministic we need to remove common prefixes. This process is called as Left Factoring.

The grammar: $A \to \alpha\beta_1/\alpha\beta_2$ can be transformed into

$A \to \alpha A'$

$A' \to \beta_1/\beta_2$

Example 5: What is the resultant grammar after left factoring the following grammar?

$S \to iEtS/iEtSeS/a$

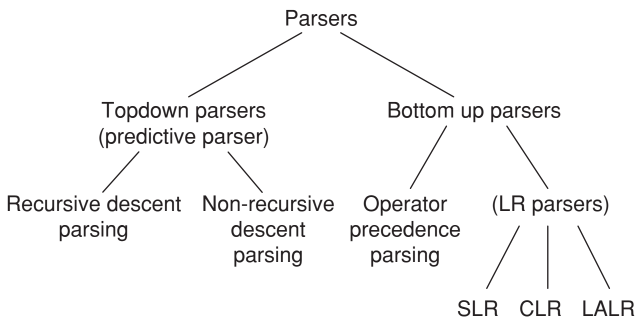
$E \to b$

Solution: $S \to iEtSS'/a$

$S' \to eS/\in$

$E \to b$

Types of Parsing


Parsers
Topdown parsers (predictive parser)
Bottom up parsers
Recursive descent parsing
Non-recursive descent parsing
Operator precedence parsing
(LR parsers)
SLR
CLR
LALR

Topdown Parsing

A parse tree is constructed for the input starting from the root and creating the nodes of the parse tree in preorder. It simulates the left most derivation.

Backtracking Parsing

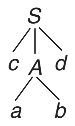
If we make a sequence of erroneous expansions and subsequently discover a mismatch we undo the effects and roll back the input pointer.

This method is also known as brute force parsing.

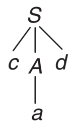
Example: $S \to cAd$

$A \to ab/a$

Let the string w = cad is to generate:



The string generated from the above parse tree is cabd.
but, w = cad, the third symbol is not matched.
So, report error and go back to A.
Now consider the other alternative for production A.



String generated 'cad' and $w = cad$. Now, it is successful.

In this we have used back tracking. It is costly and time consuming approach. Thus an outdated one.

Predictive Parsers

By eliminating left recursion and by left factoring the grammar, we can have parse tree without backtracking. To construct a predictive parser, we must know,

1. Current input symbol
2. Non-terminal which is to be expanded

A procedure is associated with each non-terminal of the grammar.

Recursive descent parsing

In recursive descent parsing, we execute a set of recursive procedures to process the input.

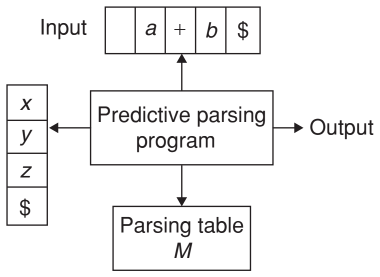
The sequence of procedures called implicitly, defines a parse tree for the input.

Non-recursive predictive parsing

(table driven parsing)

- It maintains a stack explicitly, rather than implicitly via recursive calls.
- A table driven predictive parser has

$\to$ An input buffer
$\to$ A stack
$\to$ A parsing table
$\to$ Output stream



Constructing a parsing table

To construct a parsing table, we have to learn about two functions:

1. FIRST ( )
2. FOLLOW ( )

FIRST(X) To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ε can be added to any FIRST set.

1. If X is a terminal, then FIRST(X) is $\{X\}$.
2. If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X).
3. If X is non-terminal and $X \rightarrow Y_1 Y_2 - Y_k$ is a production, then place 'a' in FIRST(X) if for some i, a is an FIRST (Y_i) and $\in$ is in all of FIRST(Y_1), …, FIRST(Y_{i-1}); that is, $Y_1, \ldots, Y_{i-1} \overset{*}{\Rightarrow} \in$. If $\in$ is in FIRST (Y_j) for all $j = 1, 2, \ldots, k$, then add $\in$ to FIRST(X). For example, everything in FIRST (Y_1) is surely in FIRST(X). If Y_1 does not derive $\in$, then add nothing more to FIRST(X), but if $Y_1 \overset{*}{\Rightarrow} \in$, then add FIRST ($Y_2$) and so on.

FOLLOW (A): To compute FOLLOW (A) for all non-terminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place \$ in FOLLOW(S), where S is the start symbol and \$ is input right end marker.
2. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST (β) except ε is placed in FOLLOW (B).
3. If there is a production $A \rightarrow \alpha B$ or a production $A \rightarrow \alpha B \beta$, where FIRST ($\beta$) contains ε, then everything in FOLLOW (A) is in FOLLOW (B).

Example: Consider the grammar

$E \rightarrow TE'$
$E' \rightarrow +TE'/\varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT'/\varepsilon$
$F \rightarrow (E)/id$. Then
FIRST (E) = FIRST (T) = FIRST (F) = {(, $id$}
FIRST (E') = {$^+$, ε}
FIRST (T') = {*, ε}
FOLLOW (E) = FOLLOW (E') = {), \$}
FOLLOW (T) = FOLLOW (T') = {+,), \$}
FOLLOW (F) = {*, +,), \$}

Steps for the construction of predictive parsing table

1. For each production $A \rightarrow \alpha$ of the grammar, do steps 2 and 3.
2. For each terminal a in FIRST (α), add $A \rightarrow \alpha$ to $M\,[A, a]$
3. If ε is in FIRST (α), add $A \rightarrow \alpha$ to $M\,[A, b]$ for each terminal b in FOLLOW (A). If ε is in FIRST (α) and \$ is in FOLLOW (A), add $A \rightarrow \alpha$ to $M\,[A, \$]$
4. Make each undefined entry of M be error.

By applying these rules to the above grammar, we will get the following parsing table.

	Input Symbol					
Non-terminal	**id**	**+**	*****	**(**	**)**	**\$**
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow$ id			$F \rightarrow (E)$		

The parser is controlled by a program. The program consider x, the symbol on top of the stack and 'a' the current input symbol.

1. If $x = a = \$$, the parser halts and announces successful completion of parsing.
2. If $x = a \neq \$$, the parser pops x off the stack and advances the input pointer to the next input symbol.
3. If x is a non-terminal, the program consults entry $M[x, a]$ of the parsing table M. This entry will be either an x-production of the grammar or an error entry. If $M[x, a] = \{x \rightarrow UVW\}$, the parser replaces x on top of the stack by WVU with U on the top.

If $M[x, a]$ = error, the parser calls an error recovery routine.

For example, consider the moves made by predictive parser on input id + id * id, which are shown below:

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′
id+	TE′\$	id*id\$	Match+
id+	FT′E′\$	id*id\$	Output T → FT′
id+	idT′E′\$	id*id\$	Output F → id
id+id	T′E′\$	*id\$	Match id
id+id	*FT′E′\$	*id\$	Output T′ → *FT′
id+id*	FT′E′\$	id\$	Match*
id+id*	idT′E′\$	id\$	OutputF → id
id+id*id	T′E′\$	\$	Match id
id+id*id	E′\$	\$	Output T′ → ε
id+id*id	\$	\$	Output E′ → ε

Bottom up Parsing

- This parsing constructs the parse tree for an input string beginning at the leaves and working up towards the root.
- General style of bottom-up parsing is shift-reduce parsing.

Shift–Reduce Parsing

Reduce a string to the start symbol of the grammar. It simulates the reverse of right most derivation.

In every step a particular substring is matched (in left right fashion) to the right side of some production and then it is substituted by the non-terminal in the left hand side of the production.

For example consider the grammar

$S \to aABe$

$A \to Abc/b$

$B \to d$

In bottomup parsing the string 'abbcde' is verified as

$$\left.\begin{array}{l} abbcde \\ aAbcde \\ aAde \\ aABe \\ S \end{array}\right\} \to \quad \text{reverse order}$$

Stack implementation of shift–reduce parser

The shift reduce parser consists of input buffer, Stack and parse table.

Input buffer consists of strings, with each cell containing only one input symbol.

Stack contains the grammar symbols, the grammar symbols are inserted using shift operation and they are reduced using reduce operation after obtaining handle from the collection of buffer symbols.

Parse table consists of 2 parts goto and action, which are constructed using terminal, non-terminals and compiler items.

Let us illustrate the above stack implementation.

$\to$ Let the grammar be

$S \to AA$

$A \to aA$

$A \to b$

Let the input string 'ω' be abab$

ω = abab$

Stack	Input String	Action
$	*abab*$	Shift
$*a*	*bab*$	Shift
$*ab*	*ab*$	Reduce ($A \to b$)
$*aA*	*ab*$	Reduce ($A \to aA$)
$*A*	*ab*$	Shift
$*Aa*	*b*$	Shift
$*Aab*	$	Reduce ($A \to b$)
$*AaA*	$	Reduce ($A \to aA$)
$*AA*	$	Reduce ($S \to AA$)
$*S*	$	Accept

Rightmost derivation

$S \Rightarrow aABe \Rightarrow aAde \Rightarrow aAbcde \Rightarrow abbcde$

For bottom up parsing, we are using right most derivation in reverse.

Handle of a string Substring that matches the RHS of some production and whose reduction to the non-terminal on the LHS is a step along the reverse of some rightmost derivation.

$$S \underset{*}{\overset{rm}{\Rightarrow}} \alpha Ar \Rightarrow \alpha\beta r$$

Right sentential forms of a unambiguous grammar have one unique handle.

Example: For grammar, $S \to aABe$

$A \to Abc/b$

$B \to d$

$S \Rightarrow \underline{aABe} \Rightarrow aA\underline{d}e \Rightarrow a\underline{Abc}de \Rightarrow a\underline{b}bcde$

Note: Handles are underlined.

Handle pruning The process of discovering a handle and reducing it to the appropriate left hand side is called handle pruning. Handle pruning forms the basis for a bottomup parsing.

To construct the rightmost derivation:

$$S = r_0 \Rightarrow r_1 \Rightarrow r_2 \underline{\quad\quad} \Rightarrow r_n = w$$

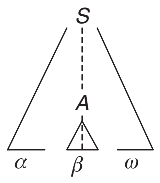
Apply the following simple algorithm:

For $i \leftarrow n$ to 1

Find the handle $A_i \to B_i$ in r_i

Replace B_i with A_i to generate r_{i-1}

Consider the cut of a parse tree of a certain right sentential form:



Here $A \to \beta$ is a handle for $\alpha\beta\omega$.

Shift reduce parsing with a stack There are 2 problems with this technique:

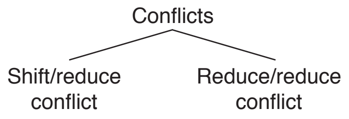
(i) To locate the handle
(ii) Decide which production to use

General construction using a stack

1. 'Shift' input symbols onto the stack until a handle is found on top of it.
2. 'Reduce' the handle to the corresponding non-terminal.
3. 'Accept' when the input is consumed and only the start symbol is on the stack.
4. Errors – call an error reporting/recovery routine.

Viable prefixes The set of prefixes of a right sentential form that can appear on the stack of a shift reduce parser are called viable prefixes.

Conflicts



Shift/reduce conflict

Example: stmt $\to$ if expr then stmt | if expr then stmt else stmt | any other statement

If exp then stmt is on the stack, in this case we can't tell whether it is a handle. i.e., 'shift/reduce' conflict.

Reduce/reduce conflict

Example: $S \to aA/bB$
$A \to c$
$B \to c$
W = ac it gives reduce/reduce conflict.

Operator Precedence Grammar

In operator grammar, no production rule can have:

- ε at the right side.
- two adjacent non-terminals at the right side.

Example 1: $E \to E + E/E - E/$ id is operator grammar.

Example 2: $\left.\begin{array}{l} E \to AB \\ A \to a \\ B \to b \end{array}\right\}$ not operator grammar

Example 3: $\underbrace{E \to E0E/\text{id}}_{\text{not operator grammar}}$

Precedence relation If
$a < b$ then b has higher precedence than a
$a = b$ then b has same precedence as a
$a > b$ then b has lower precedence than a

Common ways for determining the precedence relation between pair of terminals:

1. Traditional notations of associativity and precedence.

Example: $\times$ has higher precedence than $+ \times .> +$ (or) $+ <. \times$

2. First construct an unambiguous grammar for the language which reflects correct associativity and precedence in its parse tree.

Operator precedence relations from associativity and precedence

Let us use \$ to mark end of each string. Define $\$ <. b$ and $b > \$$ for all terminals b. Consider the grammar is:

$$E \to E + E/E \times E/\text{id}$$

Let the operator precedence table for this grammar is:

	id	+	×	\$
id		>	>	>
+	<	>	<	>
×	<	>	>	>
\$	<	<	<	accept

1. Scan the string from left until > is encountered
2. Then scan backwards (to left) over any = until < is encountered.
3. The handle contains everything to the left of the first > and to the right of the < is encountered.

After inserting precedence relation is

$$\$\text{id} + \text{id} * \text{id}\ \$ \text{ is}$$
$$\$ < \text{id} > + < \text{id} > * < \text{id} > \$$$

Precedence functions Instead of storing the entire table of precedence relations table, we can encode it by precedence functions f and g, which map terminal symbols to integers:

1. $f(a) < f(b)$ whenever $a < b$
2. $f(a) > f(b)$ whenever $a \doteq b$
3. $f(a) > f(b)$ whenever $a > b$

Finding precedence functions for a table

1. Create symbols $f(a)$ and $g(a)$ for each 'a' that is a terminal or \$.
2. Partition the created symbols into as many groups as possible in such away that $a = b$ then f (a) and g (b) are in the same group
3. Create a directed graph
   If $a < b$ then place an edge from $g(b)$ to $f(a)$
   If $a > b$ then place an edge from $f(a)$ to $g(b)$
4. If the graph constructed has a cycle then no precedence function exists.
   If there are no cycles, let $f(a)$ be the length of the longest path being at the group of $f(a)$.
   Let $g(a)$ be the length of the longest path from the group of $g(a)$.

Disadvantages of operator precedence parsing

- It can not handle unary minus.
- Difficult to decide which language is recognized by grammar.

Advantages

1. Simple
2. Powerful enough for expressions in programming language.

Error cases

1. No relation holds between the terminal on the top of stack and the next input symbol.
2. A handle is found, but there is no production with this handle as the right side.

Error recovery

1. Each empty entry is filled with a pointer to an error routine.
2. Based on the handle tries to recover from the situation.

To recover, we must modify (insert/change)

1. Stack or
2. Input or
3. Both

We must be careful that we don't get into an infinite loop.

LR Parsers

- In LR (*K*), L stands for Left to Right Scanning, R stands for Right most derivation, *K* stands for number of look ahead symbols.
- LR parsers are table-driven, much like the non-recursive LL parsers. A grammar which is used in construction of LR parser is LR grammar. For a grammar to be LR it is sufficient that a left-to-right shift-reduce parser be able to recognize handles of right-sentential forms when they appear on the top of the stack.
- The Time complexity for such parsers is $O(n^3)$
- LR parsers are faster than LL (1) parser.
- LR parsing is attractive because
  - The most general non-backtracking shift reduce parser.
  - The class of grammars that can be passed using LR methods is a proper superset of predictive parsers. LL (1) grammars $\subset$ LR (1) grammars.
  - LR parser can detect a syntactic error in the left to right scan of the input.
- LR parsers can be implemented in 3 ways:

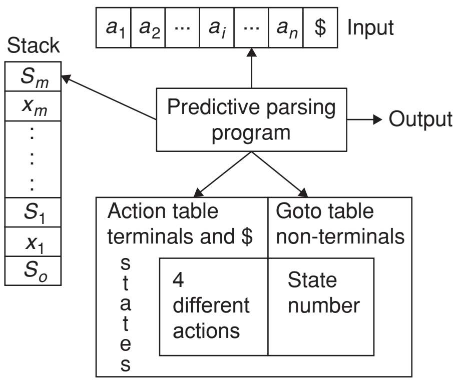
1. Simple LR (SLR): The easiest to implement but the least powerful of the three.
2. Canonical LR (CLR): most powerful and most expensive.
3. Look ahead LR (LALR): Intermediate between the remaining two. It works on most programming language grammars.

Disadvantages of LR parser

1. Detecting a handle is an overhead, parse generator is used.
2. The main problem is finding the handle on the stack and it was replaced with the non-terminal with the left hand side of the production.

The LR parsing algorithm

- It consists of an input, an output, a stack, a driver program and a parsing table that has two parts (action and goto).
- The driver/parser program is same for all these LR parsers, only the parsing table changes from parser to another.



Stack: To store the string of the form,

$S_o x_1 S_1 \ldots x_m S_m$ where

S_m: state

x_m: grammar symbol

Each state symbol summarizes the information contained in the stack below it.

Parsing table: Parsing table consists of two parts:

1. Action part
2. Goto part

ACTION Part:

Let, $S_m \rightarrow$ top of the stack
$a_i \rightarrow$ current symbol

Then action $[S_m, a_i]$ which can have one of four values:

1. Shift *S*, where *S* is a state
2. Reduce by a grammar production $A \rightarrow \beta$
3. Accept
4. Error

GOTO Part:

If goto $(S, A) = X$ where $S \rightarrow$ state, $A \rightarrow$ non-terminal, then GOTO maps state *S* and non-terminal *A* to state *X*.

Configuration

$$(S_o\, x_1 S_1\, x_2 S_2 - x_m S_m,\, a_i a_{i+1} - a_n\$)$$

The next move of the parser is based on action $[S_m, a_i]$ The configurations are as follows.

1. If action $[S_m, a_i]$ = shift *S*
$$(S_o\, x_1 S_1\, x_2 S_2 \text{---}\, x_m S_m,\, a_i a_{i+1} \text{---}\, a_n\$)$$
2. If action $[S_m, a_i]$ = reduce $A \rightarrow \beta$ then
$$(S_o\, x_1 S_1\, x_2 S_2 \text{---}\, x_{m-r} S_{m-r},\, AS,\, a_i a_{i+1} \text{---}\, a_n\$)$$
Where S = goto $[S_{m-r}, A]$
3. If action $[S_m, a_i]$ = accept, parsing is complete.
4. If action $[S_m, a_i]$ = error, it calls an error recovery routine.

Example: Parsing table for the following grammar is shown below:

1. $E \rightarrow E + T$ 2. $E \rightarrow T$

3. $T \to T * F$ 4. $T \to F$
5. $F \to (E)$ 6. $F \to \text{id}$

	Action						Goto		
State	id	+	×	(	)	$	E	T	F
0	S_5			S_4			1	2	3
1		S_6				*acc*			
2		r_2	S_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	S_5			S_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_1				
9		r_1	S_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

Moves of LR parser on input string id*id+id is shown below:

Stack	Input	Action
0	id * id + id$	Shift 5
0id 5	* id + id$	reduce 6 means reduce with 6th production $F \to \text{id}$ and goto [0, F] = 3
0F 3	* id + id$	reduce 4 i.e $T \to F$ goto [0, T] = 2
0T 2	* id + id$	Shift 7
0T2 * 7	id + id$	Shift 5
0T2 * 7 id 5	+ id$	reduce 6 i.e $F \to \text{id}$ goto [7, F] = 10
0T2 * 7 F 10	+ id$	reduce 3 i.e $T \to T * F$
0T 2	+ id$	goto [0, T] = 2
0E 1	+ id$	reduce 2 i.e $E \to T$ & goto [0, E] = 1
0E1 + 6	id$	Shift 6
0E1 + 6 id 5	$	Shift 5
0E1 + 6F 3	$	reduce 6 & goto [6, F] = 3
0E1 + 6T 9	$	reduce 4 & goto [6, T] = 9
0E1	$	reduce 1 & goto [0, E] = 1
0E1	$	accept

Constructing SLR parsing table

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

Example: $A \to BCD$
Possible LR (0) items are

$A \to .BCD$
$A \to B.CD$
$A \to BC.D$
$A \to BCD.$

$A \to B.CD$ means we have seen an input string derivable from B and hope to see a string derivable from CD.

The LR (0) items are constructed as a DFA from grammar to recognize viable prefixes.

The items can be viewed as the states of NFA.

The LR (0) item (or) canonical LR (0) collection, provides the basis for constructing SLR parser.

To construct LR (0) items, define
(a) An augmented grammar
(b) closure and goto

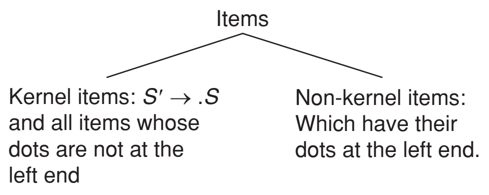
Augmented grammar (G') If G is a grammar with start symbol S, G' the augmented grammar for G, with new start symbol S' and production $S' \to S$.

Purpose of G' is to indicate when to stop parsing and announce acceptance of the input.

Closure operation Closure (I) includes
1. Intially, every item in I is added to closure (I)
2. If A $\to \alpha.B\beta$ is in closure (I) and $\beta \to \gamma$ is a production then add $B \to .\gamma$ to I.

Goto operation

Goto (I, x) is defined to be the closure of the set of all items [$A \to \alpha X.\beta$] such that [$A \to \alpha.X\beta$] is in I.


Items
Kernel items: $S' \to .S$ and all items whose dots are not at the left end
Non-kernel items: Which have their dots at the left end.

Construction of sets of Items

```
Procedure items (G′)
Begin
C: = closure ({[S′ → .S]});
repeat
For each set of items I in C and each grammar symbol x
Such that goto (I, x) is not empty and not in C do add goto
(I, x) to C;
Until no more sets of items can be added to C, end;
```

Example: LR (0) items for the grammar

$E' \to E$
$E \to E + T/T$
$T \to T * F/F$
$F \to (E)/\text{id}$

is given below:
I_0: $E' \to .E$
$E \to .E + T$
$E \to .T$
$T \to .T * F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .\text{id}$

I_1: got (I_0, E)

$E' \rightarrow E.$

$E \rightarrow E. + T$

I_2: goto (I_0, T)

$E \rightarrow T.$

$T \rightarrow T. * F$

I_3: goto (I_0, F)

$T \rightarrow F.$

I_4: goto $(I_0, ()$

$F \rightarrow (.E)$

$E \rightarrow .E + T$

$E \rightarrow .T$

$E \rightarrow .T * F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .\text{id}$

I_5: goto (I_0, id)

$F \rightarrow \text{id}.$

I_6: got $(I_1, +)$

$E \rightarrow E+ .T$

$T \rightarrow .T * F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

I_7: goto $(I_2, *)$

$T \rightarrow T* .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

I_8: goto (I_4, E)

$F \rightarrow (E.)$

I_9: goto (I_6, T)

$E \rightarrow E+ T.$

$T \rightarrow T.* F$

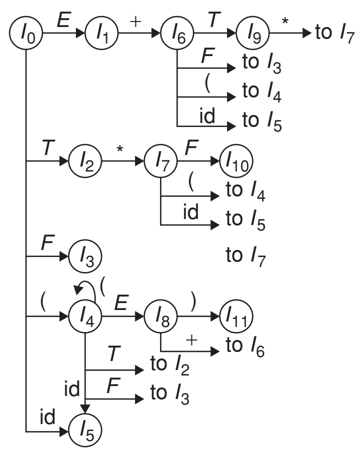
I_{10}: goto (I_7, F)

$T \rightarrow T* F.$

I_{11}: goto $(I_8,))$

$F \rightarrow (E).$

For viable prefixes construct the DFA as follows:



SLR parsing table construction

1. Construct the canonical collection of sets of LR (0) items for G'.
2. Create the parsing action table as follows:
   (a) If a is a terminal and $[A \rightarrow \alpha.a\beta]$ is in I_i, goto $(I_i, a) = I_j$ then action (i, a) to shift j. Here 'a' must be a terminal.
   (b) If $[A \rightarrow \alpha.]$ is in I_i, 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'.
3. Create the parsing goto table for all non-terminals A, if goto $(I_i, A) = I_j$ then goto $[i, A] = j$.
4. All entries not defined by steps 2 and 3 are made errors.
5. Initial state of the parser contains $S' \rightarrow S$.
   The parsing table constructed using the above algorithm is known as SLR (1) table for G.

Note: Every SLR (1) grammar is unambiguous, but every unambiguous grammar is not a SLR grammar.

Example 6: Construct SLR parsing table for the following grammar:

1. $S \rightarrow L = R$
2. $S \rightarrow R$
3. $L \rightarrow * R$
4. $L \rightarrow \text{id}$
5. $R \rightarrow L$

Solution: For the construction of SLR parsing table, add $S' \rightarrow S$ production.

$S' \rightarrow S$

$S \rightarrow L = R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow \text{id}$

$R \rightarrow L$

LR (0) items will be

I_0: $S' \to .S$

$S \to .L = R$

$S \to .R$

$L \to .*R$

$L \to .\text{id}$

$R \to .L$

I_1: goto (I_0, S)

$S' \to S.$

I_2: goto (I_0, L)

$S \to L. = R$

$R \to L.$

I_3: got (I_0, R)

$S \to R.$

I_4: goto (I_0, *)

$L \to *.R$

$R \to .L$

$L \to .*R$

$L \to .\text{id}$

I_5: goto(I_0, id)

$L \to \text{id}.$

I_6: goto(I_2, =)

$S \to L = .R$

$R \to .L$

$L \to .*R$

$L \to .\text{id}$

I_7: goto(I_4, R)

$L \to *R.$

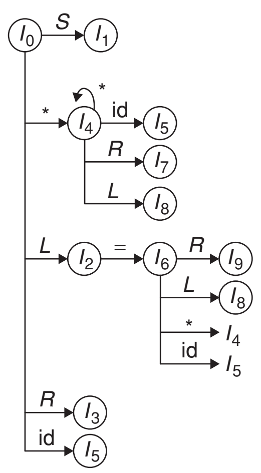
I_8: goto(I_4, L)

$R \to L.$

I_9: goto(I_6, R)

$S \to L = R.$

The DFA of LR(0) items will be



	Action				Goto		
States	=	*	id	$	*S*	*L*	*R*
0		S_4	S_5		1	2	3
1				acc			
2	S_6, r_5			r_5			
3							
4		S_4	S_5			8	7
5							
6		S_4	S_5			8	9
7							
8							
9							

FOLLOW (S) = {\$}

FOLLOW (L) = {=}

FOLLOW (R) = {\$, =}

For action [2, =] = S_6 and r_5

∴ Here we are getting shift – reduce conflict, so it is not SLR (1).

Canonical LR Parsing (CLR)

- To avoid some of invalid reductions, the states need to carry more information.
- Extra information input into a state by including a terminal symbol as a second component of an item.
- The general form of an item

  $[A \to \alpha.\beta, a]$

  Where $A \to \alpha\beta$ is a production.

  a is terminal/right end marker (\$). We will call it as LR (1) item.

LR (1) item

It is a combination of LR (0) items along with look ahead of the item. Here 1 refers to look ahead of the item.

Construction of the sets of LR (1) items Function closure (I):

```
Begin
Repeat
For each item [$A \to \alpha.B\beta, a$] in I,
Each production $B \to .\gamma$ in $G'$,
And each terminal $b$ in FIRST ($\beta a$)
Such that [$B \to .\gamma, b$] is not in $I$ do
Add [$B \to .\gamma, b$] to $I$;
End;
Until no more items can be added to $I$;
```

Example 7: Construct CLR parsing table for the following grammar:

$S' \to S$

$S \to CC$

$C \to cC/d$

Solution: The initial set of items is

I_0: $S' \to .S, \$$
$S \to .CC, \$$
$A \to \alpha.B\beta, a$

Here $A = S$, $\alpha = \in$, $B = C$, $\beta = C$ and $a = \$$
First (βa) is first ($C\$$) = first (C) = $\{c, d\}$
So, add items $[C \to .cC, c]$
$[C \to .cC, d]$

$\therefore$ Our first set I_0: $S' \to .S, \$$
$S \to .CC, \$$
$C \to$ .coca, c/d
$C \to .d, c/d.$

I_1: goto (I_0, X) if $X = S$
$S' \to S., \$$

I_2 : goto (I_0, C)
$S \to C.C, \$$
$C \to .cC, \$$
$C \to .d, \$$
I_3: goto (I_0, c)
$C \to c.C, c/d$
$C \to .cC, c/d$
$C \to .d\ c/d$

I_4: goto (I_0, d)
$C \to d., c/d$

I_5: goto (I_2, C)
$S \to CC., \$$

I_6: goto (I_2, c)
$C \to c.C; \$$
$C \to ..cC, \$$
$C \to ..d, \$$

I_7: goto (I_2, d)
$C \to d.\ \$$

I_8: goto (I_3, C)
$C \to cC., c/d$

I_9: goto (I_6, C)
$C \to cC., \$$

CLR table is:

	Action			Goto	
States	***c***	***1***	**$**	***S***	***C***
I_0	S_3	S_4		1	2
I_1			acc		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	R_3	r_3			
I_5			r_1		
I_6	S_6	S_7			9
I_7			r_3		
I_8	R_2	r_2			
I_9			r_2		

Consider the string derivation '*dcd*':

$$S \Rightarrow CC \Rightarrow CcC \Rightarrow Ccd \Rightarrow dcd$$

Stack	Input	Action
0	*dcd*$	shift 4
0*d*4	*Cd*$	reduce 3 i.e. $C \to d$
0*C*2	*Cd*$	shift 6
0*C*2*C*6	*D*$	shift 7
0*C*2*C*6*d*7	$	reduce $C \to d$
0*C*2*C*6*C*9	$	reduce $C \to cC$
0*C*2*C*5	$	reduce $S \to CC$
0*S*1	$	

Example 8: Construct CLR parsing table for the grammar:
$S \to L = R$
$S \to R$
$L \to *R$
$L \to id$
$R \to L$

Solution: The canonical set of items is
I_0: $S' \to .S, \$$
$S \to .L = R, \$$
$S \to .R, \$$
$L \to .*R, =$ [first (= R$) = {=}]
$L \to$ .id, =
$R \to .L, \$$

I_1: got (I_0, S)
$S' \to S., \$$

I_2: goto (I_0, L)
$S \to L. = R, \$$
$R \to L., \$$

I_3: goto (I_0, R)
$S \to R., \$$

I_4: got (I_0, *)
$L \to *. R, =$
$R \to .L, =$
$L \to .*R, =$
$L \to$ .id, =
I_5: goto (I_0, id)
$L \to$ id.,=
I_6: goto (I_7, L)
$R \to L., \$$

I_7: goto (I_2, =)
$S \to L = .R,$
$R \to .L, \$$
$L \to .*R, \$$
$L \to$ .id, $

I_8: goto (I_4, R)
$L \to *R., =$

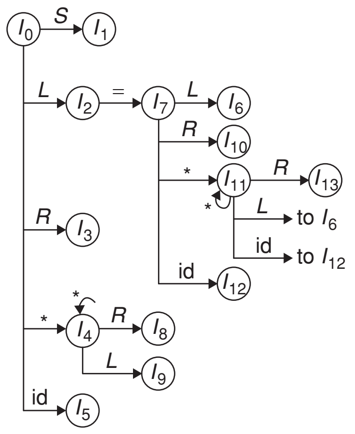
I_9: goto (I_4, L)
$R \rightarrow L., =$

I_{10}: got (I_7, R)
$S \rightarrow L = R., \$$

I_{11}: goto $(I_7, *)$
$L \rightarrow *.R, \$$
$R \rightarrow .L, \$$
$L \rightarrow .*R, \$$
$L \rightarrow .\text{id}, \$$

I_{12}: goto (I_7, id)
$L \rightarrow \text{id}. , \$$

I_{13}: goto (I_{11}, R)
$L \rightarrow *R., \$$



We have to construct CLR parsing table based on the above diagram.

In this, we are going to have 13 states

The shift –reduce conflict in the SLR parser is reduced here.

States	id	*	=	$	S	L	R
0	S_5	S_4			1	2	3
1				acc			
2			S_7	r_5			
3				r_2			
4	S_5	S_4				9	8
5			r_4				
6				r_5			
7	s_{12}	s_{11}				6	10
8			r_3				
9			r_5				
10				r_1			
11	S_{12}	S_{11}					13
12				r_4			
13				r_3			

Stack	Input
0	*Id* = *id*$
0*id*5	= *id*$
0*L*2	= *id*$
0*L*2 = 7	*id*$
0*L*2 = 7! *d*12	$
0*L*2 = 7*L*6	$
0*L*2= 7*R*10	$
0*S*1 (accept)	$

Every SLR (1) grammar is LR (1) grammar.
CLR (1) will have 'more number of states' than SLR Parser.

LALR Parsing Table

- The tables obtained by it are considerably smaller than the canonical LR table.
- LALR stands for Lookahead LR.
- The number of states in SLR and LALR parsing tables for a grammar G are equal.
- But LALR parsers recognize more grammars than SLR.
- YACC creates a LALR parser for the given grammar.
- YACC stands for 'Yet another Compiler'.
- An easy, but space-consuming LALR table construction is explained below:
  1. Construct $C = \{I_0, I_1, –I_n\}$, the collection of sets of LR (1) items.
  2. Find all sets having the common core; replace these sets by their union
  3. Let $C' = \{J_O, J_1 \text{ --- } J_m\}$ be the resulting sets of LR (1) items. If there is a parsing action conflict then the grammar is not a LALR (1).
  4. Let k be the union of all sets of items having the same core. Then goto $(J, X) = k$
- If there are no parsing action conflicts then the grammar is said to LALR (1) grammar.
- The collection of items constructed is called LALR (1) collection.

Example 9: Construct LALR parsing table for the following grammar:

$S' \rightarrow S$
$S \rightarrow CC$
$C \rightarrow cC/d$

Solution: We already got LR (1) items and CLR parsing table for this grammar.
After merging I3 and I6 are replaced by I36.

I_{36}: $C \rightarrow c.C, c/d/\$$
$C \rightarrow .cC, c/d/\$$
$C \rightarrow .d, c/d/\$$

I_{47}: By merging I_4 and I_7
$C \rightarrow d.\ c/d/\$$

I_{89}: I_8 and I_9 are replaced by I_{89}
$C \rightarrow cC.,\ c/d/\$$

The LALR parsing table for this grammar is given below:

	Action			goto	
State	***c***	***d***	**$**	***S***	***C***
0	S_{36}	S_{47}		1	2
1			*acc*		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	R_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

Example: Consider the grammar:

$S' \rightarrow S$
$S \rightarrow aAd$
$S \rightarrow bBd$
$S \rightarrow aBe$
$S \rightarrow bAe$
$A \rightarrow c$
$B \rightarrow c$

Which generates strings acd, bcd, ace and bce

LR (1) items are

I_0: $S' \rightarrow .S,\ \$$
$S \rightarrow .aAd,\ \$$
$S \rightarrow .bBd,\ \$$
$S \rightarrow .aBe,\ \$$
$S \rightarrow .bAe,\ \$$

I_1: goto (I_0, S)
$S' \rightarrow S.,\ \$$

I_2: goto (I_0, a)
$S \rightarrow a.Ad,\ c$
$S \rightarrow a.Be,\ c$
$A \rightarrow .c, d$
$B \rightarrow .c, e$

I_3: goto (I_0, b)
$S \rightarrow b.Bd,\ c$
$S \rightarrow b.Ae,\ c$
$A \rightarrow .c,\ e$
$B \rightarrow .c,\ e$

I_4: goto (I_2, A)
$S \rightarrow aA.d,\ c$

I_5: goto (I_2, B)
$S \rightarrow aB.e,\ c$

I_6: goto (I_2, c)
$A \rightarrow c.,\ d$
$B \rightarrow c.,\ e$

I_7: goto (I_3, c)
$A \rightarrow c.,\ e$
$B \rightarrow c.,\ d$

I_8: goto (I_4, d)
$S \rightarrow aAd.,\ c$

I_9: goto (I_5, e)
$S \rightarrow aBe.,\ c$

If we union I_6 and I_7
$A \rightarrow c.,\ d/e$
$B \rightarrow c.,\ d/e$

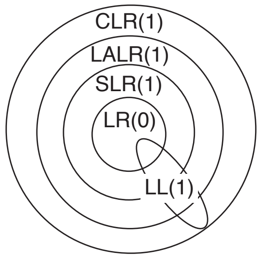
It generates reduce/reduce conflict.

Notes:

1. The merging of states with common cores can never produce a shift/reduce conflict, because shift action depends only on the core, not on the lookahead.
2. SLR and LALR tables for a grammar always have the same number of states (several hundreds) whereas CLR have thousands of states for the same grammar.

Comparison of parsing methods

Method	Item	Goto and Closures	Grammar it Applies to
SLR (1)	LR(0) item	Different from LR(1)	SLR (1) ⊂ LR(1)
LR (1)	LR(1) item		LR(1) – Largest class of LR grammars
LALR(1)	LR(1) item	Same as LR(1)	LALR(1) ⊂ LR(1)



Every LR (0) is SLR (1) but vice versa is not true.

Difference between SLR, LALR and CLR parsers

Differences among SLR, LALR and CLR are discussed below in terms of size, efficiency, time and space.

Table 1 *Comparison of parsing methods*

Sl. No.	Factors	SLR Parser	LALR Parser	CLR Parser
1	Size	Smaller	Smaller	Larger
2.	Method	It is based on FOLLOW function	This method is applicable to wider class than SLR	This is most powerful than SLR and LALR.
3.	Syntactic features	Less exposure compared to other LR parsers	Most of them are expressed	Less
4.	Error detection	Not immediate	Not immediate	Immediate
5.	Time and space complexity	Less time and space	More time and space complexity	More time and space complexity

Exercises

Practice Problems I

Directions for questions 1 to 15: Select the correct alternative from the given choices.

1. Consider the grammar
   $S \to a$
   $S \to ab$
   The given grammar is:
   (A) LR (1) only
   (B) LL (1) only
   (C) Both LR (1) and LL (1)
   (D) LR (1) but not LL (1)

2. Which of the following is an unambiguous grammar, that is not LR (1)?
   (A) $S \to$ Uab | Vac
   $U \to d$
   $V \to d$
   (B) $S \to$ Uab/Vab/Vac
   $U \to d$
   $V \to d$
   (C) $S \to AB$
   $A \to a$
   $B \to b$
   (D) $S \to Ab$
   $A \to a/c$

Common data for questions 3 and 4: Consider the grammar:

$S \to T; S/\in$
$T \to UR$
$U \to x/y/[S]$
$R \to .T/\in$

3. Which of the following are correct FIRST and FOLLOW sets for the above grammar?
   (i) FIRST(S) = FIRST (T) = FIRST (U) = $\{x, y, [, \varepsilon\}$
   (ii) FIRST (R) = $\{,\varepsilon\}$
   (iii) FOLLOW (S) = {], \$}
   (iv) FOLLOW (T) = Follow (R) = {;}
   (v) FOLLOW (U) = {. ;}
   (A) (i) and (ii) only
   (B) (ii), (iii), (iv) and (v) only
   (C) (ii), (iii) and (iv) only
   (D) All the five

4. If an LL (1) parsing table is constructed for the above grammar, the parsing table entry for [$S \to$ [ ] is
   (A) $S \to T; S$ (B) $S \to \in$
   (C) $T \to UR$ (D) $U \to [S]$

Common data for questions 5 to 7: Consider the augmented grammar

$S \to X$
$X \to (X)/a$

5. If a DFA is constructed for the LR (1) items of the above grammar, then the number states present in it are:
   (A) 8 (B) 9
   (C) 7 (D) 10

6. Given grammar is
   (A) Only LR (1)
   (B) Only LL (1)
   (C) Both LR (1) and LL (1)
   (D) Neither LR (1) nor LL (1)

7. What is the number of shift-reduce steps for input (a)?
   (A) 15 (B) 14
   (C) 13 (D) 16

8. Consider the following two sets of LR (1) items of a grammar:

$X \to c.X, c/d$	$X \to c.X, \$$
$X \to .cX, c/d$	$X \to .cX, \$$
$X \to d, c/d$	$X \to .d, \$$

   Which of the following statements related to merging of the two sets in the corresponding LALR parser is/are FALSE?
   1. Cannot be merged since look ahead are different.
   2. Can be merged but will result in $S - R$ conflict.
   3. Can be merged but will result in $R - R$ conflict.
   4. 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

9. Which of the following grammar rules violate the requirements of an operator grammar?
   (i) $A \to BcC$ (ii) $A \to dBC$
   (iii) $A \to C/\in$ (iv) $A \to cBdC$

(A) (i) only (B) (i) and
(C) (ii) and (iii) only (D) (i) and (iv) only

10. The FIRST and FOLLOW sets for the grammar:

$S \to SS+/SS*/a$

(A) First $(S) = \{a\}$
Follow $(S) = \{+, *, \$\}$
(B) First $(S) = \{+\}$
Follow $(S) = \{+, *, \$\}$
(C) First $(S) = \{a\}$
Follow $(S) = \{+, *\}$
(D) First $(S) = \{+, *\}$
Follow $(S) = \{+, *, \$\}$

11. A shift reduces parser carries out the actions specified within braces immediately after reducing with the corresponding rule of the grammar:
$S \to xxW$ [print '1']
$S \to y$ [print '2']
$W \to Sz$ [print '3']
What is the translation of '$x\,x\,x\,x\,y\,z\,z$'?
(A) 1231 (B) 1233
(C) 2131 (D) 2321

12. After constructing the predictive parsing table for the following grammar:
$Z \to d$
$Z \to XYZ$
$Y \to c/\in$
$X \to Y$
$X \to a$
The entry/entries for $[Z, d]$ is/are
(A) $Z \to d$
(B) $Z \to XYZ$
(C) Both (A) and (B)
(D) $X \to Y$

13. The following grammar is
$S \to AaAb/BbBa$
$A \to \varepsilon$
$B \to \varepsilon$
(A) LL (1) (B) Not LL (1)
(C) Recursive (D) Ambiguous

14. Compute the FIRST (P) for the below grammar:
$P \to AQR$be/mn/DE
$A \to ab/\varepsilon$
$Q \to q_1q_2/\varepsilon$
$R \to r_1r_2/\varepsilon$
$D \to d$
$E \to e$
(A) $\{m, a\}$ (B) $\{m, a, q_1, r_1, b, d\}$
(C) $\{d, e\}$ (D) $\{m, n, a, b, d, e, q_1, r_1\}$

15. After constructing the LR(1) parsing table for the augmented grammar
$S' \to S$
$S \to BB$
$B \to aB/c$
What will be the action $[I_3, a]$?
(A) Accept (B) S_7
(C) r_2 (D) S_5

Practice Problems 2

Directions for questions 1 to 19: Select the correct alternative from the given choices.

1. Consider the grammar
$S \to aSb$
$S \to aS$
$S \to \varepsilon$
This grammar is ambiguous by generating which of the following string.
(A) *aa* (B) $\in$
(C) *aaa* (D) *aab*

2. To convert the grammar $E \to E + T$ into LL grammar
(A) use left factor
(B) CNF form
(C) eliminate left recursion
(D) Both (B) and (C)

3. Given the following expressions of a grammar
$E \to E \times F/F + E/F$
$F \to F$? F/id
Which of the following is true?
(A) $\times$ has higher precedence than +
(B) ? has higher precedence than $\times$
(C) + and? have same precedence
(D) + has higher precedence than *

4. The action of parsing the source program into the proper syntactic classes is known as
(A) Lexical analysis
(B) Syntax analysis
(C) Interpretation analysis
(D) Parsing

5. Which of the following is not a bottom up parser?
(A) LALR (B) Predictive parser
(C) CLR (D) SLR

6. A system program that combines separately compiled modules of a program into a form suitable for execution is
(A) Assembler.
(B) Linking loader.
(C) Cross compiler.
(D) None of these.

7. Resolution of externally defined symbols is performed by a
(A) Linker (B) Loader.
(C) Compiler. (D) Interpreter.

8. LR parsers are attractive because
(A) They can be constructed to recognize CFG corresponding to almost all programming constructs.
(B) There is no need of backtracking.
(C) Both (A) and (B).
(D) None of these

9. YACC builds up
(A) SLR parsing table
(B) Canonical LR parsing table
(C) LALR parsing table
(D) None of these

10. Language which have many types, but the type of every name and expression must be calculated at compile time are
(A) Strongly typed languages
(B) Weakly typed languages
(C) Loosely typed languages
(D) None of these

11. Consider the grammar shown below:
$S \rightarrow iEtSS'/a/b$
$S' \rightarrow eS/\varepsilon$
In the predictive parse table M, of this grammar, the entries $M[S', e]$ and $M[S', \$]$ respectively are
(A) $\{S' \rightarrow eS\}$ and $\{S' \rightarrow \in\}$
(B) $\{S' \rightarrow eS\}$ and $\{\ \}$
(C) $\{S' \rightarrow \in\}$ and $\{S' \rightarrow \in\}$
(D) $\{S' \rightarrow eS, S' \rightarrow \varepsilon\}\}$ and $\{S' \rightarrow \in\}$

12. Consider the grammar $S \rightarrow CC$, $C \rightarrow cC/d$.
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)

13. Consider the grammar
$E \rightarrow E + n/E - n/n$
For a sentence $n + n - n$, the handles in the right sentential form of the reduction are
(A) n, $E + n$ and $E + n - n$
(B) n, $E + n$ and $E + E - n$
(C) n, $n + n$ and $n + n - n$
(D) n, $E + n$ and $E - n$

14. A top down parser uses ___ derivation.
(A) Left most derivation
(B) Left most derivation in reverse
(C) Right most derivation
(D) Right most derivation in reverse

15. Which of the following statement is false?
(A) An unambiguous grammar has single leftmost derivation.
(B) An LL (1) parser is topdown.
(C) LALR is more powerful than SLR.
(D) An ambiguous grammar can never be LR (K) for any k.

16. Merging states with a common core may produce ___ conflicts in an LALR parser.
(A) Reduce – reduce
(B) Shift – reduce
(C) Both (A) and (B)
(D) None of these

17. LL (K) grammar
(A) Has to be CFG
(B) Has to be unambiguous
(C) Cannot have left recursion
(D) All of these

18. The I_0 state of the LR (0) items for the grammar
$S \rightarrow AS/b$
$A \rightarrow SA/a$.
(A) $S' \rightarrow .S$
$S \rightarrow .As$
$S \rightarrow .b$
$A \rightarrow .SA$
$A \rightarrow .a$
(B) $S \rightarrow .AS$
$S \rightarrow .b$
$A \rightarrow .SA$
$A \rightarrow .a$
(C) $S \rightarrow .AS$
$S \rightarrow .b$
(D) $S \rightarrow A$
$A \rightarrow .SA$
$A \rightarrow .a$

19. In the predictive parsing table for the grammar:
$S \rightarrow FR$
$R \rightarrow \times S/e$
$F \rightarrow \text{id}$
What will be the entry for [S, id]?
(A) $S \rightarrow FR$
(B) $F \rightarrow \text{id}$
(C) Both (A) and (B)
(D) None of these

Previous Years' Questions

1. Consider the grammar:
$S \to (S) \mid a$
Let the number of states in SLR (1), LR (1) and LALR (1) parsers for the grammar be n_1, n_2 and n_3 respectively. The following relationship holds good: **[2005]**
(A) $n_1 < n_2 < n_3$ (B) $n_1 = n_3 < n_2$
(C) $n_1 = n_2 = n_3$ (D) $n_1 \geq n_3 \geq n_2$

2. Consider the following grammar:
$S \to S * E$
$S \to E$
$E \to F + E$
$E \to F$
$F \to \text{id}$
Consider the following LR (0) items corresponding to the grammar above.
(i) $S \to S * .E$
(ii) $E \to F. + E$
(iii) $E \to F + .E$
Given the items above, which two of them will appear in the same set in the canonical sets-of items for the grammar? **[2006]**
(A) (i) and (ii) (B) (ii) and (iii)
(C) (i) and (iii) (D) None of the above

3. Consider the following statements about the context-free grammar
$G = \{S \to SS, S \to ab, S \to ba, S \to \in\}$
(i) G is ambiguous
(ii) G produces all strings with equal number of a's and b's
(iii) G can be accepted by a deterministic PDA.
Which combination below expresses all the true statements about G? **[2006]**
(A) (i) only (B) (i) and (iii) only
(C) (ii) and (iii) only (D) (i), (ii) and (iii)

4. Consider the following grammar:
$S \to FR$
$R \to *S|\varepsilon$
$F \to \text{id}$
In the predictive parser table, M, of the grammar the entries M[S, id] and M[R, \$] respectively. **[2006]**
(A) $\{S \to FR\}$ and $\{R \to \varepsilon\}$
(B) $\{S \to FR\}$ and $\{\ \}$
(C) $\{S \to FR\}$ and $\{R \to *S\}$
(D) $\{F \to \text{id}\}$ and $\{R \to \varepsilon\}$

5. Which one of the following grammars generates the language $L = \{a^i b^j | i \neq j\}$? **[2006]**
(A) $S \to AC|CB$
$C \to aCb|a|b$
$A \to aA|\in$
$B \to Bb|\in$
(B) $S \to aS|Sb|a|b$
(C) $S \to AC|CB$
$C \to aC|b|\in$
$A \to aA|\in$
$B \to Bb|\in$
(D) $S \to AC|CB$
$C \to aCb|\in$
$A \to aA|a$
$B \to Bb|b$

6. In the correct grammar above, what is the length of the derivation (number of steps starting from S) to generate the string $a^\ell b^m$ with $\ell \neq m$? **[2006]**
(A) max (l, m) + 2
(B) l + m + 2
(C) l + m + 3
(D) max (l, m) + 3

7. Which of the following problems is undecidable? **[2007]**
(A) Membership problem for CFGs.
(B) Ambiguity problem for CFGs.
(C) Finiteness problem for FSAs.
(D) Equivalence problem for FSAs.

8. Which one of the following is a top-down parser? **[2007]**
(A) Recursive descent parser.
(B) Operator precedence parser.
(C) An LR (k) parser.
(D) An LALR (k) parser.

9. Consider the grammar with non-terminals $N = \{S, C, and\ S_1\}$, terminals $T = \{a, b, i, t, e\}$ with S as the start symbol, and the following set of rules: **[2007]**
$S \to iCtSS_1/a$
$S_1 \to eS/\varepsilon$
$C \to b$
The grammar is NOT LL (1) because:
(A) It is left recursive
(B) It is right recursive
(C) It is ambiguous
(D) It is not context-free.

10. Consider the following two statements:
P: Every regular grammar is LL (1)
Q: Every regular set has a LR (1) grammar
Which of the following is TRUE? **[2007]**
(A) Both P and Q are true
(B) P is true and Q is false
(C) P is false and Q is true
(D) Both P and Q are false

Common data for questions 11 and 12: Consider the CFG with $\{S, A, B\}$ as the non-terminal alphabet, $\{a, b\}$ as the terminal alphabet, S as the start symbol and the following set of production rules:

$S \to aB$	$S \to bA$
$B \to b$	$A \to a$
$B \to bS$	$A \to aS$
$B \to aBB$	$S \to bAA$

11. Which of the following strings is generated by the grammar? **[2007]**

(A) *aaaabb* (B) *aabbbb*
(C) *aabbab* (D) *abbbba*

12. For the correct answer strings to Q.78, how many derivation trees are there? **[2007]**

(A) 1 (B) 2
(C) 3 (D) 4

13. Which of the following describes a handle (as applicable to LR-parsing) appropriately? **[2008]**

(A) It is the position in a sentential form where the next shift or reduce operation will occur.
(B) It is non-terminal whose production will be used for reduction in the next step.
(C) It is a production that may be used for reduction in a future step along with a position in the sentential form where the next shift or reduce operation will occur.
(D) It is the production *p* that will be used for reduction in the next step along with a position in the sentential form where the right hand side of the production may be found.

14. Which of the following statements are true?

(i) Every left-recursive grammar can be converted to a right-recursive grammar and vice-versa
(ii) All ε-productions can be removed from any context-free grammar by suitable transformations
(iii) The language generated by a context-free grammar all of whose productions are of the form $X \rightarrow w$ or $X \rightarrow wY$ (where, w is a string of terminals and Y is a non-terminal), is always regular
(iv) The derivation trees of strings generated by a context-free grammar in Chomsky Normal Form are always binary trees **[2008]**

(A) (i), (ii), (iii) and (iv) (B) (ii), (iii) and (iv) only
(C) (i), (iii) and (iv) only (D) (i), (ii) and (iv) only

15. An LALR (1) parser for a grammar *G* can have shift-reduce (*S*–*R*) conflicts if and only if **[2008]**

(A) The SLR (1) parser for *G* has *S*–*R* conflicts
(B) The LR (1) parser for *G* has *S*–*R* conflicts
(C) The LR (0) parser for *G* has *S*–*R* conflicts
(D) The LALR (1) parser for *G* has reduce-reduce conflicts

16. $S \rightarrow aSa \mid bSb \mid a \mid b;$
The language generated by the above grammar over the alphabet $\{a, b\}$ is the set of **[2009]**

(A) All palindromes.
(B) All odd length palindromes.
(C) Strings that begin and end with the same symbol.
(D) All even length palindromes.

17. Which data structure in a compiler is used for managing information about variables and their attributes? **[2010]**

(A) Abstract syntax tree
(B) Symbol table
(C) Semantic stack
(D) Parse table

18. The grammar $S \rightarrow aSa|bS|c$ is **[2010]**

(A) LL (1) but not LR (1)
(B) LR (1) but not LR (1)
(C) Both LL (1) and LR (1)
(D) Neither LL (1) nor LR (1)

19. The lexical analysis for a modern computer language such as Java needs the power of which one of the following machine models in a necessary and sufficient sense? **[2011]**

(A) Finite state automata
(B) Deterministic pushdown automata
(C) Non-deterministic pushdown automata
(D) Turing machine

Common data for questions 20 and 21: For the grammar below, a partial LL (1) parsing table is also presented along with the grammar. Entries that need to be filled are indicated as E_1, E_2, and E_3. Is the empty string, \$ indicates end of input, and, *I* separates alternate right hand side of productions

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

$A \rightarrow S$

$B \rightarrow S$

	a	b	\$
S	E_1	E_2	$S \rightarrow e$
A	$A \rightarrow S$	$A \rightarrow S$	Error
B	$B \rightarrow S$	$B \rightarrow S$	E_3

20. The FIRST and FOLLOW sets for the non-terminals A and B are **[2012]**

(A) FIRST $(A) = \{a, b, \varepsilon\}$ = FIRST (B)
FOLLOW $(A) = \{a, b\}$
FOLLOW $(B) = \{a, b, \$\}$
(B) FIRST $(A) = \{a, b, \$\}$
FIRST $(B) = \{a, b, \varepsilon\}$
FOLLOW $(A) = \{a, b\}$
FOLLOW $(B) = \{\$\}$
(C) FIRST $(A) = \{a, b, \varepsilon\}$ = FIRST (B)
FOLLOW $(A) = \{a, b\}$
FOLLOW $(B) = \varnothing$
(D) FIRST $(A) = \{a, b\}$ = FIRST (B)
FOLLOW $(A) = \{a, b\}$
FOLLOW $(B) = \{a, b\}$

21. The appropriate entries for E_1, E_2, and E_3 are **[2012]**

(A) E_1: $S \rightarrow a\ A\ b\ B,\ A \rightarrow S$
E_2: $S \rightarrow b\ A\ a\ B,\ B \rightarrow S$
E_3: $B \rightarrow S$

(B) E_1: $S \to a\,A\,b\,B, S \to \varepsilon$
E_2: $S \to b\,A\,a\,B, S \to \varepsilon$
E_3: $S \to \in$
(C) E_1: $S \to a\,A\,b\,B, S \to \varepsilon$
E_2: $S \to b\,A\,a\,B, S \to \varepsilon$
E_3: $B \to S$
(D) E_1: $A \to S, S \to \varepsilon$
E_2: $B \to S, S \to \varepsilon$
E_3: $B \to S$

22. What is the maximum number of reduce moves that can be taken by a bottom-up parser for a grammar with no epsilon-and unit-production (i.e., of type $A \to \in$ and $A \to a$) to parse a string with n tokens? **[2013]**

(A) $n/2$ (B) $n - 1$
(C) $2n - 1$ (D) 2^n

23. Which of the following is/are undecidable?
(i) G is a CFG. Is $L(G) = \phi$?
(ii) G is a CFG, Is $L(G) = \Sigma^*$?
(iii) M is a Turing machine. Is $L(M)$ regular?
(iv) A is a DFA and N is an NFA. Is $L(A) = L(N)$? **[2013]**

(A) (iii) only
(B) (iii) and (iv) only
(C) (i), (ii) and (iii) only
(D) (ii) and (iii) only

24. Consider the following two sets of LR (1) items of an LR (1) grammar. **[2013]**

$X \to c.X, c/d$ $X \to c.X, \$$
$X \to .cX, c/d$ $X \to .cX, \$$
$X \to .d, c/d$ $X \to .d, \$$

Which of the following statements related to merging of the two sets in the corresponding LALR parser is/are FALSE?
(i) Cannot be merged since look - ahead are different.
(ii) Can be merged but will result in S-R conflict.
(iii) Can be merged but will result in R-R conflict.
(iv) Cannot be merged since goto on c will lead to two different sets.

(A) (i) only (B) (ii) only
(C) (i) and (iv) only (D) (i), (ii), (iii) and (iv)

25. A canonical set of items is given below
$S \to L. > R$
$Q \to R.$
On input symbol < the sset has **[2014]**
(A) A shift–reduce conflict and a reduce–reduce conflict.
(B) A shift–reduce conflict but not a reduce–reduce conflict.
(C) A reduce–reduce conflict but not a shift reduce conflict.
(D) Neither a shift–reduce nor a reduce–reduce conflict.

26. Consider the grammar defined by the following production rules, with two operators * and +
$S \to T * P$
$T \to U | T * U$
$P \to Q + P | Q$
$Q \to Id$
$U \to Id$
Which one of the following is TRUE? **[2014]**
(A) + is left associative, while * is right associative
(B) + is right associative, while * is left associative
(C) Both + and * are right associative.
(D) Both + and * are left associative

27. Which one of the following problems is undecidable? **[2014]**
(A) Deciding if a given context -free grammar is ambiguous.
(B) Deciding if a given string is generated by a given context-free grammar.
(C) Deciding if the language generated by a given context-free grammar is empty.
(D) Deciding if the language generated by a given context free grammar is finite.

28. Which one of the following is TRUE at any valid state in shift-reduce parsing? **[2015]**
(A) Viable prefixes appear only at the bottom of the stack and not inside.
(B) Viable prefixes appear only at the top of the stack and not inside.
(C) The stack contains only a set of viable prefixes.
(D) The stack never contains viable prefixes.

29. Among simple LR (SLR), canonical LR, and look-ahead LR (LALR), which of the following pairs identify the method that is very easy to implement and the method that is the most powerful, in that order? **[2015]**
(A) SLR, LALR
(B) Canonical LR, LALR
(C) SLR, canonical LR
(D) LALR, canonical LR

30. Consider the following grammar G
$S \to F \mid H$
$F \to p \mid c$
$H \to d \mid c$
Where S, F and H are non-terminal symbols, p, d and c are terminal symbols. Which of the following statement(s) is/are correct? **[2015]**
S_1. LL(1) can parse all strings that are generated using grammar G
S_2. LR(1) can parse all strings that are generated using grammar G

(A) Only S_1 (B) Only S_2
(C) Both S_1 and S_2 (D) Neither S_1 nor S_2

31. Match the following: **[2016]**

(P) Lexical analysis	(i) Leftmost derivation
(Q) Top down parsing	(ii) Type checking
(R) Semantic analysis	(iii) Regular expressions
(S) Runtime environments	(iv) Activation records

(A) $P \leftrightarrow$ i, $Q \leftrightarrow$ ii, $R \leftrightarrow$ iv, $S \leftrightarrow$ iii
(B) $P \leftrightarrow$ iii, $Q \leftrightarrow$ i, $R \leftrightarrow$ ii, $S \leftrightarrow$ iv
(C) $P \leftrightarrow$ ii, $Q \leftrightarrow$ iii, $R \leftrightarrow$ i, $S \leftrightarrow$ iv
(D) $P \leftrightarrow$ iv, $Q \leftrightarrow$ i, $R \leftrightarrow$ ii, $S \leftrightarrow$ iii

32. A student wrote two context - free grammars **G1** and **G2** for generating a single C-like array declaration. The dimension of the array is at least one.

For example, int a [10] [3];

The grammars use D as the start symbol, and use six terminal symbols int; id [ ] num. **[2016]**

Grammar **G1**	Grammar **G2**
D → **int** L;	D → **int**L;
L → **id** [E	L → **id** E
E → **num**]	E → E [**num**]
E → **num**] [E	E → [**num**]

Which of the grammars correctly generate the declaration mentioned above?

(A) Both **G1** and **G2**
(B) Only **G1**
(C) Only **G2**
(D) Neither **G1** nor **G2**

33. Consider the following grammar:

$$\boxed{\begin{array}{l} P \rightarrow xQRS \\ Q \rightarrow yz \mid z \\ R \rightarrow w \mid \varepsilon \\ S \rightarrow y \end{array}}$$

What is FOLLOW (Q)? **[2017]**

(A) $\{R\}$ (B) $\{w\}$
(C) $\{w, y\}$ (D) $\{w, \$\}$

34. Which of the following statements about parser is/are CORRECT? **[2017]**

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) I and III only

35. Consider the following expression grammar G :

E – > E – T | T
T – > T + F | F
F – > (E) | id

Which of the following grammars is not left recursive, but is equivalent to G? **[2017]**

(A) E – >E – T | T
T – > T + F | F
F – > (E) | id

(B) E – > TE
E′ – > –TE | ∈
T – > T + F | F
F – > (E) | id

(C) E – > TX
X – > –TX | ∈
T – > FY
Y – > + FY | ∈
F – > (E) | id

(D) E – > TX | (TX)
X – > –TX | +TX | ∈
T – > id

36. Which one of the following statements is FALSE? **[2018]**

(A) Context-free grammar can be used to specify both lexical and syntax rules.
(B) Type checking is done before parsing.
(C) High-level language programs can be translated to different Intermediate Representations.
(D) Arguments to a function can be passed using the program stack.

37. A lexical analyzer uses the following patterns to recognize three tokens T_1, T_2, and T_3 over the alphabet $\{a, b, c\}$.

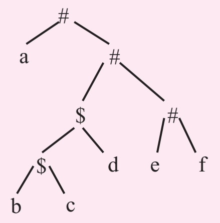
T_1: $a?(b|c)^*a$
T_2: $b?(a|c)^*b$
T_3: $c?(b|a)^*c$

Note that 'x?' means 0 or 1 occurrence of the symbol x. Note also that the analyzer outputs the token that matches the longest possible prefix.

If the string bbaacabc is processed by the analyzer, which one of the following is the sequence of tokens it outputs? **[2018]**

(A) $T_1T_2T_3$ (B) $T_1T_1T_3$
(C) $T_2T_1T_3$ (D) T_3T_3

38. Consider the following parse tree for the expression a#b\$c\$d#e#f, involving two binary operators \$ and #.



Which one of the following is correct for the given parse tree? **[2018]**

(A) \$ has higher precedence and is left associative; # is right associative
(B) # has higher precedence and is left associative; \$ is right associative
(C) \$ has higher precedence and is left associative; # is left associative
(D) # has higher precedence and is right associative; \$ is left associative

Answer Keys

Exercises

Practice Problems 1

1. D	**2.** A	**3.** B	**4.** A	**5.** D	**6.** C	**7.** C	**8.** D	**9.** C	**10.** A
11. C	**12.** C	**13.** A	**14.** B	**15.** D					

Practice Problems 2

1. D	**2.** C	**3.** B	**4.** A	**5.** B	**6.** B	**7.** A	**8.** C	**9.** C	**10.** A
11. D	**12.** A	**13.** D	**14.** A	**15.** D	**16.** A	**17.** C	**18.** A	**19.** A	

Previous Years' Questions

1. B	**2.** D	**3.** B	**4.** A	**5.** D	**6.** A	**7.** B	**8.** A	**9.** C	**10.** A
11. C	**12.** B	**13.** D	**14.** C	**15.** B	**16.** B	**17.** B	**18.** C	**19.** A	**20.** A
21. C	**22.** B	**23.** D	**24.** D	**25.** D	**26.** B	**27.** A	**28.** C	**29.** C	**30.** D
31. B	**32.** A	**33.** C	**34.** A	**35.** C	**36.** B	**37.** D	**38.** A		