1 Introduction to Compiler Design

Language processing system:

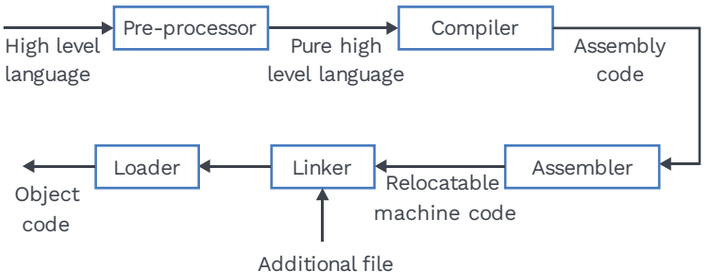


Fig. 1.1 Language Processing System

- The pre-processor is the first program to get invoked.
- The main function of a pre-processor is to substitute the macros with the exact value.
- The compiler is the next code to run. This program accepts a preprocessed file as input and outputs assembly language.
- The assembler command is used to translate the assembly file into a relocatable object file in the third stage. The relocatable object file is not in a format that can be read by humans. (It comes in ELF (executable and linking format) and COFF formats) (common object file format).
- The final step is the invocation of the linker and loader to generate the executable file. The linker utility links the relocatable object file with the system-wide startup objects file and makes it executable.

Compiler and translator:

A translator is a program that takes a program as input written in one programming language (the source language) and produce a program as output in another language (The object or target language).

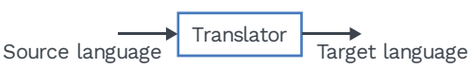


Fig. 1.2 Translator

If the source language is a high-level language and the object language is a low-level language such as assembly language, Then such translator is called compiler.

Fig. 1.3 Compiler

Compilation and execution:

A program written in a high-level language is usually executed in two steps.

Step 1:
First, the source program must be compiled, or converted into the object program.

Fig. 1.4 Compiler

Step 2:
The generated object program is loaded into memory and executed.

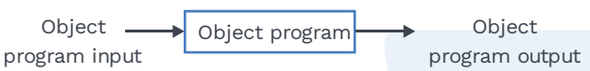


Fig. 1.5 Object Program

Note

- The translator is known as an assembler if the source language is assembly, and the target language is machine language.
- Pre-processors are sometimes used for translators that take programs in one high-level language into an equivalent programs in another high-level language.

Rack Your Brain

Why do we need translators?

Phases of a compiler:
A phase is a logically consistent procedure that accepts one representation of a source programme as input and outputs another one.

- Lexical analysis, often known as the scanner divides characters from the source language into tokens, or groupings of characters that logically belong together.
- The tokens are the basic unit of programming language, which can not be broken up further.
- Syntax analysis phase takes tokens as an input and produce parse tree as output.

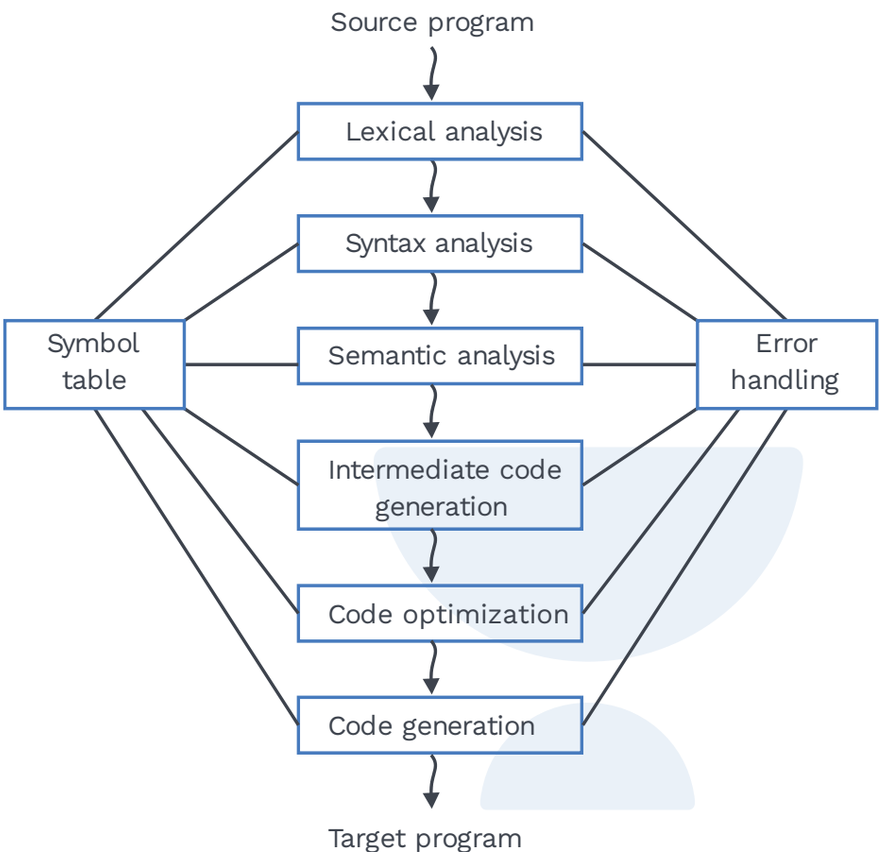


Fig. 1.6 Flow of a Program Execution

- Semantic analysis phase takes the parse tree as input and produces annotated parse tree (decorated parse tree/ semantically verified parse tree) as output.
- The intermediate code generator creates a stream of simple instructions using the structure generated by the semantic analyzer. (for example 3-address code).
- Code optimization is an optional phase that aims to enhance intermediate code in order to make the final object program run quicker and/or take-up less space.
- Code generation generates object code by deciding on data memory locations.
- The symbol table is a data structure that keeps track of the names used by the program and records important information about each one, such as its type (integer, real, etc.).
- When an error in the source program is discovered, the error handler is invoked.

Note

- One of the most difficult aspects of compiler design, both practically and theoretically, is creating a C code generator that produces truly efficient object program.
- Symbol table and error handling routines interact with all phases of the compiler

Analysis and synthesis phase:

- There are two major parts of the compiler:
  1) Analysis phase
  2) Synthesis phase
- Analysis phase contains a lexical analyzer, syntax analyzer and semantic analyzer.
- Synthesis phase contains an Intermediate code generator code optimiser and target code generator.

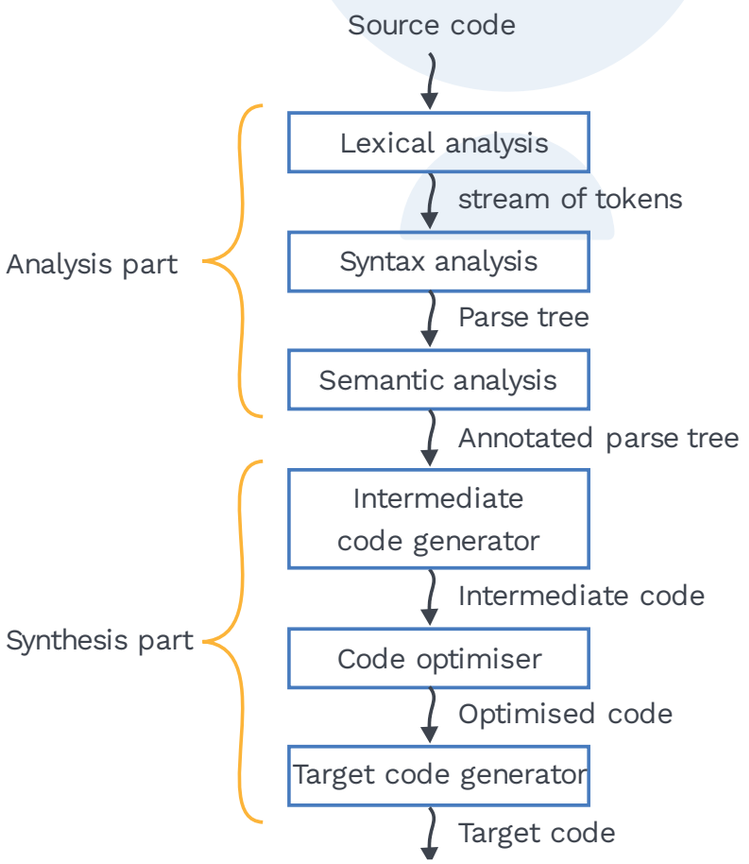


Fig. 1.7 Parts of Flow of Program Execution

Front end and back end of a compiler:

- The front end is divided into two phases: one that is dependent on the input (source language) and the other that is independent of the destination machine (target language).
- The phases of the compiler's front end are as follows.

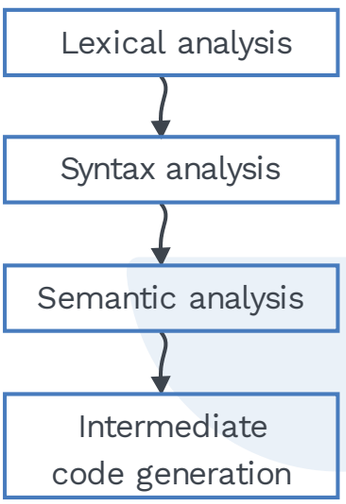


Fig. 1.8 Analysis Part

- Back end is subdivided into phases of a compiler that depend on the target machine and Independent of the source language.
- The following phases constitute the compiler's back end.

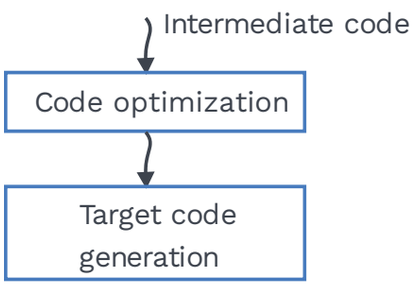
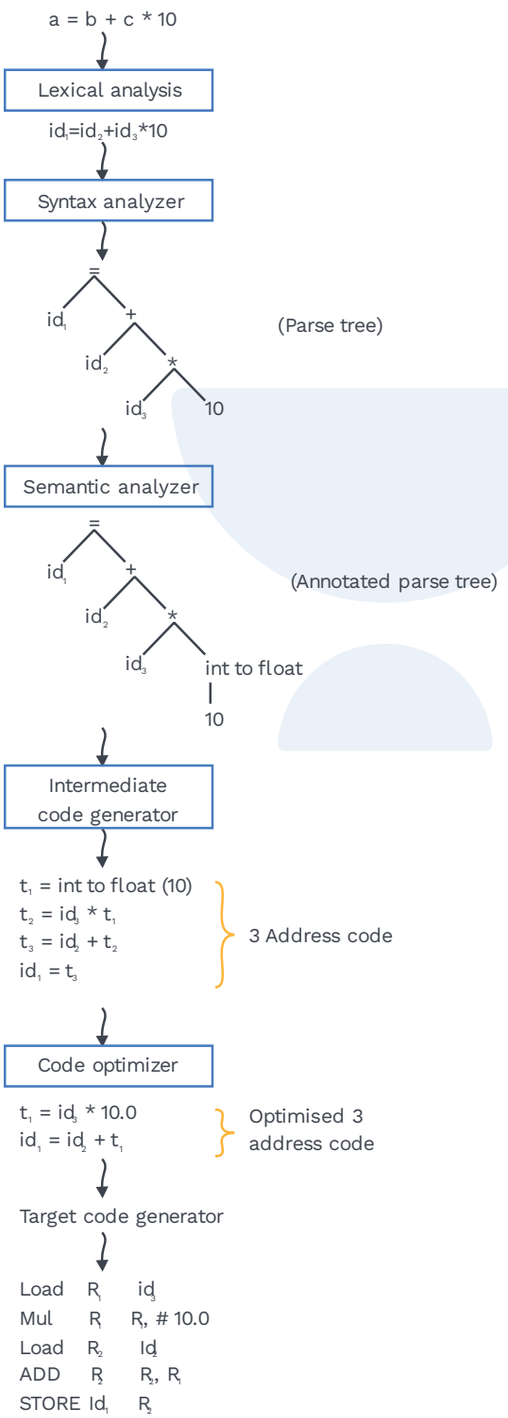


Fig. 1.9 Synthesis Part

SOLVED EXAMPLES

Q1 **Consider the C statement a = b + c * 10, and write the output of each phase after compilation.**

Sol:


a = b + c * 10
Lexical analysis
id1=id2+id3*10
Syntax analyzer
=
id1
+
id2
*
id3
10
(Parse tree)
Semantic analyzer
=
id1
+
id2
*
id3
int to float
10
(Annotated parse tree)
Intermediate code generator
t1 = int to float (10)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
3 Address code
Code optimizer
t1 = id3 * 10.0
id1 = id2 + t1
Optimised 3 address code
Target code generator
Load R1 id3
Mul R1 R1, # 10.0
Load R2 Id2
ADD R2 R2, R1
STORE Id1 R2

Passes:

- A pass is a module that contains sections of one or more phases of a compiler implementation.
- A pass takes the source programme or the output of the previous pass, executes the transformation defined by its phases, and writes the output into an intermediate file that can be read by the next pass.
- If all phases are grouped into one single module is called single-pass compiler.
- A multipass compiler processes the source code of a program multiple times.
- In a multipass compiler, we divide phases into two passes. The front end is considered as the first pass, while the back end is considered as the second pass.
- Multipass compiler, also known as Two-pass compiler.
- A multipass compiler can be developed to take up less space than a single pass compiler since the space taken by the compiler software for one pass can be reused by the following pass.
- Because each pass reads and writes an Intermediate file, a multipass compiler is slower than a single pass compiler.

Data structures in a compiler:

Compiler phases communicate with one another using a shared data structure. These are two crucial tables that are used during the compilation process. They are as follows:

1) Symbol table
2) Literal table

- A symbol table is a table that contains information about the identifiers that were used in the input source program. The strings and constants encountered in the input source program are stored in a literal table.
- The literal table's principal purpose is to save memory by reusing constants and strings.

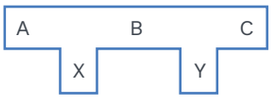
Bootstrapping:

- The source language, the object language, and the language in which it is written are the three languages that define a compiler.
- A cross compiler is a compiler that can run on one machine and create object code for another machine.

Example:

1) Compiler runs on machine X, which takes input language A and produce output language B.

2) Compiler runs on machine X and produces code B for another machine Y.



Chapter Summary

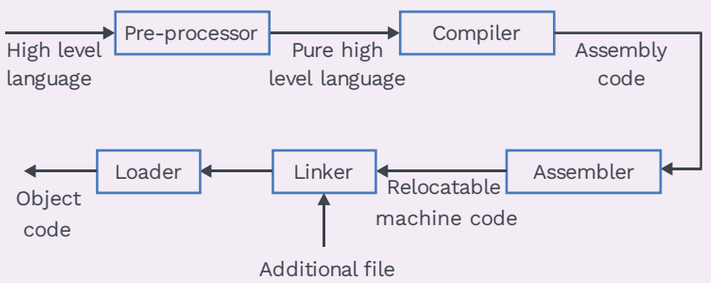


Fig. 1.10 Flow of Program

- Translator translates the source language into the target language.
- Compiler converts high–level language to low–level language.
- Translator is also known as an assembler.
- **Phases of the compiler:**

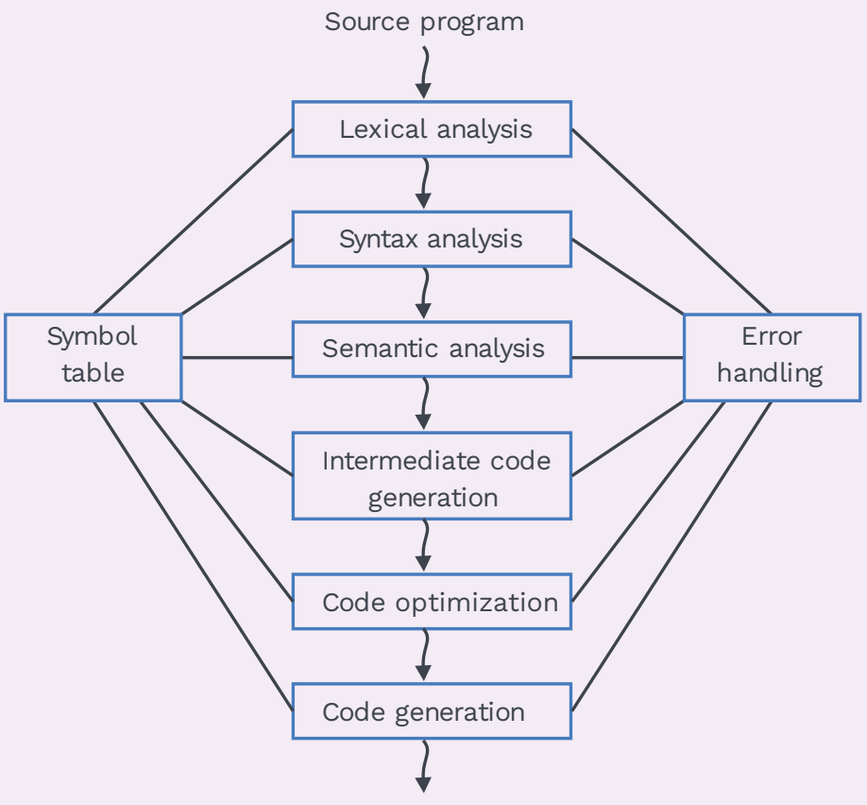


Fig. 1.11 Flow of Program Execution

- Data structures used in the compiler:
- Symbol table
- Literal table

2 Lexical Analysis

Lexical analyzer:

- The lexical analyzer reads the input source program and produces as output a sequence of tokens that the parser uses for syntax analysis.
- The lexical analyzer separates the input into various types of tokens (some meaning full strings) like keywords, Identifiers, special symbols, constants, and operators.
- The part of the input stream that qualifies for a certain type is called a lexeme.
- The lexical analyzer keeps track of newline characters so that it can output the line number with an associated error message.
- The lexical analyzer recognises the comments, white space and stripes out in the source program.
- The lexical analyzer outputs the token that matches the longest possible prefix.
- Lexical analyzer can be implemented with the deterministic finite automata.
- Lexical analyzer, in conjunction with the parser, is responsible for creating symbols to get the next table.
- Lexical analyzer is the first phase of a compiler, which is also known as a scanner.

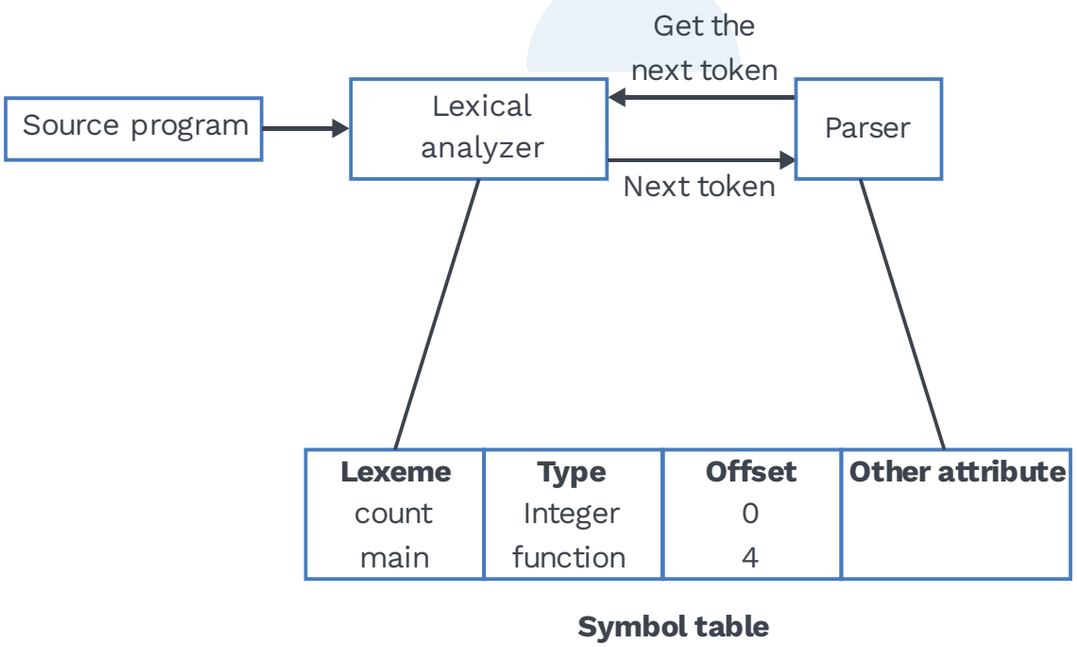


Fig. 2.1 Lexical Analyzer

Keywords:
The table below lists all keywords reserved by the C language.

auto	else	long	switch
break	enum	register	typedef
case	extern	return	union
char	float	short	unsigned
const	for	signed	void
continue	goto	sizeof	volatile
default	if	static	while
do	int	struct	_packed
double			

Table 2.1 List of Keywords

Previous Years' Question

In a compiler, keywords of a language are recognised during
a) Parsing of the program
b) The code generation
c) the lexical analysis of the program
d) Dataflow analysis
Sol: c) **[GATE-CS-2011]**

SOLVED EXAMPLES

Q1 **Find the number of tokens in the following C-program.**

```
int main ( )
  {
    int count ;
    /* This is a comment */
    printf ("Hello world\n");
  }
```

Sol: There are 14 tokens in the above C-program.

S.no.	Lexeme	Token
1)	int	keyword
2)	main	Identifier
3)	(	Special symbol
4)	)	Special symbol
5)	{	Special symbol
6)	int	keyword
7)	count	Identifier
8)	;	Separator
9)	printf	Identifier
10)	(	Special symbol
11)	“Hello world\n”	String literal
12)	)	Special symbol
13)	;	Separator
14)	}	Special symbol

Table 2.2 Lexemes to Token

Rack Your Brain

Find the number of tokens in the following C program

```
Int main ( )
  {
    int i = 0;
    while (i < 4)
  {
    printf (“Hi\n”);
  }
    for (i = 0; i < 8; i++);
  {
    printf (“Bye\n”);
  }
  }
```

Previous Years’ 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

Sol: c) **[GATE: CS-2000]**

Lexical specification and lexical analyzer generator:

- The lexical specification for any programming language consists of information about identifying each and every token that is defined for it. (e.g., identifying keywords, operators, Identifiers, string literals).
- The rules for the basic building blocks of a language are called its language specification.
- A lexical analyzer generator is a tool that can generate a code to perform lexical analysis of the input.
- A notation called regular expression is used to write lexical specifications.
- In order to understand the concept of regular expression, we shall use a utility 'egrep' {extended global regular expression print) available on Linux, Unix platform.
- Lex, Flex, JFlex are commonly available tools for lexical analyzer generators to understand the process of generating the lexical analyzer from lexical specifications.

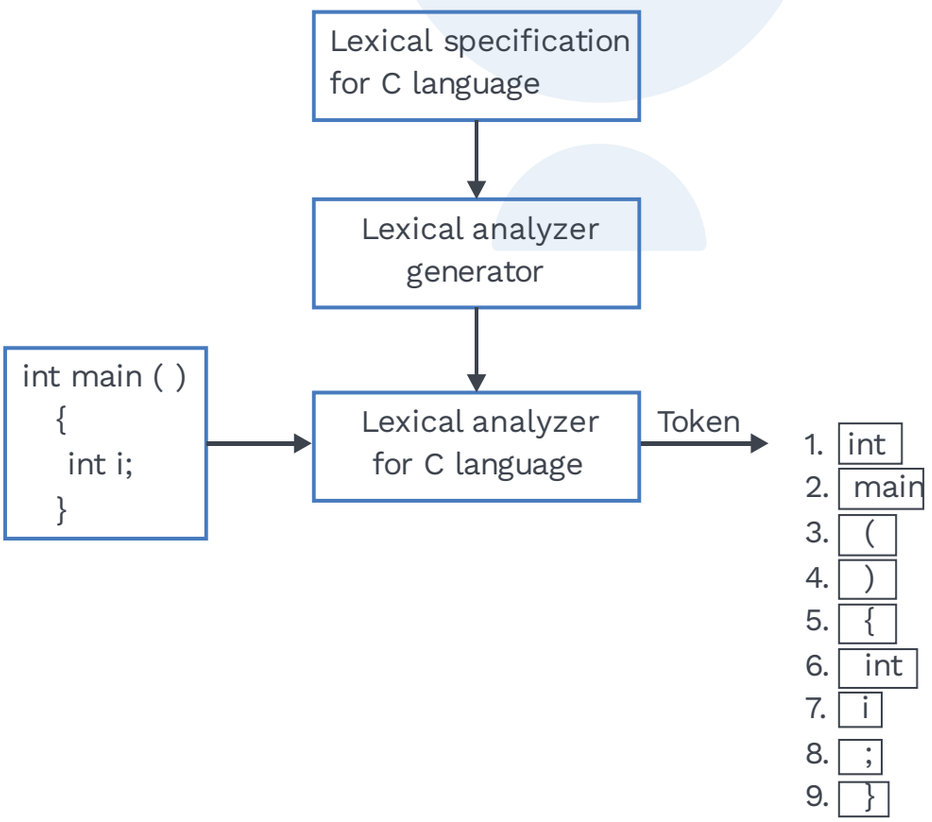
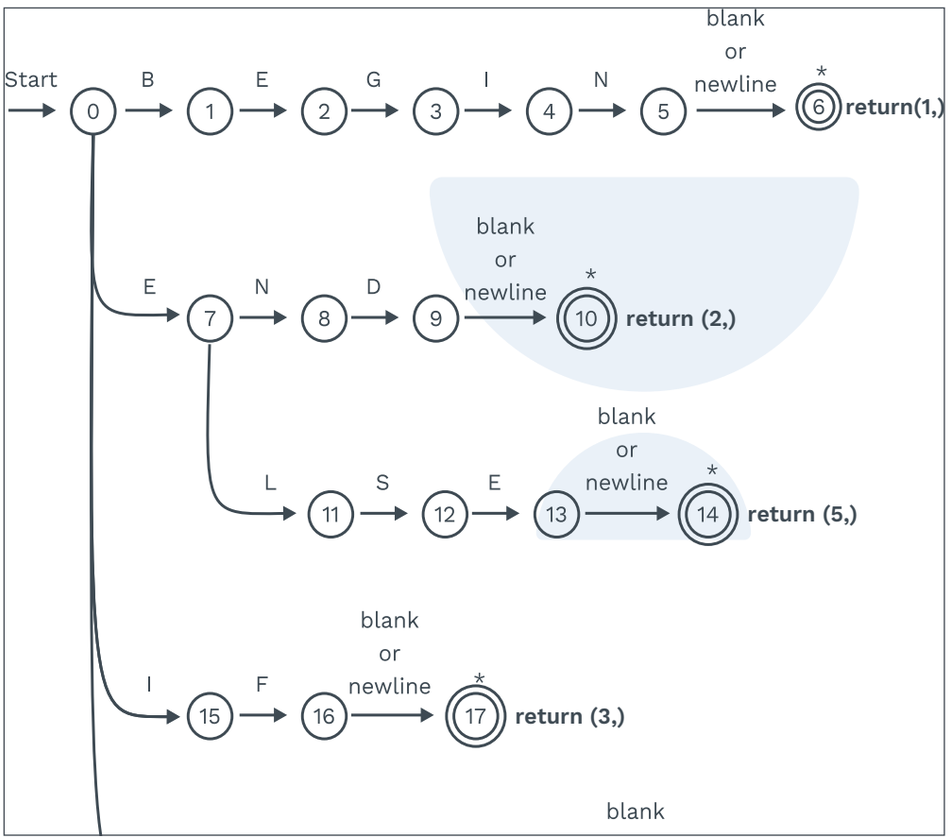


Fig. 2.2 Lexemes to Token Conversion

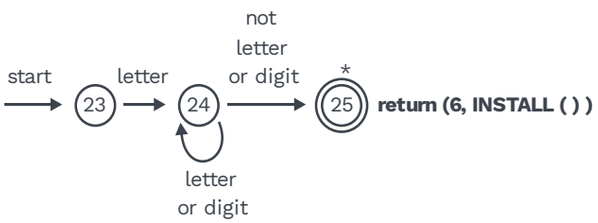
Design of lexical analyzer:

- Designing of any program is to describe the behaviour of the program by a flowchart.
- In lexical analyzer, we use a special kind of flow chart known as transition diagram.

Transition diagram for keywords

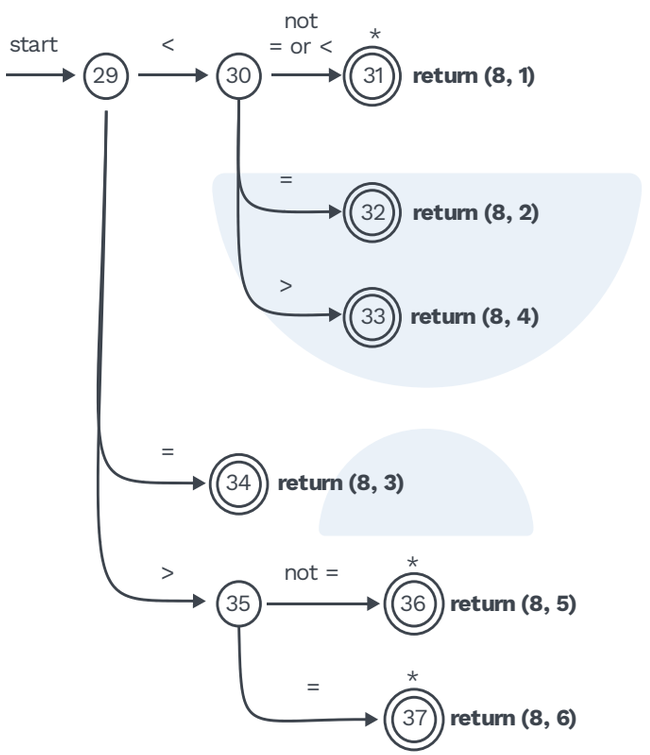


Transition diagram for identifier:



Transition diagram for constant:

Transition diagram for relational operators:



Previous Years' Question

In some programming languages, an identifier is permitted to be later followed by any number of letters or digits. If L and D denote the set of letters and digits, respectively. Which of the following expression defines an identifier?

a) (L + D)* **b)** (L.D)*

c) L(L + D)* **d)** L(L.D)*

Sol: c) **[ISRO-CS-2017]**

Previous Years' Question

A lexical analyzer uses the following patterns to recognise three tokens T1, T2, and T3 over the alphabet {a,b,c}. T1: a?(b|c)*a T2: b?(a|c)*b T3: 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?

a) $T_1T_2T_3$ **b)** $T_1T_1T_3$
c) $T_2T_1T_3$ **d)** T_3T_3

Sol: d) **[GATE: CS-2018]**

LEX tool:

- LEX tool can produce a lexical analyzer automatically.
- A lex source program contains a set of regular expressions and an action for each expression that is basically used to specify the lexical analyzer.
- When a regular expression is recognised the corresponding action of the piece of code is executed.
- LEX gives output as a lexical analyzer program generated from the source specification.

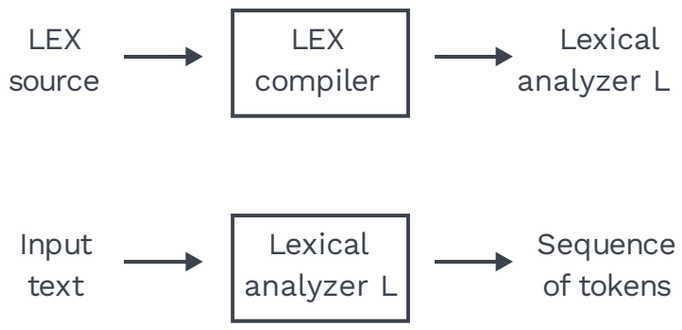
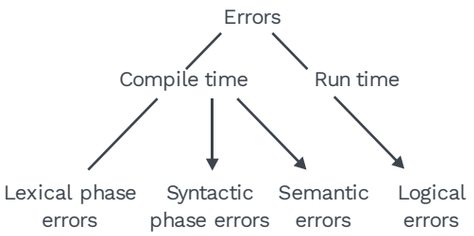


Fig 2.3 The Role of LEX

Error detection in the compiler:

- Compiler detects the errors in the codes made by a programmer and inform, the programmer after compilation. The process of locating an error is known as error detection.
- Error detection and informing the programmer about the error is known as error handling.

Classification of errors:



Lexical phase error:

- Lexical errors are detected during the lexical phase.
- The lexical phase can detect errors when the character remaining in the input does not form any token of the language. Typical lexical errors are:
  1) Appearance of illegal identifiers.
  2) Unmatched strings.
  3) Token in the source program is misspelt.
  4) Exceeding the length of identifiers or numeric constants.

Example:
printf (“prepladder”); $
This is a lexical error since the illegal character $ appears at the end of the statement.

Rack Your Brain

Which of the following gives a lexical error?

a) a123; **b)** 1a;
c) 1 **d)** a_1;

Syntactic phase error:
These errors are detected during syntactic phase Typical syntax errors are:

1) Missing semicolon
2) Unbalanced parentheses
3) Missing operators

Semantic error:
These errors are detected during the semantic analysis phase; typical semantic errors are:

1) Undeclared variables
2) Incompatible type of operands
3) Function computability
4) Type checking.

Logical error:
These errors are detected during runtime. Typical logical errors are:

1) Infinite loop
2) Code not reachable.

Chapter Summary

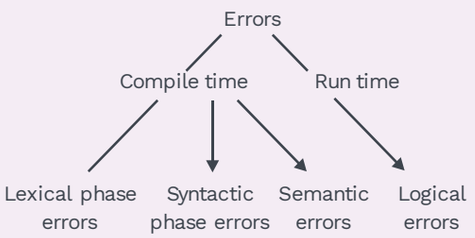
Lexical analyzer:

- The lexical analyzer separates the input into various types of tokens like keywords, identifiers, special symbols and constants operators.
- The lexical analyzer can be implemented with the deterministic finite automata.
- Lex, Flex, JFlex are commonly available tools for lexical analyzer generators to understand the process of generating the lexical analyzer from lexical specifications.

LEX tool:

- A lex source program is a specification of a lexical analyzer, consisting of a set of regular expressions together with an action for each regular expression.

Classification of errors:



Lexical phase error:

1) Appearance of illegal identifiers.
2) Unmatched strings.
3) Token in the source program is misspelt.
4) Exceeding the length of identifiers or numeric constants.

Syntactic phase error:

1) Missing semicolon.
2) Unbalanced parentheses.
3) Missing operators.

Semantic error:

1) Undeclared variables.
2) Incompatible type of operands.
3) Function computability.
4) Type checking.

Logical error:

1) Infinite loop.
2) Code not reachable.

3 Parsing

3. CONTEXT FREE GRAMMAR

In context free grammar, we are using BNF (Backus Normal Form) notation to express the syntactic specification (Rules of language) of a computer language.

- A context free grammar gives a precise, yet easy to understand syntactic specification for the programs of a particular programming language.
- An efficient parser can be constructed automatically from a properly designed grammar.
- A context free grammar imparts a structure to a program that is useful for its translation into object code and for the detection of errors.
- In general, a grammar involves four sets G(V,T,P,S): a set of nonterminal or set of variables (V), a set of terminals(T), a set of production rules(P) and a start symbol(S).

Example: $E \rightarrow EAE \mid (E) \mid -E \mid id$

$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$

V = {A,E}
T = {+ ,-, * , / , ↑ ,(,),id}
S = E
P = {
E → EAE,
E → (E)
E → –E
E → id ,
A → + ,
A → ↑,
A → *,
A → / ,
A → -}

Parse tree:

- A graphical form for derivations that eliminates the replacement order option.
- Tree representation of the derivation is known as the derivation tree / parse tree.
- Parse tree is the hierarchical syntactic structure of a string that is implied by the grammar.
- The leaves of the parse tree are labelled by terminals and read from left to right, they constitute a sentential form called yield or frontier of the tree.
- Each step in the derivation is one sentential form.
- If the derivation is left most, then the sentential form is left sentential form.
- If the derivation is right most, then the sentential form is the right sentential form.

Ambiguous and unambiguous grammar:

- A grammar is ambiguous if it provides multiple parse trees for the same string.
- In another way ∃(there exists) more than one parse tree for an input string.
- Ambiguous grammar is one that produces more than one left most derivation or more than one rightmost derivation for a string.
- A grammar is said to be unambiguous if every string in the language produced by that grammar has a unique parse tree.
- It is desirable for some types of parsers that the grammar be made unambiguous; otherwise, we will not be able to identify which parse tree to select for a string in a unique way.

Example:

Consider the following grammar $S \rightarrow SS / a$, for input w = 'aaa' we have more than one parse tree so, given grammar is ambiguous.

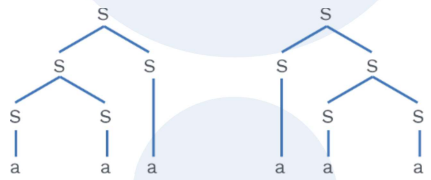


Fig. 3.1

Note

If the grammar is both left recursive and right recursive for a terminal then it is an ambiguous grammar.

Conversion from ambiguous to unambiguous grammar:

- If the grammar is ambiguous, we can describe the associativity and precedence of arithmetic operators to resolve the ambiguity.
- There is no such algorithm to convert all the ambiguous grammar into an equivalent unambiguous grammar. So, this problem is undecidable.
- There is no algorithm that can determine the ambiguity of a CFG, so it is undecidable.

Rack Your Brain

Which of the following grammar is ambiguous:

a) $S \rightarrow SaS \mid SbS \mid \in$

b) $E \rightarrow E + E \mid id$

c) $S \rightarrow SSS \mid a$

d) $E \rightarrow E + id \mid id$

- The ambiguous grammar (for which no other unambiguous grammar exists) produces the same language as the ambiguous grammar is called inherently ambiguous grammar.

SOLVED EXAMPLES

Q.1 **Consider the following grammar for arithmetic expression.**

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

Check the above grammar is ambiguous or not, if ambiguous, then convert into equivalent unambiguous grammar.

Sol:

- The given grammar is ambiguous as the grammar is both left recursive and right recursive. In another way, we can say that we have more than one parse tree for id + id * id.
- In the given grammar, associativity and precedence of operators are not maintained as all operators are at the same level.
- Associativity and precedence of operators are sufficient to disambiguate both grammars.
- The input id + id * id has two distinct leftmost derivations with the corresponding parser tree.

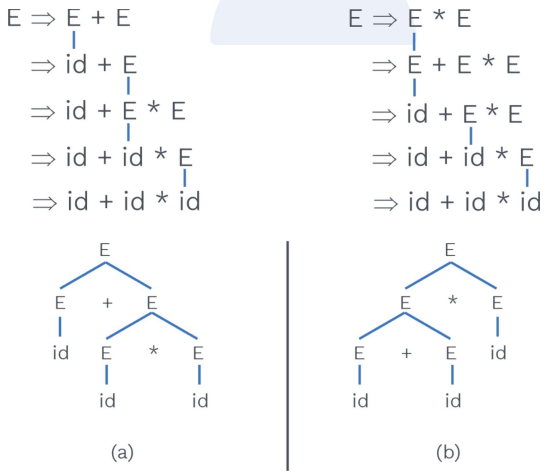


Fig. 3.2 Derivation Trees

- The parse tree of figure 'a' is correct in that it reflects the commonly assumed precedence of + and * while the figure 'b' does not.
- According to C language rules, '*' as having higher precedence than '+' so '*' is evaluated first, then '+'.

- If we take input as 2 + 3 * 4 in the first parse tree, we will get the correct output i.e. 14.
- As * and + have left associative, id * id * id means (id * id) * id and id + id + id implies (id + id) + id.
- The final unambiguous grammar is:

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

Previous Years' Question

Consider the grammar defined by the following production rules, with two operators * and +

$$S \rightarrow T * P$$

$$T \rightarrow U \mid T * U$$

$$P \rightarrow Q + P \mid Q$$

$$Q \rightarrow Id$$

$$U \rightarrow Id$$

Which one of the following is TRUE?

a) + is left associative while * is right associative
b) + is right associative, while * is left associative
c) Both + and * right associative
d) Both + and * are left associative

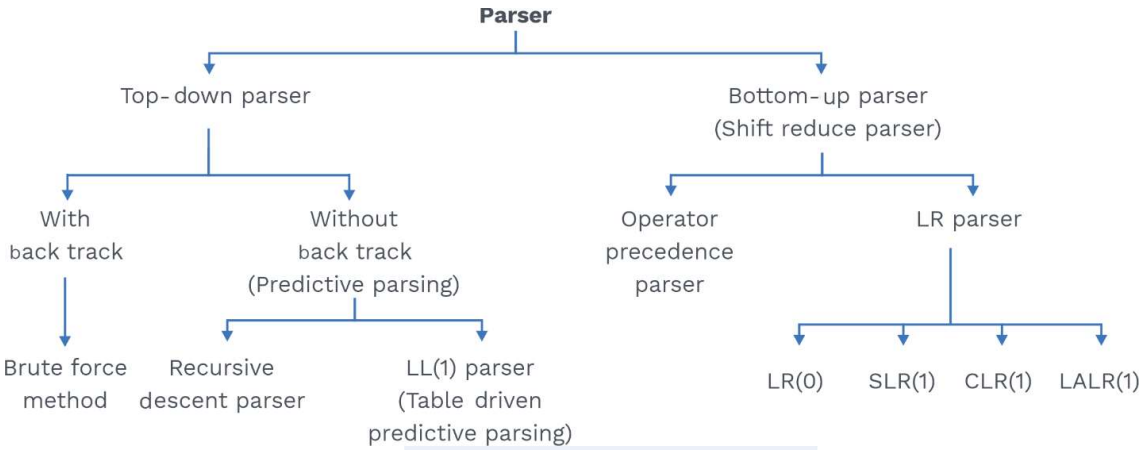
Sol: b) **[GATE: CS 2014]**

3.1 PARSER

- For a grammar G, parser is a program that accepts a string as input and produces output as a parse tree.
- Syntax analyzers are also referred to as parsers.
- There are two types of parsers: top-down and bottom-up.

Rack Your Brain

Convert the given ambiguous grammar into unambiguous grammar.
Boolean expression → not (Boolean expression) / Boolean expression and Boolean expression / Boolean expression or Boolean expression / True / False

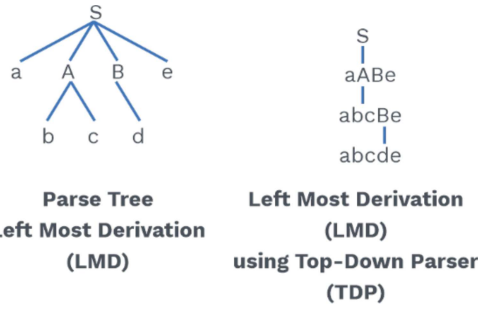


Top-down parser(TDP):

- The top-down parsers build the parse tree starting from the root node and work down to the leaf nodes.
- The input to the parser is scanned from left to right, one symbol at a time, in a top-down parser.
- For example, consider the grammar, draw the parse tree for input 'abcde' using top-down parser

$S \rightarrow aABe$
$A \rightarrow bc / b$
$B \rightarrow d$

- Top-down parser follows Left Most Derivation (LMD).



Parse Tree Left Most Derivation (LMD)

Left Most Derivation (LMD) using Top-Down Parser (TDP)

Fig. 3.3 Derivation Tree

Bottom-up parser:

- The bottom-up parsers construct a parse tree from the leaves (strings) and proceed to start symbol.

- In bottom-up parser the input to the parser is being scanned from left to right, one symbol at a time.
- A bottom-up parser builds a parse tree by reversing the rightmost derivation.
- At each stage, a string matching the right side of a production is replaced with the symbol on the left in a bottom-up parser.
- For example, consider the grammar.

$$S \rightarrow aAcBe$$
$$A \rightarrow Ab / b$$
$$B \rightarrow d$$

Draw the parse tree for input abbcde using bottom-up parser.

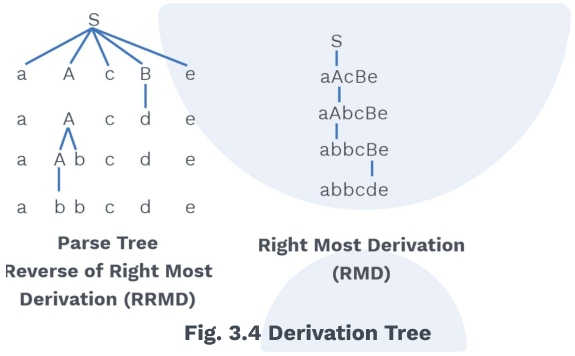


Fig. 3.4 Derivation Tree

Difference between top-down parser and bottom-up parser:

Top-down Parser	Bottom-up Parser
1) A parse tree is created from the root (top) to leaves (Bottom). **2)** The traversal of parse tree is a preorder traversal. **3)** It uses left most derivation. **4)** Expansion of tree. **5)** Less powerful.	**1)** A parse tree is created from the leaves (Bottom) to root (Top). **2)** The traversal of parse tree is a reversal of post order traversal. **3)** It is the rightmost derivation , in the reverse order. **4)** Reduction of tree. **5)** More powerful than topdown parser.

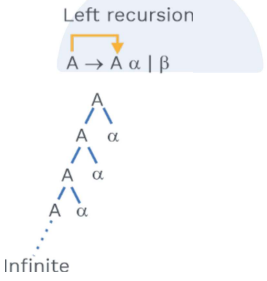
Handles:

- Reduction is the process of replacing the right side of the production with the left side of the production.
- The substring that gets replaced by its left side of the production is called a handle.
- In the previous example, “abbcde” is reduced to “abbcBe” where “d” is the handle as in the grammar, we have $B \rightarrow d$. Again “abbcBe” is reduced to “aAbcBe” where “b” is a handle as in the grammar, we have $A \rightarrow b$.
- If grammar is unambiguous, it has precisely one handle for each right sentential form.

Recursive and non-recursive grammar:

- A grammar can be classified into recursive and nonrecursive grammar.
- In recursive grammar $\exists$(there exists) atleast one production has the same variable in LHS and RHS. For e.g. $S \rightarrow Sa \mid b$.
- In non recursive grammar, no production has the same variable in LHS and RHS.
- Recursive grammar is further classified into left recursive and right recursive grammar.
- If grammar has a nonterminal A, it is left recursive, if there is a derivation $A \rightarrow A\alpha$ (Where A is at left most place) for some α.
- A left recursive grammar can cause a topdown parser to go into an infinite loop.

For example:



Elimination of left recursion:

Consider the left recursive pair of productions where the left recursion can be eliminated by replacing the pair of production with the following rules:

Rule 1: $A \rightarrow A\alpha \mid \beta$ is replaced by

$$A \rightarrow \beta A'$$

$$A' \rightarrow \in \mid \alpha A'$$

Rule 2: $A \rightarrow A\alpha \mid \beta_1 \mid \beta_2 \mid \beta_3 ..... \mid \beta_n$ is replaced by

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' ..... \mid \beta_n A'$$

$$A' \rightarrow \in \mid \alpha A'$$

Rule 3: $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 ..... \mid A\alpha_n \mid \beta$ is replaced by

$$A \rightarrow \beta A'$$

$$A' \rightarrow \in \mid \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' ..... \mid \alpha_n A'$$

Rule 4: $A \rightarrow A\alpha_1 \mid A\alpha_2 .....A\alpha_n \mid \beta_1 \mid \beta_2 .....\beta_n$ is replaced by

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' ..... \mid \beta_n A'$$

$$A' \rightarrow \in \mid \alpha_1 A' \mid \alpha_2 A' ..... \mid \alpha_n A'$$

SOLVED EXAMPLES

Q.2 **Remove the left recursion from the grammar below.**

$A \rightarrow A + B \mid B$

$B \rightarrow B * C \mid C$

$C \rightarrow (D) \mid d$

Sol: Using rule 1

$$A \rightarrow BA'$$

$$A' \rightarrow \in \mid +BA'$$

$$B \rightarrow CB'$$

$$B' \rightarrow \in \mid *CB'$$

$$C \rightarrow (D) \mid d$$

Previous Years' Question

Which one of the following grammars is free from left recursion?

a) $S \rightarrow AB$
$A \rightarrow Aa \mid b$
$B \rightarrow c$

b) $S \rightarrow Ab \mid Bb \mid c$
$A \rightarrow Bd \mid \in$
$B \rightarrow e$

c) $S \rightarrow Aa \mid B$
$A \rightarrow Bb \mid Sc \mid \in$
$B \rightarrow d$

d) $S \rightarrow Aa \mid Bb \mid c$
$A \rightarrow Bd \mid \in$
$B \rightarrow Ae \mid \in$

a) a **b)** b **c)** c **d)** d

Sol: b)

[GATE: CS 2016 (Set-2)]

Left factoring:

- Consider a production $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ and an input that begins with a non-empty string derived from α, it is not clear whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$.
- We can solve this by left factoring

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

- The grammar that is suitable for predictive parsing (after elimination of left recursion and left factoring) are called LL(K) grammars. Where first 'L' implies that we can scan from left to right, the second L stands for leftmost derivation, and K stands for the number of tokens of look ahead.

Rack Your Brain

Eliminate the left recursion from the following grammar:

$$E \rightarrow AaB$$
$$A \rightarrow aA \mid Ba$$
$$B \rightarrow AB \mid b$$

SOLVED EXAMPLES

Q.3 **Consider the following grammar**

$A \rightarrow aAb \mid aAc \mid d$

$A \rightarrow aA \mid \in$

$B \rightarrow bB \mid \in$

Remove left factoring?

Sol: Consider the following grammar

$$A \rightarrow aAb \mid aAc \mid d$$

Write down the left factored form of the above grammar.

$$A \rightarrow aAA' \mid d$$
$$A' \rightarrow b \mid c$$

Recursive-descent parser:

- A parser that uses a set of recursive procedures to recognize its input with no backtracking is called recursive-descent parser.
- Recursive descent parsers are simple and easy to implement.
- A recursive descent parser needs a larger stack due to the recursion of the procedures.

Predictive parser / LL(1) parser:

- By explicitly preserving the stack, a predictive parser is an efficient approach for achieving recursive-descent parsing.
- An input buffer, a stack, a parsing table, and an output are all included in the predictive parser.

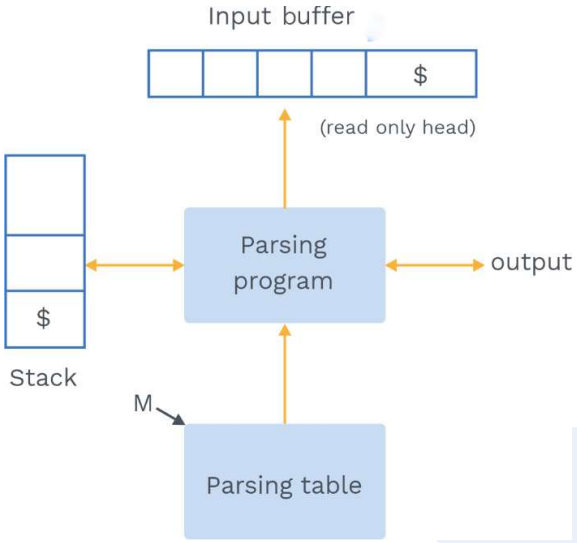


Fig. 3.5 Parser Diagram

Rack Your Brain

Consider the following grammar

1) S → iEts | iEtSes | a

E → b

2) S → assbs | asasb | asb | a

Write down the left factored form of the above grammar.

Model of predictive parsers / LL(1) parser:

- The string to be parsed is placed in an Input buffer, followed by $ (a symbol that indicates the end of the input).
- The sequence of grammar symbols in a stack, with $ at the bottom of the stack marker, is called a stack.
- The stack starts with the grammar's start symbol, which is preceded by $.
- The grammar's start symbol, which is preceded by $, is at the top of the stack.
- The parsing table is a two-dimensional array M [A, a], where A denotes non-terminal, and a denotes the terminal symbols including $.

Predictive parsing program / LL(1) algorithm:

If a is the current input symbol and X is the top of the stack.

1) If X = a = $, the parser halts and announces successful completion of parsing.

2) If X = a ≠ $, then pop-top of stack and increment input pointer.

3) If X is a nonterminal, the program consults the parsing table M's entry M[X, a]. This entry will be either a grammar production or an error entry. If M [X, a] = {X → uvw}, the parser substitutes wvu for X at the top of the stack (with u on top of stack)

4) If M[X, a] = blank then error.

First and follow:

We require two functions connected with a grammar 'G' to populate the entries of a predictive parsing table. FIRST() and FOLLOW() are these functions.

Rules for computing first:

The set of all the terminals that X can begin with is the FIRST set of non-terminal X.

Rule 1: If X is a terminal, then FIRST(X) = {X}

Rule 2: For a non-terminal X, if there exists a production $X \rightarrow \in$, then add $\in$ to FIRST(X).

Rule 3: For a production $X \rightarrow Y_1Y_2Y_3.....Y_k.....Y_n$

FIRST(X) = FIRST(Y_1), if FIRST(Y_1) does not contain $\in$

= FIRST(Y_1) U FIRST(Y_2), if FIRST(Y_1) contains $\in$ and FIRST (Y_2) does not contain $\in$

= FIRST(Y_1) U FIRST(Y_2) U FIRST(Y_3), if FIRST(Y_1), FIRST(Y_2) both contain $\in$ and FIRST(Y_3) does not contain $\in$.

Generalising:

FIRST(X) = FIRST(Y_1) U FIRST(Y_2) U FIRST(Y_3) U FIRST(Y_k) if FIRST(Y_1), FIRST(Y_2) ..... FIRST(Y_{k-1}) all contain $\in$ and FIRST(Y_k) does not contain $\in$.

FIRST(X) = FIRST(Y_1) U FIRST(Y_2) U FIRST(Y_3) U..... FIRST(Y_n) U {$\in$} if FIRST(Y_1), FIRST(Y_2) ..... FIRST(Y_n) all contain $\in$

Rules for computing follow:

A NON-TERMINAL SET OF FOLLOWING A is the collection of all terminals that can be followed by A.

Rule 1:

1) FOLLOW(S) = $, where S is the start symbol and $ is the symbol to indicate the end of input.

2) If the Grammar is

$$S \rightarrow \alpha AB \qquad B \not\rightarrow \in$$

Then FOLLOW (A) = FIRST (B)

3) If the Grammar is

$$S \rightarrow \alpha B$$

FOLLOW (B) = FOLLOW (S)

4) If the Grammar is

$S \rightarrow \alpha AB \qquad B \rightarrow \in$

FOLLOW (A) = {FIRST(B)– $\in$} $\cup$ {FOLLOW(S)}

Note

FOLLOW of a non-terminal does not contain $\in$.

SOLVED EXAMPLES

Q.4 **Find FIRST and FOLLOW set of the following grammar.**
$S \rightarrow AB$; $A \rightarrow aA \mid \in$; $B \rightarrow bB \mid \in$

Sol:

	FIRST	FOLLOW
S	a, b, $\in$	$
A	a, $\in$	b, $
B	b, $\in$	$

LL(1) Parsing table construction algorithm:

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 FIRST(α) contain $\in$, add $A \rightarrow \alpha$ to M[A, x] for each terminal x in FOLLOW (A).
If $\in$ is in FIRST(α) and $ is in FOLLOW(A), add $A \rightarrow \alpha$ to M[A, $]

4) Make each undefined entry of M as error.

Rack Your Brain

Find FIRST and follow set of the following grammar:

$S \rightarrow AbB \mid CdB \mid Ba$

$A \rightarrow ea \mid BC$

$B \rightarrow f \mid \in$

$C \rightarrow g \mid \in$

SOLVED EXAMPLES

Q.5 **Suppose the input is "id + id * id", to parse this input construct LL(1) parsing table considering the following grammar.**

E → TE'

E' → +TE' |∈

T → FT'

T' → *FT' |∈

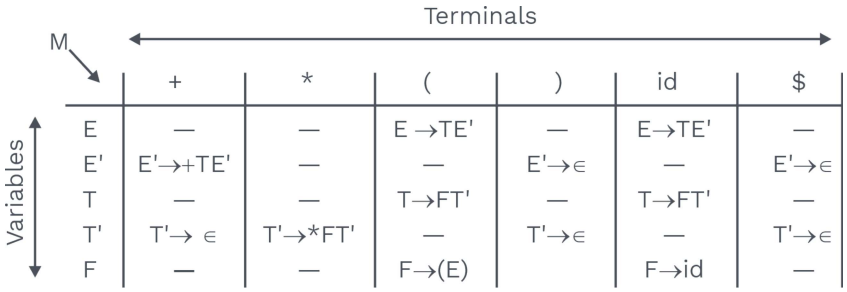
F → (E) | id

and parse the input id + id * id

Sol:

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

- Add E → TE', FIRST(TE') = FIRST(T) = {(, id}
- Production E → TE' causes M[E, ( ] and M[E, id] to acquire the entry E → TE'
- Production E' → +TE' causes M[E', +] to acquire E' → +TE'
- Production E' →∈ causes M[E',)] and M[E', $] to acquire E' →∈ since FOLLOW(E') = {), $}
- Production T → FT' causes M[T, ( ] and M[T, id] to acquire T → FT' since FIRST(F) = {(, id}
- Production T' → *FT' causes M[T', *] to acquire T' → *FT'
- Production T' →∈ causes M[T, +], M[T,)] and M[T, $] to acquire T' → ∈ since FOLLOW(T') = {+,), $}
- Production F → (E) causes M[F, (] to acquire F → (E)
- Production F → id causes M[F, id] to acquire F → id



All the entries in LL(1) table contain single entries so the grammar is LL(1). Now with the help of LL(1) parsing table, we can parse input id + id * id.

Moves by predictive parser:

Stack	Input	Output
$E	id+id*id$	
$E'T	id+id*id$	E → TE'
$E'T'F	id+id*id$	T → FT'
$E'T'id	id+id*id$	F → id
$E'T'	+id*id$	
$E'	+id*id$	T' → ∈
$E'T+	+id*id$	E' → +TE'
$E'T	id*id$	
$E'T'F	id*id$	T → FT'
$E'T'id	id*id$	F → id
$E'T'	*id$	
$E'T'F*	*id$	T' → * FT'
$E'T'F	id$	

Stack	Input	Output
$E'T'id	Id$	F → id
$E'T'	$	
$E'	$	T' → ∈
$	$	E' → ∈

Table 3.1

Previous Years' Question

Consider the following grammar:

$$S \rightarrow FR.$$

$$R \rightarrow *S \mid \in$$

$$F \rightarrow id$$

In the predictive parser table, M, of the grammar the entries M[S,id] and M[R, $] respectively

a) $\{S \rightarrow FR\}$ and $\{R \rightarrow \in\}$

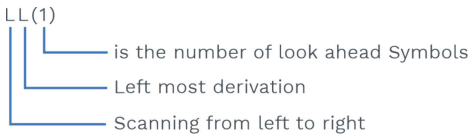
b) $\{S \rightarrow FR\}$ and $\{\ \}$

c) $\{S \rightarrow FR\}$ and $\{R \rightarrow *S\}$

d) $\{S \rightarrow id\}$ and $\{R \rightarrow \in\}$

Sol: a) **[GATE: CS 2006]**

Checking for LL(1) grammar:



- The LL(1) grammar is unambiguous, non left recursive and left factored.
- To determine whether a grammar is LL(1) or not:

1) If $A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3$ then

$$\left.\begin{array}{l} FIRST(\alpha_1) \cap FIRST(\alpha_2) = \phi \\ FIRST(\alpha_2) \cap FIRST(\alpha_3) = \phi \\ FIRST(\alpha_1) \cap FIRST(\alpha_3) = \phi \end{array}\right| \text{Pairwise disjoint}$$

Then the grammar is LL(1)

2) If $A \rightarrow \alpha_1 \mid \alpha_2 \mid \in$ then

$$\left.\begin{array}{l} FIRST(\alpha_1) \cap FIRST(\alpha_2) = \phi \\ FIRST(\alpha_2) \cap FOLLOW(A) = \phi \\ FIRST(\alpha_1) \cap FOLLOW(A) = \phi \end{array}\right| \text{Pairwise disjoint}$$

Then the grammar is LL(1)

- Grammar is considered to be LL(1) if it's LL(1) the parsing table is free from multiple entries.
- A left recursive grammar can not be a LL(1) grammar.
- An ambiguous grammar can not be LL(1) grammar.
- If a grammar has a common prefix, it cannot be LL(1).

Rack Your Brain

Construct the LL(1) parsing table for the following grammar

$$S \rightarrow (L) \mid a$$
$$L \rightarrow SL'$$
$$L' \rightarrow \in \mid SL'$$

And parse the input string (a, a, a).

Rack Your Brain

Which of the following is true.

1) Every ambiguous grammar is not LL(1)

2) Every unambiguous grammar need not be LL(1)

3) Every LL(1) grammar is unambiguous

4) None

Previous Years' Question

The grammar $A \rightarrow AA \mid (A) \mid \in$ is not suitable for predictive parsing because the grammar is

a) Ambiguous
b) Left-recursive
c) Right-recursive
d) An operator-grammar

Sol: a) **[GATE: CS 2005]**

Shift reduce parser:

- The shift reduce parser attempts to construct a parse tree in the same way that bottom-up parsing does.
- In other words, the parse tree is built from leaf (bottom) to root (top).
- LR parser is a more general form of shift reduce parser.
- The shift reduce parser required a data structure known as a stack.
- A stack that stores and retrieves the production rule.
- An input buffer to store the input string.

Basic operations in shift reduce parsing:

There are four actions possible in shift reduce parser:
Shift: The top of the stack will be updated with the next input symbol.
Reduce: If the handle appears on top of the stack, then it reduced by using appropriate production rule is done i.e., right hand of production rule is popped out of the stack, and the left hand of a production rule is pushed on to the stack.
Accept:Parser will consider a parsing as a successful parsing if and only if the start symbol is present in the stack and the input buffer is empty.
Error: The ERR("error recovery routine") will be triggered by the parser if there is a syntax error, and during this the parser stops shift and reduces operations.

Example:

Consider the grammar

$$A \rightarrow A + A$$
$$A \rightarrow A * A$$
$$A \rightarrow id$$

and parse the input string $id_1 + id_2 + id_3$ using shift reduce parser and find total number of shifts and reduce operation.

Stack	Input	Action
$	$id_1+id_2+id_3\$$	Shift
$\$id_1$	$+id_2+id_3\$$	Reduce by A → id
$A	$+id_2+id_3\$$	Shift
$A+	$id_2+id_3\$$	Shift
$A+id_2$	$+id_3\$$	Reduce by A → id
$A+A	$+id_3\$$	Shift
$A+A+	$Id_3\$$	Shift
$A+A+id_3$	$	Reduce by A → id
$A+A+A	$	Reduce by A → A+A
$A+A	$	Reduce by A → A+A
$A	$	Accept

Table 3.3

Total shift operation = 5
Total reduce operation = 5

Rack Your Brain

Consider the grammar

$A \rightarrow A + A$
$A \rightarrow A * A$
$A \rightarrow id$

Perform shift reduce parsing for input string $id_1+id_2*id_3$ and find total number of shift and reduce operation.

Previous Years' Question

Which one of the following derivations is used by LR parsers?

a) Leftmost
b) Leftmost in reverse
c) Rightmost
d) Rightmost in reverse

Sol: d) **[GATE: CS 2019]**

LR parser:

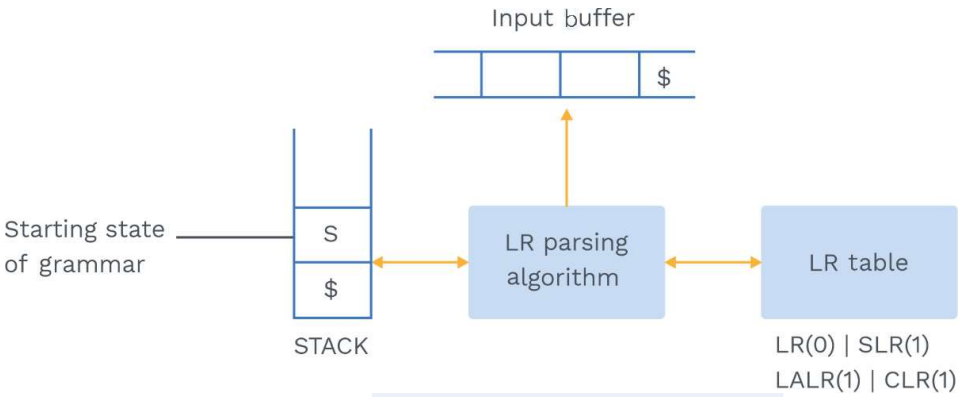
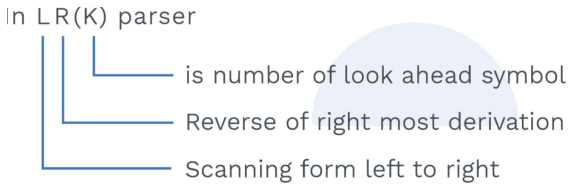


Fig. 3.6 Parser Diagram

- LR parser can be built for an unambiguous grammar.
- An operator precedence can be built for both ambiguous and unambiguous grammar.



- Bottom–up parser's average complexity is $O(n^3)$
- Handle pruning is a significant overhead for a bottom-up parser.
- Parsing algorithm is same for all LR parsers but parsing table may differ.

LR parsing algorithm:

If S is TOP of stack and a is look ahead symbol.

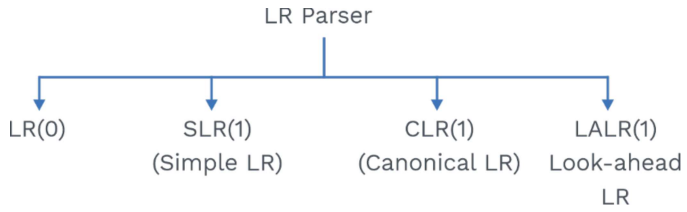
1) If action [S, a] = S_i (shift) then shift a and i and Increment input pointer.

2) If action [S, a] = r_j (reduce) and r_j is $\alpha \rightarrow \beta$, then pop $2 \times |\beta|$ from top of stack and replace by α.
S_{m-1} is the state below α, then push goto $[S_{m-1}, \alpha]$

3) If action [S, a] = ACCEPT then successful completion of parsing.

4) If action [S, a] = blank then error.

Types of LR parser:



LR(0) and SLR(1):

- LR(0) and SLR(1) work on smallest class of grammar.
- LR(0) and SLR(1) contain few number of states; hence, the table size is small.
- LR(0) and SLR(1) are simple and fast to construct.

CLR(1):

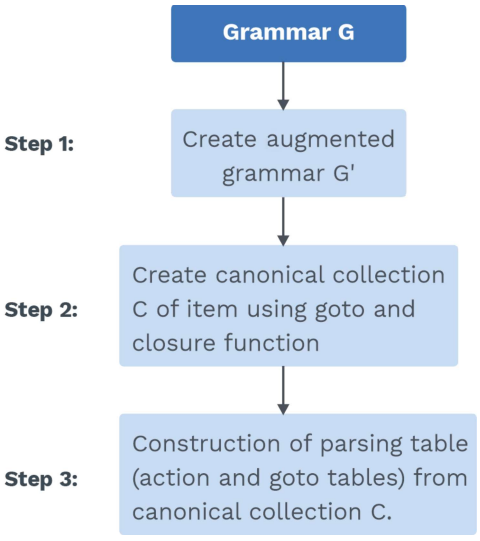
- CLR(1) works on complete set of LR(1) grammar.
- CLR(1) contain large number of states; hence, table size is large.
- CLR(1) is slow construction.

LALR(1) parser:

- LALR(1) works on intermediate size of grammar.
- Number of states are same as in SLR(1) and LR(0).

Constructing an LR parsing table:

- To generate the parsing table from the grammar, three steps are required.



- In the first step, an extra production rule is added to the original set of production to create an augmented grammar G. All LR parsers i.e., LR(0), SLR(1), CLR(1), LALR(1) follow the same procedure.
- In the second step of creating the canonical collection C of entities called items, the LR(0) item, while CLR(1) and LALR(1) use a more specialized entity called LR(1) item.
- LR(0) and SLR(1) parser has different closure and goto functions compared to CLR(1) and LALR(1).
- In step three, we convert the canonical collection of a set of items into the parsing table by applying certain Rules.

Augmented grammar:

If G is a grammar with the start symbol S, then the augmented grammar G' for G consists of all productions in G and an additional production $S' \rightarrow S$. Where S' is the start symbol in G'.

The main use of augmented grammar lies in the fact that the acceptance of input happens when the parser is about to reduce production $S' \rightarrow S$.

LR(0) item and LR(1) item:

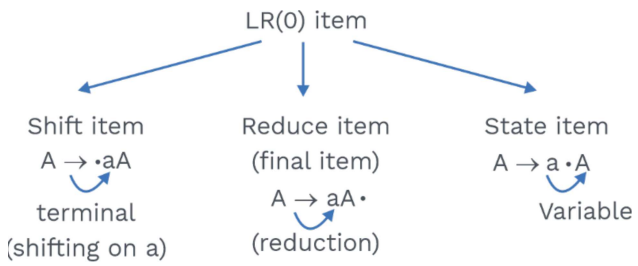
- An LR(0) item (or simply item) of a grammar G is production of G with a dot in some position of right-hand side.
- For a grammar consisting of a production $S \rightarrow ABC$, the LR(0) items are:

$$S \rightarrow \cdot ABC$$
$$S \rightarrow A \cdot BC$$
$$S \rightarrow AB \cdot C$$
$$S \rightarrow ABC \cdot$$

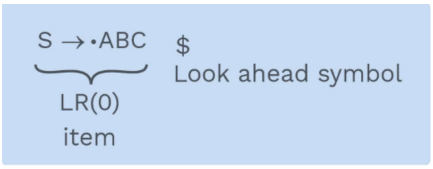
- For production $S \rightarrow \in$, there is only one LR(0) item.

$$S \rightarrow \cdot$$

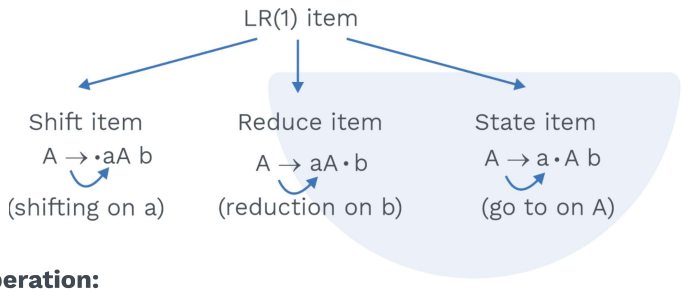
- Dot(.) can be used to show the portion of input that is already consumed. If the dot moves to the rightmost point of the production, we can use it to reduce that production.



- An LR(1) item of a grammar G is LR(0) item and look ahead symbol.
- For e.g.



LR(1) items



Closure operation:

If I is a set of items for a grammar G, then closure(I) is the set of items formed from I by the following two rules.

Rule 1: Every element of I is added to the closure of I.

Rule 2: If $S \rightarrow A \cdot BC$, is in closure(I) and there exists a production $B \rightarrow b$, then add the item $B \rightarrow \cdot b$ to I, if it is not already in closure(I). Keep applying this rule until there are no more elements added.

Example:

$$A' \rightarrow A$$

$$A \rightarrow aA \mid b$$

$\text{Closure}(A' \rightarrow A) = A' \rightarrow A$ ('A' is non-terminal so add production of A)

$A \rightarrow \cdot aA$

$\cdot b$

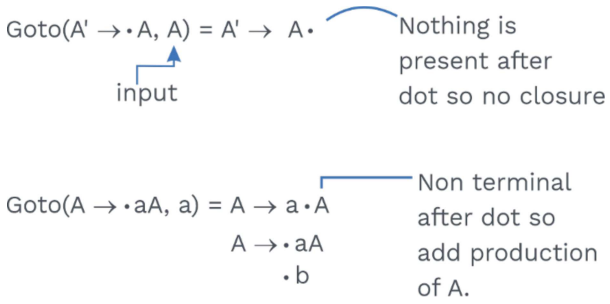
Goto operation:

Goto (I, X), I is the set of items which the goto (I, X) needs to be computed.

1) Add I by moving dot after X.

2) Apply closure to step 1.

Example:



LR(0) Parsing table construction algorithm:

Let C = $\{I_0, I_1, ...I_n\}$ the states of the parser are 0, 1, ...n state i being constructed from I_i the parsing action for state I are determined as follows.

1) If goto $(I_i, a) = I_J$, then set ACTION[i, a] = S_J (Shift entry)

2) If goto $(I_i, A) = I_J$ then set ACTION[i, A] = J (State entry)

3) If I_J contain $A \rightarrow \beta\cdot$ (reduced production) then action [i, all entries] = R_P (reduce entry) where P is a production number $(P : A \rightarrow \beta\cdot)$

4) If $[A' \rightarrow A\cdot]$ is in I_i then set action [i, $] to accept.

Note

If any conflicts are generated due to the above rules, we say grammar is not LR(0). The algorithm fails to produce a valid parser in this case. If a state contains either SR or RR conflicts, then that state is called inadequate state.

LR(0) conflicts :

There are two types of conflict in LR(0)

1) Shift-reduce conflict (SR conflict)

- If both shift and reduced productions are in same state.
  For e.g.

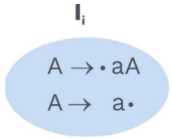


Fig. 3.7 SR Conflict

2) Reduce-reduce conflict (RR conflict):
If two reduce production (reduced items) are present in the same state.

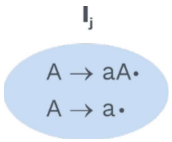


Fig. 3.8 Caption RR Conflict

Note

If any state has LR(0) conflict, then it is not LR(0) grammar.

SLR(1) parsing table construction:

- SLR(1) parsing table construction is same as LR(0) except for reduction entries.
- If $[A \rightarrow \beta \cdot]$ is in I_i, then find FOLLOW(A) and for every a in FOLLOW(A), set action [i, a] = R_P where P is a production number.
- Number of shift entries is same in LR(0) and SLR(1).
- Size of SLR(1) parser table is same as size of LR(0) parser table.
- Number of state entry is also same in LR(0) and SLR(1).
- Number of states is same in LR(0) and SLR(1).
- Every LR(0) grammar is SLR(1) but every SLR(1) need not be LR(0).

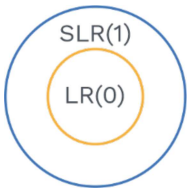


Fig. 3.9 Parsers

Conflict in SLR(1) parser:

1) Shift-reduce conflict (SR conflict): State I_i shift reduce conflict if FOLLOW $(B) \cap \{\alpha\} \neq \phi$

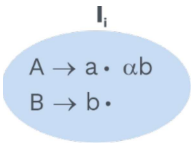


Fig. 3.10 SR Conflict

2) Reduce-reduce conflict:

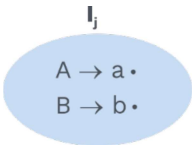


Fig. 3.11 Caption RR Conflict

State I_j is reduce-reduce conflict if FOLLOW (A) $\cap$ FOLLOW (B) $\neq \phi$

SOLVED EXAMPLES

Q.6 **Construct LR(0) and SLR(1) parsing table for the following grammar.**

$A \rightarrow aA$

$A \rightarrow b$

Sol: **Step1:** Construct augmented grammar

$A' \rightarrow A$
$A \rightarrow aA$
$A \rightarrow b$

Step2: Make LR(0) item and use closure and goto to construct a DFA.

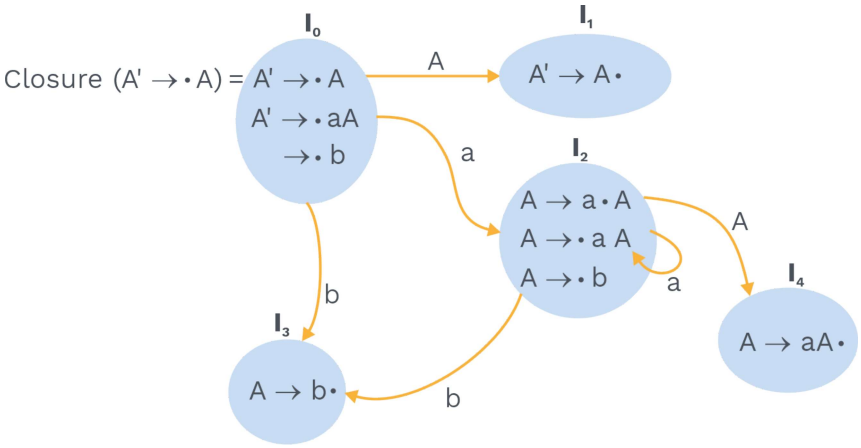


Fig. 3.12 Canonical form of LR Items

Step3: Construction of LR(0) parsing table

numbering for reduce :
$A \rightarrow aA$ ..... (1)
$A \rightarrow b$ ..... (2)

	Action			Goto
	a	b	$	A
I_0	s_2	s_3	—	1
I_1	—	—	accept	—
I_2	s_2	s_3	—	4
I_3	r_2	r_2	r_2	—
I_4	r_1	r_1	r_1	—

LR(0) / SLR(1) parsing table

As there is no conflict so, the given grammar is LR(0). (if the grammar is LR(0). It will definitely SLR(1))

Q.7 **Construct LR(0) and SLR(1) parsing table for the following grammar.**
$A \rightarrow a \mid ab \mid bB$
$B \rightarrow c$

Sol: **Step1:** Construct augmented grammar

$A' \rightarrow A$
$A \rightarrow a \mid ab \mid bB$
$B \rightarrow c$

Step2: Make LR(0) item and use closure and goto to construct a DFA.

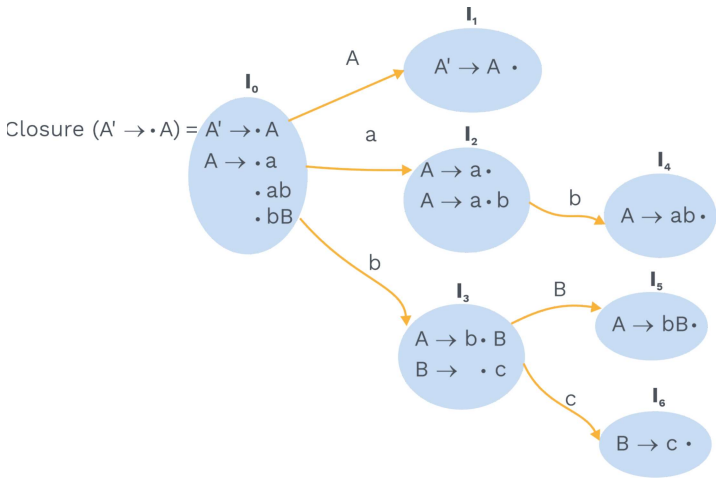


Fig. 3.13 Cononical form of LR Elements

Step 3: Construction of LR(0) parsing table

	Action				Goto	
	a	b	c	$	A	B
I_0	s_2	s_3	—		1	—
I_1	—	—	—	accept	—	—
I_2	r_1	s_4/r_1	r_1	r_1	—	—
I_3	—	—	s_6	—	—	5
I_4	r_2	r_2	r_2	r_2	—	—
I_5	r_3	r_3	r_3	r_3	—	—
I_6	r_4	r_4	r_4	r_4	—	—

numbering for reduce:
$A \rightarrow a$ .....(1)
$A \rightarrow ab$ .....(2)
$A \rightarrow bB$ .....(3)
$B \rightarrow c$ .....(4)

- As in the LR(0) parsing table, there is shift-reduce conflict so the grammar is not LR(0).
- State 2 is inadequate state (SR conflict).

SLR(1) Parsing table:
As there is no SR (shift reduce) or RR (reduce reduce) so the grammar is SLR(1).

	Action				Goto	
	a	b	c	$	A	B
I_0	s_2	s_3	—	—	1	—
I_1	—	—	—	accept	—	—
I_2	—	s_4	—	r_1	—	—
I_3	—	—	s_6	—	—	5
I_4	—	—	—	r_2	—	—
I_5	—	—	—	r_3	—	—
I_6	—	—	—	r_4	—	—

Q.8 **Construct LR(0) and SLR(1) parsing table for the following grammar**
A $\rightarrow$ AA | a | ϵ

Sol: **Step 1:** Construct augmented grammar

$$A' \rightarrow A$$

$$A \rightarrow A\,A$$

$A \rightarrow a$

$A \rightarrow \in$

Step 2: Make LR(0) item and use closure and goto to construct a DFA.

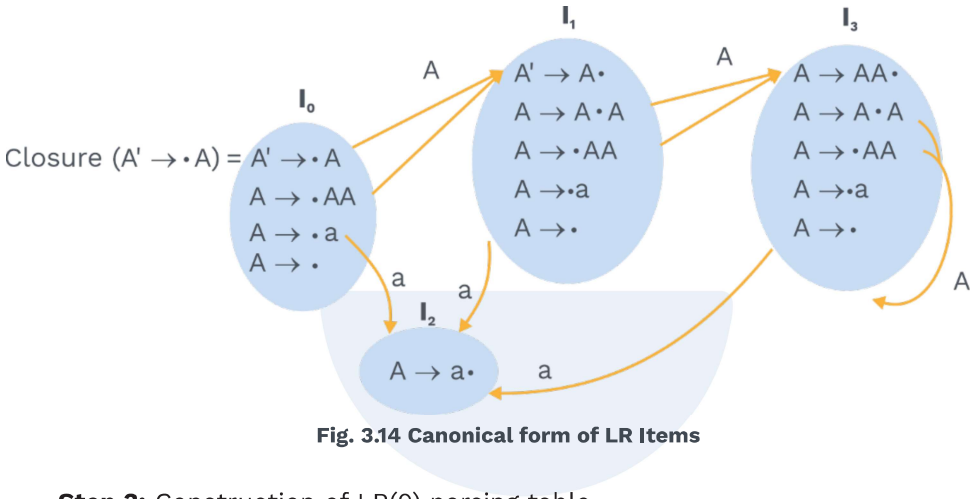


Fig. 3.14 Canonical form of LR Items

Step 3: Construction of LR(0) parsing table

	Action		Goto
	a	$	A
I_0	s_2/r_3	r_3	1
I_1	s_2/r_3	accept/r_3	3
I_2	r_2	r_2	—
I_3	$s_2/r_1/r_3$	r_1/r_3	—

numbering for reduce :
$A \rightarrow AA$.....(1)
$A \rightarrow a$.....(2)
$A \rightarrow \in$.....(3)

- There are both shift reduce and reduce conflict so the grammar is not LR(0).
- The parsing table is same for SLR(1). So, the grammar is not SLR(1) also.

Note

The given grammar is ambiguous grammar and an ambiguous grammar can never be LR(0) and SLR(1).

Q.9 **Construct LR(0) and SLR(1) parsing table for the following grammar**
$A \rightarrow B + A \mid B$
$B \rightarrow b$

Sol: **Step 1:** Construct augmented grammar

$$A' \rightarrow A$$

$$A \rightarrow B + A \mid B$$

$$B \rightarrow b$$

Step 2: Make LR(0) item and use closure and goto to construct a DFA.

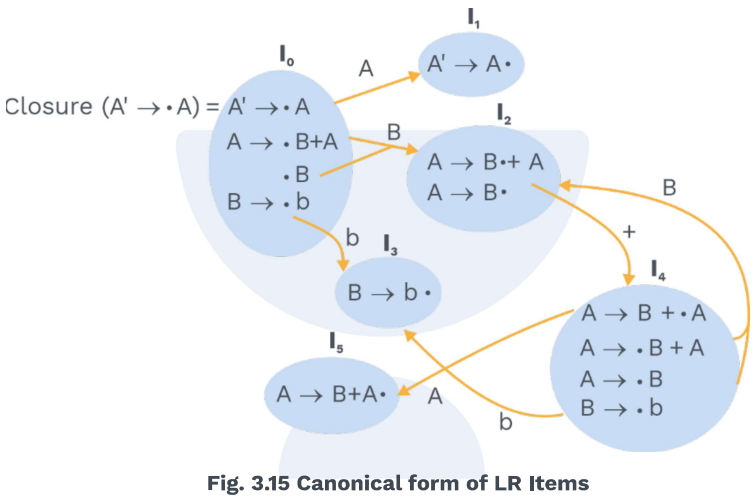


Fig. 3.15 Canonical form of LR Items

Step 3: Construction LR(0) parsing table

	Action			Goto	
	b	+	$	A	B
I_0	s_3	—	—	1	2
I_1	—	—	accept	—	—
I_2	r_2	s_4/r_2	r_2	—	—
I_3	r_3	r_3	r_3	—	—
I_4	s_3	—	—	5	2
I_5	r_1	r_1	r_1	—	—

numbering for reduce :
$A \rightarrow B+A$.....(1)
$A \rightarrow B$.....(2)
$B \rightarrow b$.....(3)

- As there is SR conflict in LR(0) paring table so, the grammar is not LR(0)
- In SLR(1) parsing table, the reduction $A \rightarrow B \cdot$ will go in FOLLOW (A) = $ so no conflict; hence the grammar is SLR(1).

Q.10 **Construct LR(0) and SLR(1) parsing table for the following grammar**
A → BB
B → cB | d

Sol: **Step 1:** Construct augmented grammar:

$$A' \rightarrow A$$

$$A \rightarrow BB$$

$$B \rightarrow cB$$

$$B \rightarrow d$$

Step 2: Make LR(0) item and use closure and goto to construct a DFA.

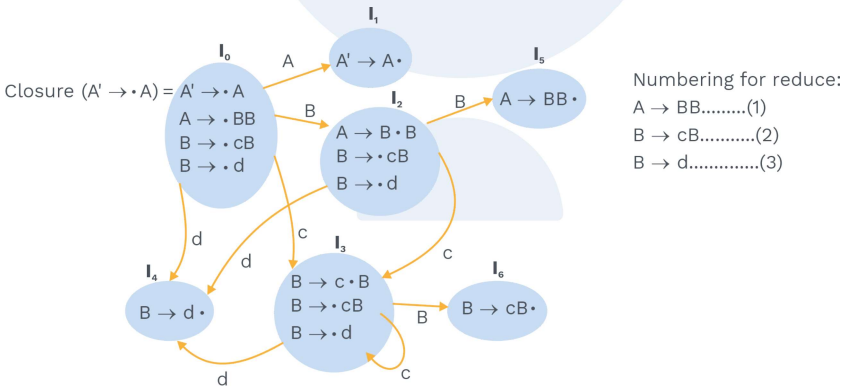


Fig. 3.16 Canonical form of LR Items

Step 3: Construct LR(0) parsing table.

	←Action→			←Goto→	
	c	d	$	A	B
I_0	s_3	s_4	—	1	2
I_1	—	—	accept	—	—
I_2	s_3	s_4	—	—	5
I_3	s_3	s_4	—	—	6
I_4	r_3	r_3	r_3	—	—
I_5	r_1	r_1	r_1	—	—
I_6	r_2	r_2	r_2	—	—

- In the above LR(0) parsing table there is no conflict so it is LR(0), if the grammar is LR(0), it is SLR(1) too.

CLR(1) Parsing table construction:

- Except reduce entries, remaining entries are same as SLR(1).
- If I_i contains $A \rightarrow a, \$$ then reduced entry (r_j) $A \rightarrow a$ in row i under the terminal given by lookhead i.e., ($).
- It takes LR(1) items. The reduce production entries are based on look ahead parsing table.

Rack Your Brain

Construct LR(0) and SLR(1) parsing table for

$S \rightarrow TuTv \mid WvWu$

$T \rightarrow \in$

$W \rightarrow \in$

LALR(1) parser:

LALR(1) parser constructed from CLR(1) parser.

- In CLR(1), if two states are having same production but contain different lookahead symbol, then those two states are combined into a single state in LALR(1).
- Every LALR(1) grammar is CLR(1), but every CLR(1) grammar need not LALR(1).
- In CLR(1) parser even if we don't have RR conflict, after merging, we might have RR conflict in LALR(1)

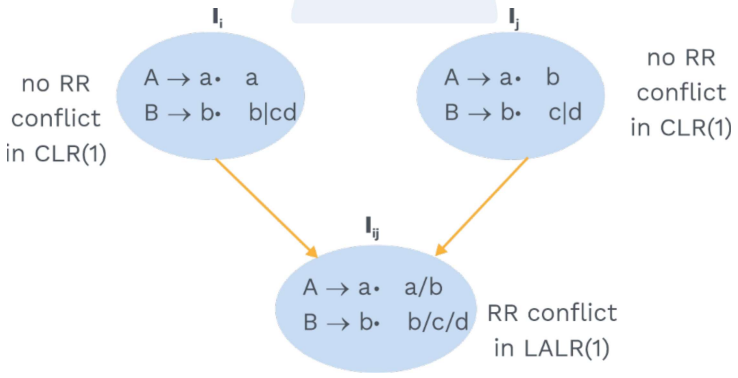


Fig. 3.17 Canonical form of LR Items

- In CLR(1) parser even if we don't have SR conflict, after merging, there will be no SR conflict in LALR(1).

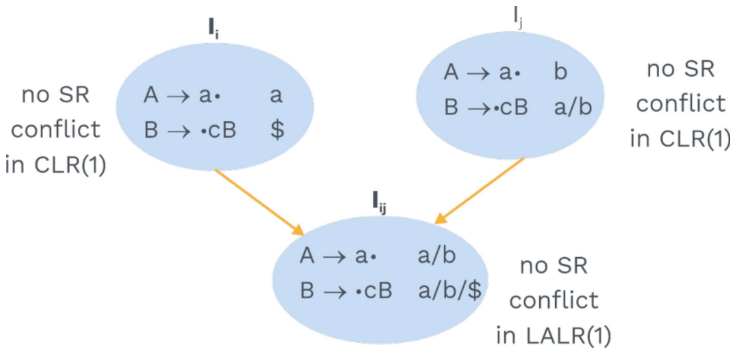


Fig. 3.18 Canonical form of LR(1) Items

- If the grammar is CLR(1) but not LALR(1) then there must be RR conflict.

Conflicts in CLR(1) and LALR(1) :

1) Shift-reduce conflict:

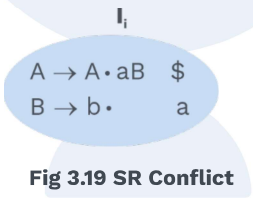


Fig 3.19 SR Conflict

State I_i is having SR conflict because both shifting and reduction on same lookahead symbol i.e., 'a'.

2) Reduce-reduce conflict:

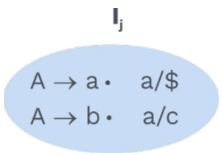


Fig. 3.20 RR Conflict

State I_j is reduce-reduce conflict because both reductions happen on same lookahead i.e., 'a'

SOLVED EXAMPLES

Q.11 **Construct CLR(1) and LALR(1) parsing tables for the following grammar**
A → a A
A → b

Sol: **Step 1:** construct augmented grammar

$A' \rightarrow A$
$A \rightarrow aA$
$A \rightarrow b$

Step2: Make LR(1) item and use closure and goto to construct a DFA.

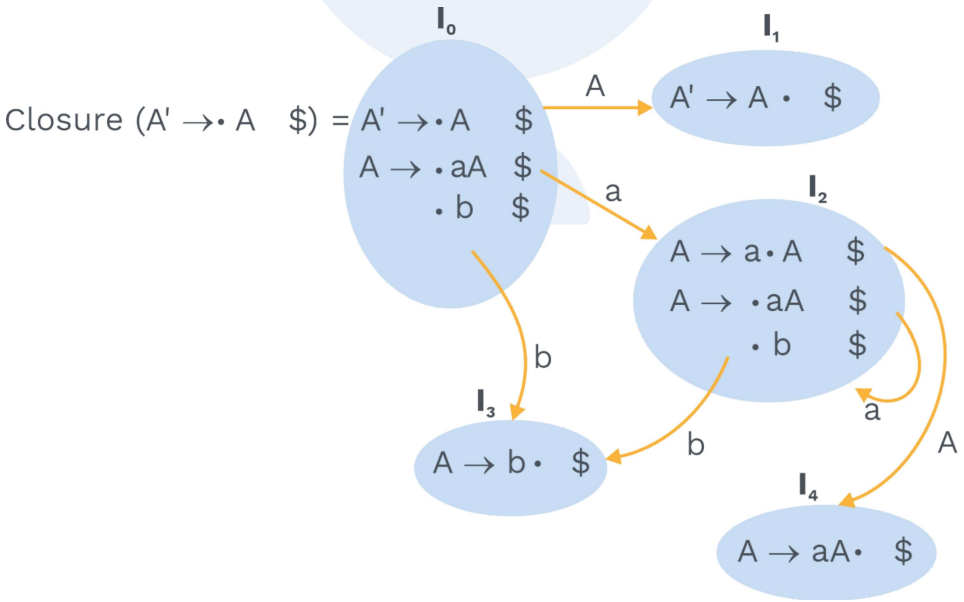


Fig. 3.21

Step 3: Construct CLR(1) parsing table.

	Action			Goto
	c	b	$	A
I_0	s_2	s_3	—	1
I_1	—	—	accept	—
I_2	s_2	s_3	—	4
I_3	—	—	r_2	—
I_4	—	—	r_1	—

numbering to reduce:
$A \rightarrow aA$.....(1)
$A \rightarrow b$.....(2)

- In the above CLR(1) parsing table, we don't have any conflict, so the grammar is CLR(1).
- Since there are no two states having the same production containing different look ahead. So it is already minimized, and it is LALR(1).
- As we see in the LR(0) example, the given grammar is also LR(0), and Every LR(0) grammar is SLR(1), CLR(1) and LALR(1).
- In the above example, we get the same number of states in LR(0), SLR(1), CLR(1) and LALR(1).

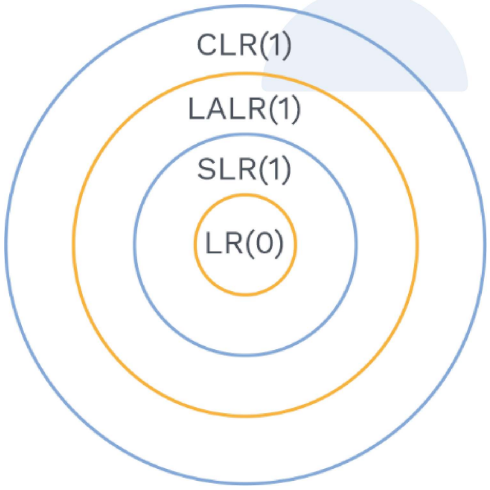


Fig. 3.22 Power of Parser

Q.12 **Construct CLR(1) and LALR(1) parsing table for the following grammar**
A → B + A | B
B → b

Sol: **Step 1:** Construct augmented grammar

$$A' \rightarrow A$$

$$A \rightarrow B + A$$
$$A \rightarrow B$$
$$B \rightarrow b$$

Step 2: Make LR(1) item and use closure and goto to construct a DFA.

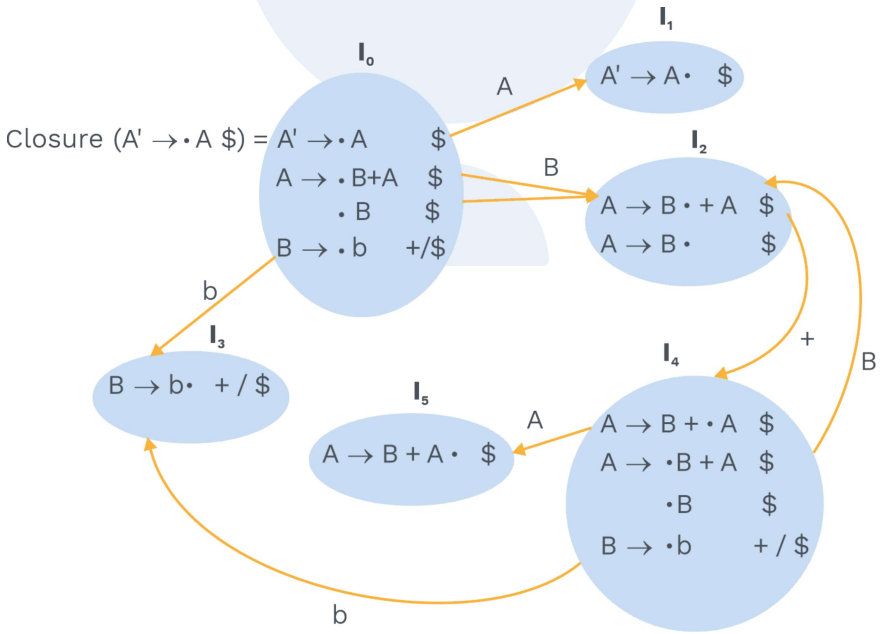


Fig. 3.23 Canonical form of LR(1) Items

Step 3: construction of CLR(1) parsing table

	Action			Goto	
	b	+	$	A	B
I_0	S_3	—	—	1	2
I_1	—	—	accept	—	—
I_2	—	S_4	r_2	—	—
I_3	—	r_3	r_3	—	—
I_4	—	—	—	5	2
I_5	—	—	r_1	—	—

numbering to reduce:
$A \rightarrow B+A$.....(1)
$A \rightarrow B$.....(2)
$B \rightarrow b$.....(3)

In the above CLR(1) parsing table, we don't have any conflict so the grammar is CLR(1) and it is minimized too so LALR(1) also (no conflict).

- The above grammar is not LR(0) as we have solved already, but it is SLR(1). And every SLR(1) grammar is also LALR(1) and CLR(1).

Q.13 **Construct augmented grammar**
$A \rightarrow AA \mid a \mid \in$

Sol: **Step 1:** construct augmented grammar.

$$A' \rightarrow \cdot A$$

$$A \rightarrow \cdot AA$$

$$A \rightarrow \cdot a$$

$$A \rightarrow \cdot \in$$

Step 2: Make LR(1) item and use closure and goto to construct a DFA.

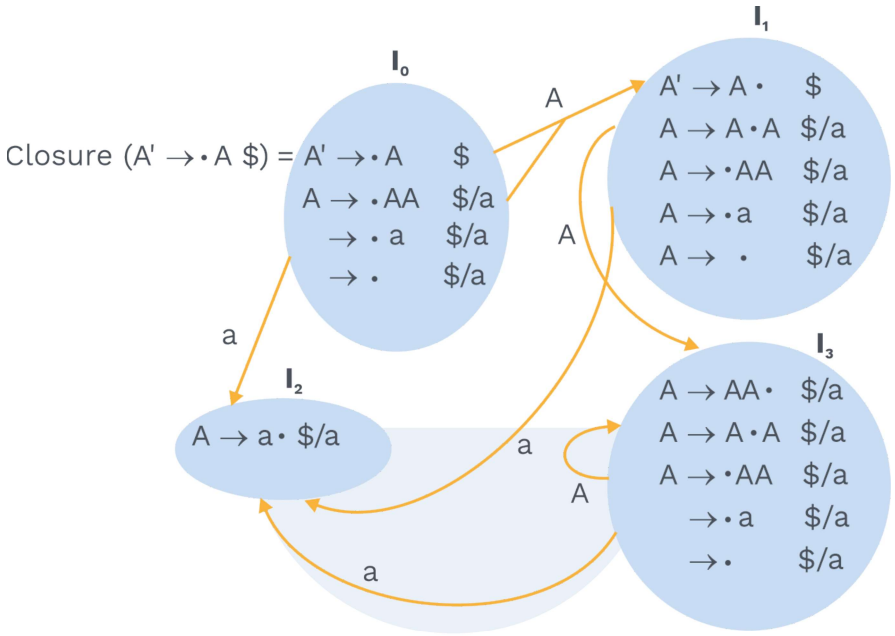


Fig. 3.24 Canonical form of LR(1) Items

Step 3:

	Action		Goto	
	a	$	A	Numbering for reduce:
I_0	s_2/r_3	r_3	1	A → AA.....(1)
I_1	s_2/r_3	accept/r_3	3	A → a.....(2)
I_2	r_2	r_2	–	A → .....(3)
I_3	$s_2/r_3/r_1$	r_3	–	

As in the above CLR(1) parsing table, there is a shift between reduce and reduce-reduce conflict, then it is not CLR(1).

- The number of states is also not further minimized so the parsing table remains the same in LALR(1).
- If the grammar is not CLR(1) then it is also not LALR(1).
- The given grammar is ambiguous, and an ambiguous grammar can never be LL(1), LR(0), SLR(1), CLR(1) and LALR(1).
- If grammar is not CLR(1) then it is not LALR(1) not SLR(1) not LR(0).

Q.14 **Construct CLR(1) and LALR(1) parsing table for the following grammar.**
A → BB
B → cB | d

Sol: **Step 1:** Construct augmented grammar

$$A' \rightarrow .A$$

$$A \rightarrow .BB$$

$$B \rightarrow .cB \mid .d$$

Step 2: Make LR(1) item and use closure and goto to construct a DFA.

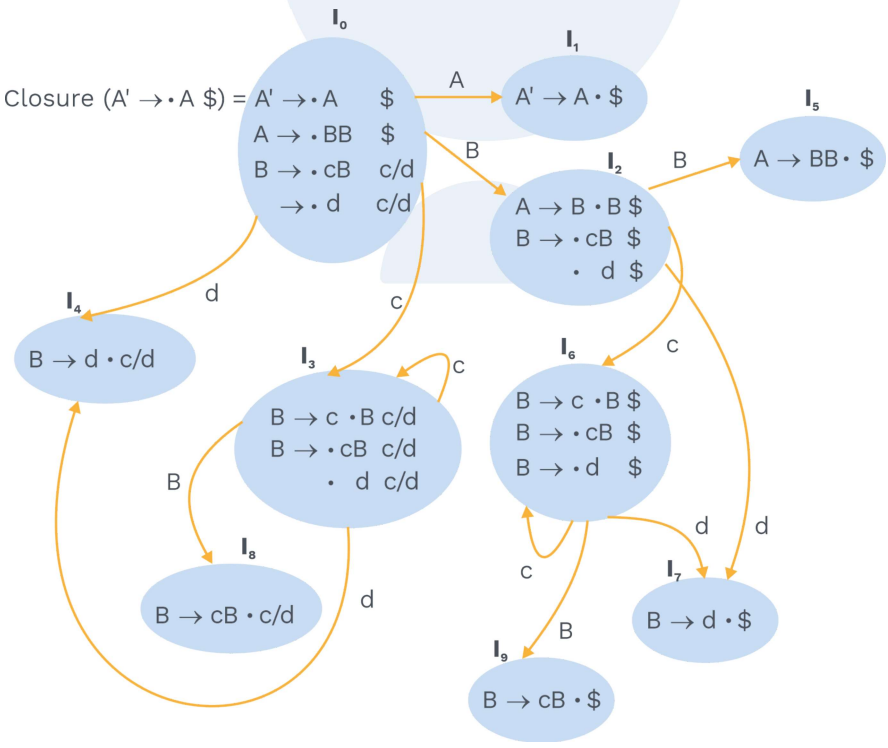


Fig. 3.25 Canonical form of LR(1) Items

	Action			Goto	
	c	d	$	A	B
I_0	s_3	s_4	accept	1	2
I_1	—	—	—	—	—
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	—	—

Numbering for reduce:
$A \rightarrow BB$.....(1)
$B \rightarrow cB$.....(2)
$B \rightarrow d$.....(3)

CLR(1) parsing table:

- In above CLR(1) parser there is no conflict, so it is CLR(1).
- Since state I_3, I_6. State I_4, I_7 and state I_8, I_9 having same production and different look ahead so we can combine in LALR(1).

$$I_3, I_6 \rightarrow I_{36}$$

$$I_4, I_7 \rightarrow I_{47}$$

$$I_8, I_9 \rightarrow I_{89}$$

	Action			Goto	
	c	d	$	A	B
I_0	s_{36}	s_{47}	—	1	2
I_1	—	—	accept	—	—
I_2	s_{36}	s_{47}	—	—	5
I_{36}	s_{36}	s_{47}	—	—	89
I_{47}	r_3	—	—	—	—
I_5	—	—	—	—	—
I_{89}	r_2	r_2	r_2	—	—

- There is no conflict in LALR(1) parsing table, so it is also LALR(1).
- Number of states in LALR(1) parser is same as LR(0) and SLR(1) parser but less than CLR(1) parser.
  Updated Canonical collection of LALR(1).

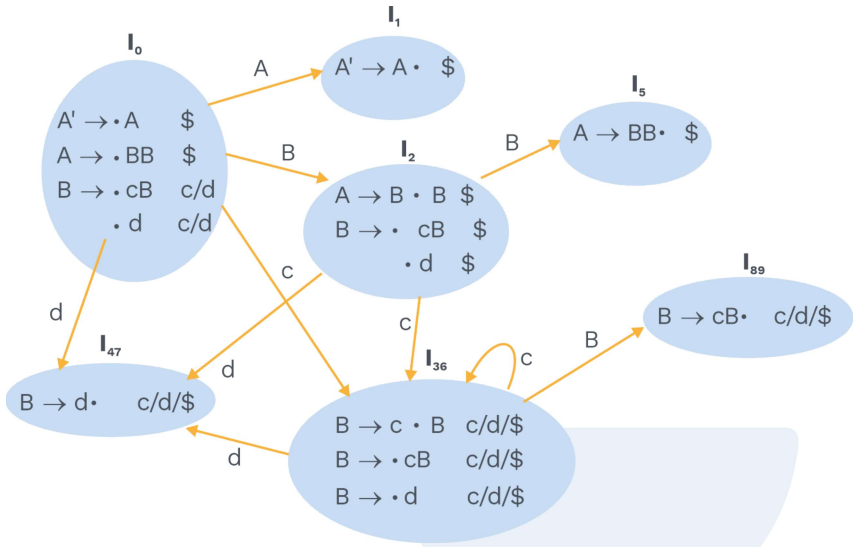


Fig. 3.26 Canonical form of LR(1) Items

Previous Years' Question

Consider the grammar

$$S \rightarrow (S) \mid a$$

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:

a) $n_1 < n_2 < n_3$ **b)** $n_1 = n_3 < n_2$

c) $n_1 = n_2 = n_3$ **d)** $n_1 \geq n_3 \geq n_2$

Sol: b) **[GATE: CS 2005]**

Operator precedence parsing:

- Operator precedence parser is a form of shift reduce parser, it is easy to implement.
- Operator grammar has the following properties:
  - **1)** A grammar does not contain $\in$ production.
  - **2)** A grammar does not contain two adjacent non-terminals.

  For e.g - $S \rightarrow AB$ is not allowed.

Example:
Consider the following expression

$$E \rightarrow EAE \mid id$$

$$A \rightarrow + \mid - \mid * \mid /$$

As the given grammar is not operator grammar, because the right side EAE has three consecutive non-terminals.

- However, if we substitute A with each of its productions we obtain the following operator grammar.

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid id$$

In operator precedence parsing, we use three disjoint precedence relations $\lessdot, \doteq$ and $\gtrdot$ between each pair of terminals.

- These precedence relations guide the selection of handles and have the following meaning.

Relation	Meaning
$a \lessdot b$	a "yields precedence to" b
$a \doteq b$	a "has the same precedence as" b
$a \gtrdot b$	a "takes precedence over" b

- There are common ways of determining what precedence relation should hold between a pair of terminals. For e.g. if * has higher precedence than +, we make $+ \lessdot *$ and $* \gtrdot +$

Example:
Consider the following operator grammar.

$E \rightarrow E + E \mid E * E \mid id$, draw the operator precedence table according to the following rules:

- * has higher precedence than + and * is left associative.
- + has lower precedence than * and + is left associative.
- id has highest precedence and $ has the lowest precedence.

Previous Years' Question

Consider the grammar shown below:

$$S \rightarrow CC$$

$$C \rightarrow cC \mid d$$

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)
Sol: a) **[GATE: CS 2003]**

	id	+	*	$
id	—	·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	—

Let a be topmost terminal symbol on stack, and b is the look ahead terminal symbol then.

1) If $a \lessdot b$ or $a \doteq b$, then push b / shift b onto the stack and increment input symbol.
2) If $a \cdot\!> b$, then pop stack until the top of stack terminal is related by $\lessdot$ to the terminal most recently popped.
3) Else error.

Operator precedence table:

Operator precedence parsing algorithm:

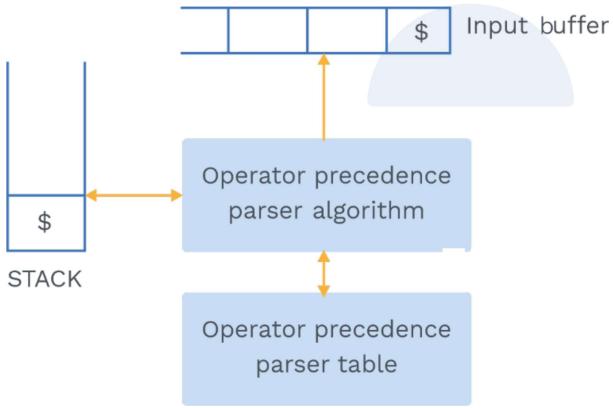


Fig. 3.27 Parser Diagram

Q. Consider the grammar $E \rightarrow E+E \mid E*E \mid id$ parser the input id+id*id using operator precedence parser.

Stack	Input	Action
$	id+id*id$	Shift
$id	+id*id$	Reduce E → id
$E	+id*id$	Shift
$E+	id*id$	Shift
$E+id	*id$	Reduce E → id
$E+E	*id$	Shift
$E+E*	id$	Shift
$E+E*id	$	Reduce E → id
$E+E*E	$	Reduce E → E*E
$E+E	$	Reduce E → E+E
$E	$	Accept

Table 3.4

Classification of bottom-up parser:

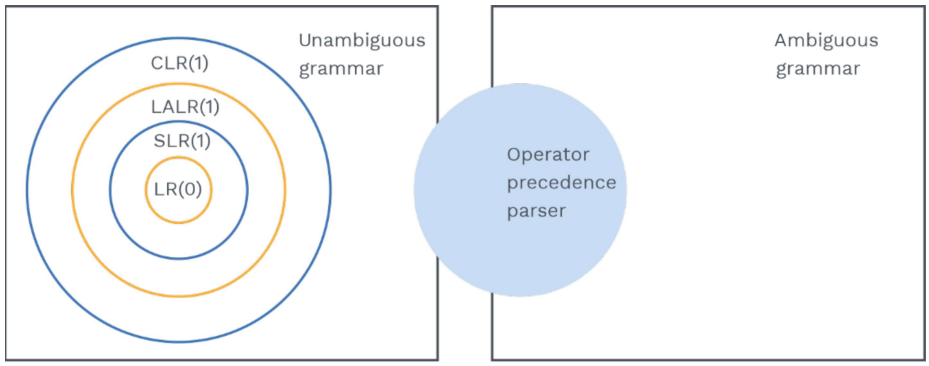


Fig. 3.28 Power of each Parser in the form of Sets

Previous Years' Question

Which of the following grammar rules violate the requirements of an operator grammar? P, Q, R are non-terminals and r, s, t are terminals.

i) $P \rightarrow QR$ **ii)** $P \rightarrow QsR$

iii) $P \rightarrow \in$ **iv)** $P \rightarrow QtRr$

a) (i) only **b)** (i) and (iii) only

c) (ii) and (iii) only **d)** (iii) and (iv) only

Sol: b) **[GATE: CS 2005]**

Previous Years' Question

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

$X \rightarrow c.X,\ c/d$ $X \rightarrow c.X,\ \$$

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

$X \rightarrow .d,\ c/d$ $X \rightarrow .d,\ \$$

Which of the following statement related to margining of the two sts in the corresponding parser is/are FALSE?

1) Cannot be merged since look aheads 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

Sol: d) **[GATE: CS 2013]**

Chapter Summary

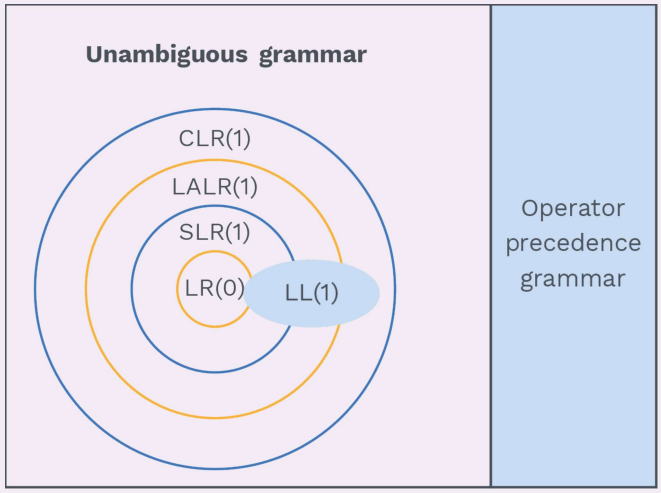


Fig. 3.29 Power of each Parser in a Bigger Picture

- Every LL(1) grammar is CLR(1).
- Every LR(0) grammar is CLR(1).
- If the grammar is not CLR(1) then it is not LALR(1), not SLR(1) and not LR(0).
- If the grammar is ambiguous it will not be LL(1), LR(0), SLR(1), LALR(1) and CLR(1) grammar.
- Number of shift entries are same in LR(0), SLR(1) and LALR(1) but less than or equal to CLR(1).
- Number of reduce entries in LR(0) $\geq$ SLR(1) $\geq$ LALR(1) $\geq$ CLR(1)
- Number of state entries (goto) are also same for LR(0), SLR(1) and LALR(1) but less than or equal to CLR(1).
- Parsing Table size is same for LR(0), SLR(1) and LALR(1) but less than or equal to CLR(1).
- Expressive power of parsers
  LL(1) < LR(0) < SLR(1) <LALR(1) < CLR(1)
- Number of states in LR parsers

$$n(\text{LR}(0)) = n(\text{SLR}(1)) = n(\text{LALR}(1)) \leq n(\text{CLR}(1))$$

- Size of LR parsing table = Number of states × (Number of symbols + 1)

4 Syntax Directed Translation

4.1 INTRODUCTION

A syntax-directed definition (SDD) is a context-free grammar with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions.

For example:

Ex. 1 $\underbrace{E \rightarrow E_1 + T}_{\text{Production}}$ $\underbrace{\{E.code = E_1.code \,||\, T.code\}}_{\text{Semantic rule}}$

Ex. 2 $\underbrace{E \rightarrow E_1 + T}_{\text{Production}}$ $\underbrace{\{print\ '+'\}}_{\text{Semantic action}}$

Ex. 3 $E \rightarrow E_1 \underbrace{\{print\ '+'\}}_{\text{Semantic action}} + T$

Note

- Position of the semantic action in the production body determines the order in which the action is executed.
- Value of each attribute at the parse tree node is defined by the semantic action for that node.

4.2 ATTRIBUTE

In grammar, every variable has its own attributes. We shall deal with two kinds of attributes for non-terminals:

a) Synthesised Attributes

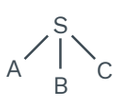
b) Inherited Attributes

Synthesised attributes:

They are those attributes where the value of the attribute depends upon the attribute value of their children.

For example:

$S \rightarrow ABC$ {S.val = A.val + B.val + C.val}

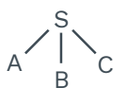


Here we can clearly see that value of attribute 'S' (S.val) depends upon its children attribute values.

Thus, we call 'S' as a synthesised attribute.

Inherited attributes:
They are those attributes where the value of attribute depends on the parent attribute value or siblings attribute values.

For example:
$S \rightarrow ABC$ {B.val = S.val + A.val + B.val}



Here we can clearly see that value of the attribute B depends upon its siblings attribute value, i.e., (A, C) and upon its parent attribute value which is (S).

Comparison between synthesised attributes and inherited attributes:

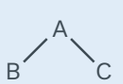
Synthesised Attributes		Inherited Attributes	
1)	Attribute value depends upon its children attribute values.	**1)**	Attribute value depends upon its sibling and parent attributes values.
2)	In the production left-hand, there must be a non-terminal.	**2)**	On the production right-hand side, there must be a non-terminal.
3)	Evaluation is done on the basis of bottom-up traversal of the parse tree.	**3)**	Evaluation is done on the basis of top-down or left to right traversal of the parse tree.
4)	These are used in both S-attributed as well as L-attributed SDT.	**4)**	These are used only in L-attributed SDT.
5)	Example: $F \rightarrow (E)$ {F.val = E.val} F ↓ E Thus parent value depends on children. ∴ F is a synthesised attribute.	**5)**	Example: $A \rightarrow BC$ {B.val = A.val * C.val}  The value of B depends on sibling C and parent A ∴ B is an inherited attribute.

Table 4.1

4.3 CONSTRUCTION OF SYNTAX TREE

Syntax tree:

Definition

A tree in which internal nodes consist of operators and leaf nodes consist of operands is known as a syntax tree.

- In order to construct a syntax tree mainly three functions are used, which are as follows:
  1) Make-node (op, left, right)
  2) Make-leaf (id, entry to symbol table)
  3) Make-leaf (num, value)
- Make-node function is used to create a node with an operator where 'op' represents the operator, left represent the left child address, and right represents the right child address.
- Make-leaf function is used to create a node with an identifier. In this function, 'id' represents identifier, and entry to the symbol table is a pointer that contains the address of the symbol table where this identifier will be stored.
- Make-leaf function is also used to create a node with constant values. In this function, value represent the constant.

SOLVED EXAMPLES

Q.1 **Construct a syntax tree for expression 7*x+4-z.**

Sol: **Functions needed for construction of syntax tree**

Symbol	Function
7	P_1 = make-leaf (num, 7)
x	P_2 = make-leaf (x, pointer to symbol table)
4	P_3 = make-leaf (num, 4)
z	P_4 = make-leaf (z, pointer to symbol table)
*	P_5 = make-node (*, P_1, P_2)
+	P_6 = make-node (+, P_5, P_3)
–	P_7 = make-node (–, P_6, P_4)

Table 4.2

By calling the above functions, syntax tree will be created as follows:

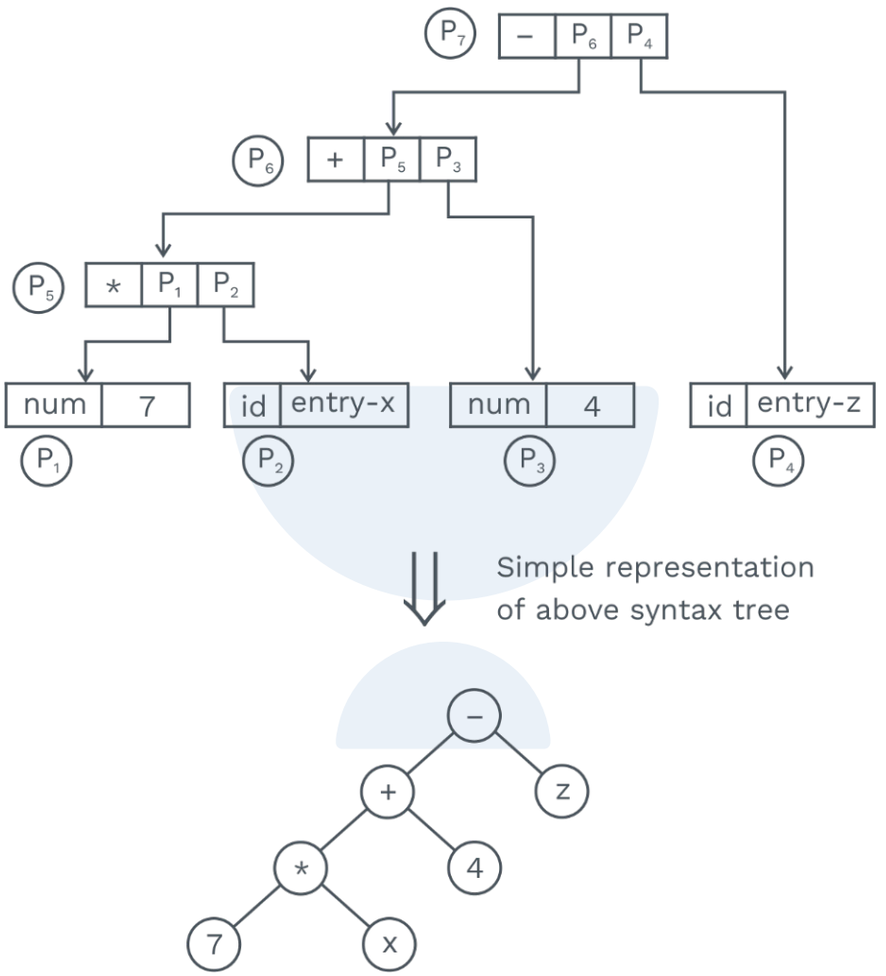


Fig. 4.1

Note

- First of all, all the functions for the construction of leaf node will be called.
- Then, for the construction of internal nodes, the functions with the highest precedence operators are called.

SOLVED EXAMPLES

Q.2 **Construct a syntax tree for the expression x = 4 + y * z.**

Sol: **Functions to be called for the syntax tree construction**

Symbol	Function
x	P_1 = make-leaf (x, pointer to symbol table)
4	P_2 = make-leaf (num, 4)
y	P_3 = make-leaf (y, pointer to symbol table)
z	P_4 = make-leaf (z, pointer to symbol table)
*	P_5 = make-node (*, P_3, P_4)
+	P_6 = make-node (+, P_2, P_5)
=	P_7 = make-node (=, P_1, P_6)

Table 4.3 Functions

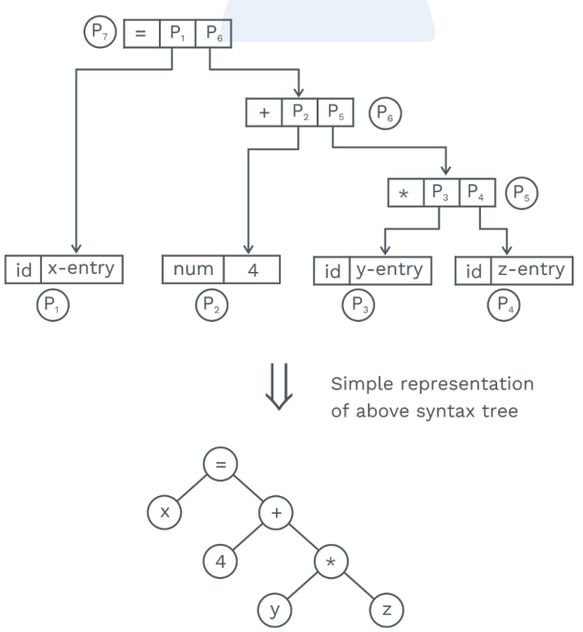


Fig. 4.2 Tree

Q.3 **Construct a syntax tree for expression a – 4 + 6**

Sol: **Functions to be called for the syntax tree construction**

Symbol	Function
a	P_1 = make-leaf (id, entry-a)
4	P_2 = make-leaf (num, 4)
6	P_3 = make-leaf (num, 6)
–	P_4 = make-node (–, P_1, P_2)
+	P_5 = make-node (+, P_3, P_4)

Table 4.4 Functions

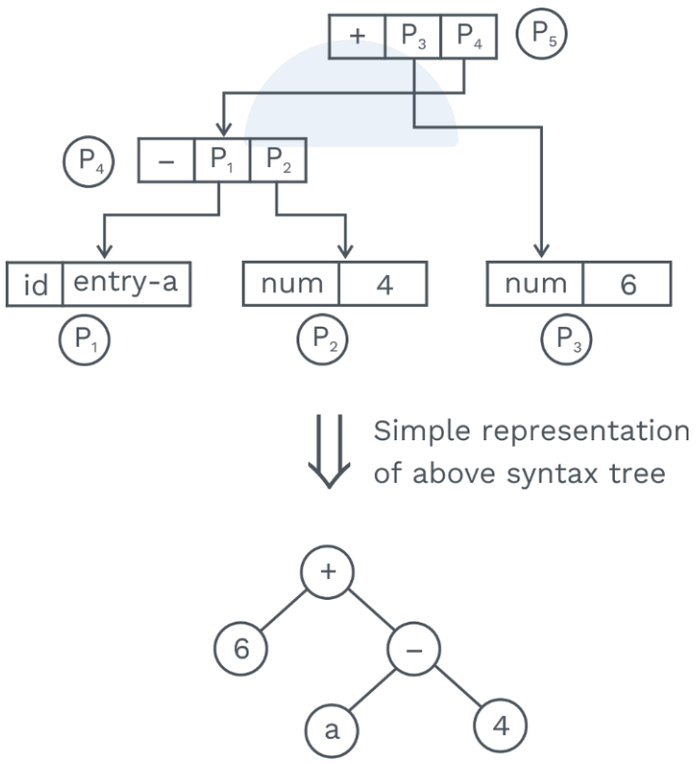


Fig. 4.3 Tree

Syntax directed definition for construction of a syntax tree:
Given below productions and semantic actions for the syntax tree construction of an expression consisting of '*' and '÷' operator. For calling functions make-leaf and make-node productions of grammar are used.

Production	Sematic Function
1) $E \rightarrow E_1 * T$	E.node = make-node (*, E_1.node, T.node)
2) $E \rightarrow E_1 \div T$	E.node = make-node (÷, E_1.node, T.node)
3) $E \rightarrow T$	E.node = T.node
4) $T \rightarrow (E)$	T.node = E.node
5) $T \rightarrow id$	T.node = make-leaf (id, id-entry)
6) $T \rightarrow num$	T.node = make-leaf (num, value)

Table 4.5 Semantic Functions

- '*' and '÷' are usually at the same precedence level and are jointly left associative.
- All non-terminals are synthesised attribute nodes which represent the node of syntax tree.
- When first production $E \rightarrow E_1 * T$ is used, its action creates a node with '*' for operator and two children, E_1.node and T.node .
- Second production has a similar semantic action.
- For production $E \rightarrow T$ no node is created. Since E.node is the same as T.node.
- Similarly, no node is created for production $E \rightarrow (T)$ value of T.node is same as E.node. Parentheses are for grouping, they influence the structure of the parse tree and syntax tree, but once their job is done there is no need to retain them in the syntax tree.
- Last two T-productions have a single terminal on the right; we use constructor make-leaf to make a suitable node which becomes the value of T.node.
- The syntax tree for input "x ÷ 4 * y" to the above SDD

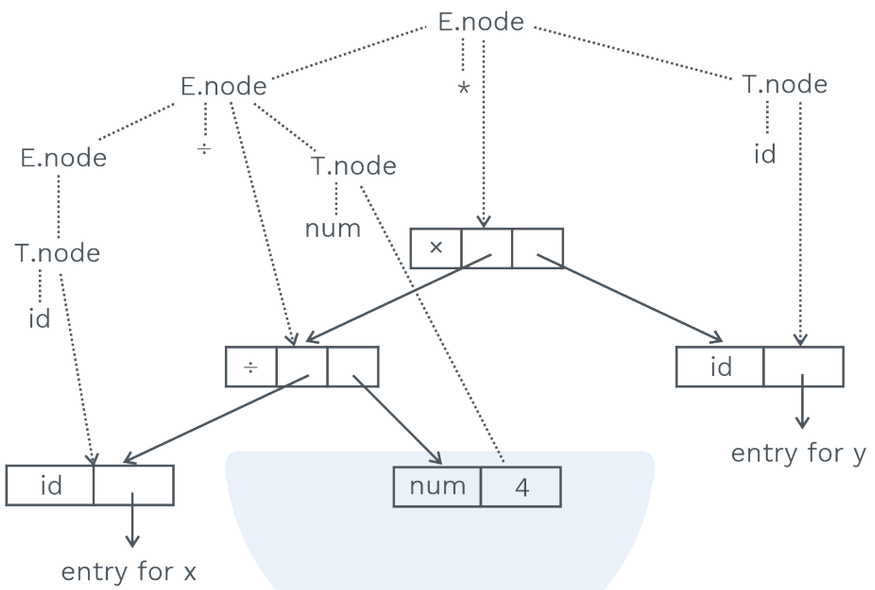


Fig. 4.4 Tree

Syntax tree for x ÷ 4 * y is shown above by solid lines.

Underlying parse tree which need not be constructed as shown with dotted edges.

- Steps in the construction of syntax tree for "x ÷ 4 * y"

1) S_1 = make-leaf (id, entry x)
2) S_2 = make-leaf (num, 4)
3) S_3 = make-node (÷, S_1, S_2)
4) S_4 = make-leaf (id, entry y)
5) S_5 = make-node (*, S_3, S_4)

SOLVED EXAMPLES

Q.4 **Construct syntax tree to execute the given arithmetic expression**
Input: 10 – 5 ÷5
Output: 9

Note

- Steps for creating syntax tree:
- Write unambiguous grammar along with semantic action
- Build tree

Sol:

$X \rightarrow X_1 - T$	**{X.val = X_1.val – T.val}**
$X \rightarrow T$	{X.val = T.val }
$T \rightarrow T_1 \div F$	{T.val = T_1.val ÷ F.val}
$T \rightarrow F$	{T.val = F.val }
$F \rightarrow id$	{F.val = id}

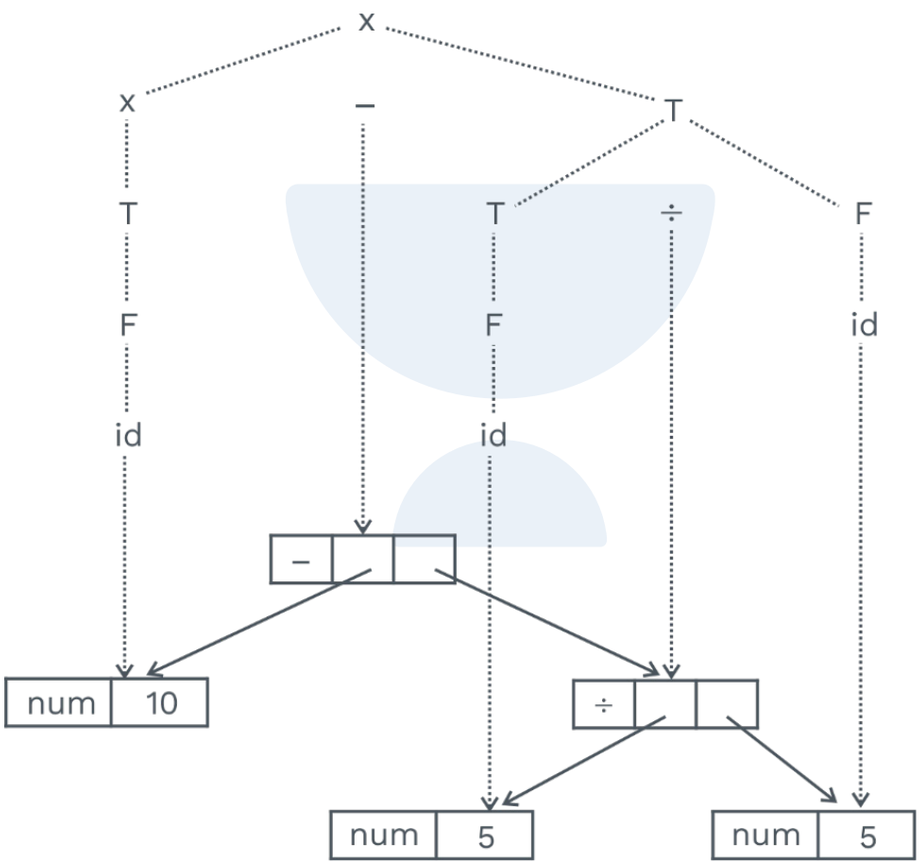


Fig. 4.5 Tree

Note

- In this example we have designed our grammar in such a way that the output should be 9.
- For getting output as 9 we have given higher precedence to '÷' operator and then '–' operator.

Q.5 **Given a SDT:**

X → YX {Print 1}

X → Y {Print 3}

X → ZX {Print 4}

X → Z {Print 9}

Y → aZ {Print 2}

Y → a {Print 5}

Z → bZ {Print 7}

Z → b{Print 8}

If the input is "abba", then what will be the output?

Sol: **Parse tree formed for the given input "abba"**

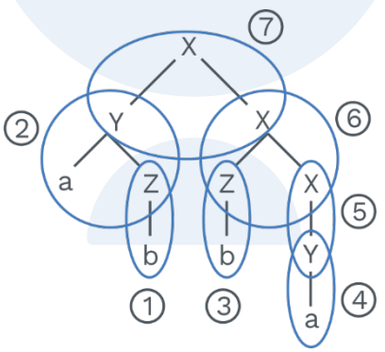


Fig. 4.6 Tree

Evaluation of parse tree will be in post order.
Thus output:

1	2	3	4	5	6	7
↓	↓	↓	↓	↓	↓	↓
8	2	8	5	3	4	1

Therefore, when the input is 'abba', output will be 8285341.

Q.6 **Consider the following SDT:**

A → A$B {A.val = A_1.val + B.val}

A → B {A.val = B.val}

B → B#F {B.val = _____}

B → F {B.val = F.val}

F → num {F.val = num}

If the input is: 7#2$3#5$4 and for the given input, the output is 33. Then what will be the missing semantic action?

a) B.val = B.val + F.val **b) B.val = B.val * F.val**

c) B.val = B.val – F.val **d) None of these**

Sol:

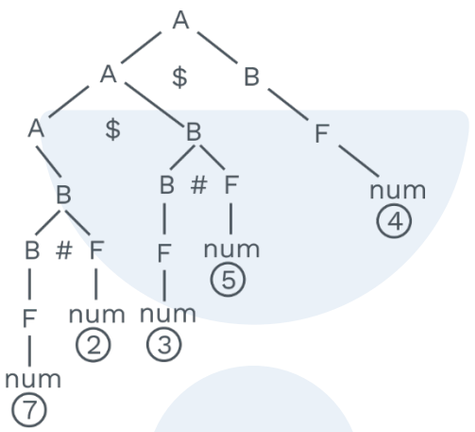


Fig. 4.7 Tree

From the parse tree, it is clear that # has higher precedence than $ because for evaluation, we use post order traversal and thus, first of all, all the '#' operations will be performed.

From semantic actions it is clear that '$' is a plus operation.

Now looking at options and evaluating the output.

Option (a): B.val = B.val + F.val

According to this # = +

$$\therefore 7 + 2 + 3 + 5 + 4 = 21 \neq 33$$

Wrong option

Option (b): B.val = B.val * F.val

$$\therefore \# = *$$

Thus, 7 * 2 + 3 * 5 + 4

As we know for parse tree # has higher precedence

$\therefore$ = 14 + 15 + 4 = 33 = output

$\therefore$ Missing semantic action is B.val = B.val * F.val

Detection of associativity of operator in SDT:

For directly checking the associativity of an operator from the syntax-directed translation, we have to see whether a new variable is coming right side of the operator or left side of the operator in the production.

If a new variable is coming right side of the operator, then its associativity is left to right.

If a new variable is coming left side of the operator, then its associativity is from right to left.

For e.g.

$$A \rightarrow A * B \mid B$$

$$B \rightarrow C + B \mid C$$

$$C \rightarrow id$$

In the given grammar, in production $A \rightarrow A * B$ new variable is coming on right side of the operator thus, '*' is left-associative.

In the production $B \rightarrow C + B$ new variable 'C' is coming on the left side of the production thus associativity of '+' is right to left.

SOLVED EXAMPLES

Q.7 **Given SDT:**

$X \rightarrow X * Z$ {X.val = X_1.val * Z.val}

$X \rightarrow Z$ {X.val = Z.val}

What is the associativity of '*' operator ?

Sol: **In production $X \rightarrow X * Z$ new variable is coming on right-hand side of the operator.**

$\therefore$ Associativity of * is left to right (left associative).

Previous Years' Question

In the following grammar:

$X \rightarrow X \oplus Y \mid Z$

$Y \rightarrow Z \odot Y \mid Z$

$Z \rightarrow id$

Which of the following is true?

a) $\oplus$ is left-associative while $\odot$ is right-associative.

b) Both $\oplus$ and $\odot$ is left-associative.

c) $\oplus$ is right-associative while $\odot$ is left-associative.

d) None of these

Sol: a) **[GATE-CS-1997]**

Checking associativity and precedence of the operator in parse tree:

- During the evaluation of a parse tree, the operator which is evaluated first have higher precedence.
- During the evaluation of parse tree, for a particular operator, if the right-most evaluation is done first, then it is right-associative; if the left-most operation is done first; then it is left-associative.

SOLVED EXAMPLES

Q.8 **Given a syntax tree:**

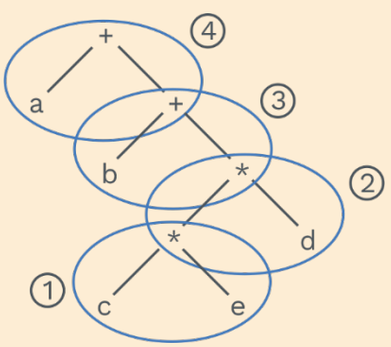


Fig. 4.8 Tree

What will be the associativity and the precedence order of the operators?

Sol:

- Expression derived from syntax tree: "a + b + c * e * d"
- By applying post order evaluation :

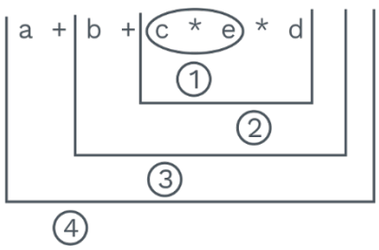


Fig. 4.9 Associativity

c * e is evaluated first, and after that (c * e) * d from this it is clear that '*' operator is having higher precedence then that '+' operator and associativity of * is from left to right because 'c * e' is evaluated first.

The Precedence of '+' operator is less than '*'

Associativity of '+' operator is from right to left because the right side '+' is evaluated first.

Q.9 **Given a syntax tree:**

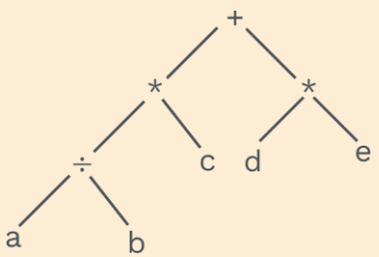


Fig. 4.10 Tree

What will be the associativity and precedence of +, *, ÷ operators?

Sol: **An Expression derived from syntax tree:**

a ÷ b * c + d * e

Now applying post order evaluation on the syntax tree.

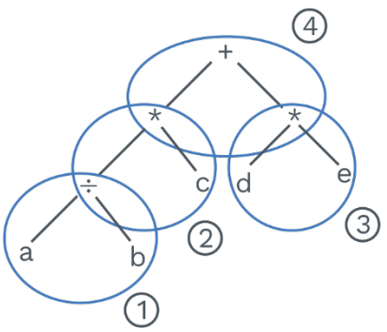


Fig. 4.11 Tree

So, order of evaluation,

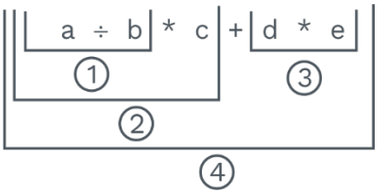


Fig. 4.12 Associativity

Therefore '÷' operator has the highest precedence, and '+' operator has the lowest precedence.

÷ > * > + {precedence order}

- Associativity of * operator is left to right associativity. The associativity of '+' and '÷' operators cannot be told because they are used only once in the expression.

Previous Years' Question

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

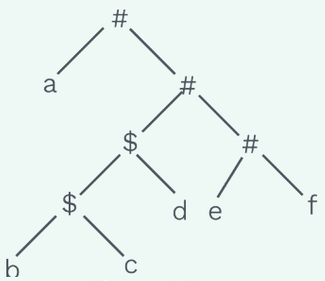


Fig. 4.13 Tree

Which one of the following is correct for the given parse tree?

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.

Sol: a) **[GATE: CS-2018]**

4.4 TYPES OF SDT

Depending upon the type of attribute syntax directed translations are of two types:

1) S-attributed SDT

2) L-attributed SDT

Rack Your Brain

Terminals of any grammar can have synthesised attributes but not inherited attributes. Think why ?

S-attributed SDT:

- In these types of SDT, only synthesised attributes are present.
- For evaluation of S-attributed SDT, bottom-up execution, also known as post order evaluation is used.
- In these types of SDT, semantic actions will always be present on the right-hand side of the production at the right-most places.

Examples:

Ex 1: A $\rightarrow$ BCD {A.val = B.val + C.val}

Given SDT is S-attributed because A is a synthesised attribute and semantic action is present on the right-hand side, right-most place.

Ex 2: A $\rightarrow$ B {A.val = B.val}CD

Given SDT is not S-attributed because semantic action is present on the right-hand side in between the production.

Ex 3: A $\rightarrow$ BCD {C.val = A.val + D.val}

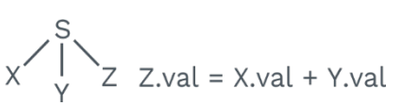
Given SDT is not S-attributed because attribute 'C' is an inherited attribute because the value of 'C' depends on the parent and sibling.

L-attributed SDT:

- In these types of SDT, both synthesised and inherited attributes are present.
- If inherited attributes are present, then that attribute can depend only upon the parent or left sibling. It should not depend on the right sibling.
- For evaluation of L-attributed SDT left to right evaluation is done.
- In this type of SDT, semantic actions can be present anywhere on the right-hand side of the production.

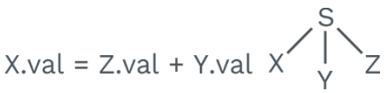
Examples:

Ex 1: S $\rightarrow$ XYZ {Z.val = X.val + Y.val}



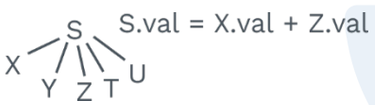
Given SDT is L-attributed because Z is an inherited attribute which depends on left sibling values X and Y.

Ex 2: S → XYZ {X.val = Y.val + Z.val}



Given SDT is neither L-attributed nor S-attributed the because attribute X is an inherited attribute which depends on the right sibling.

Ex 3: S → XYZ {S.val = X.val + Z.val}TU



Given SDT is L-attributed because attribute S is synthesised attribute and semantic action is present on the right-hand side in middle.

Rack Your Brain

Do you think all the S-attributed STD will also of type L-attributed. If yes then what is the reason ?

Comparison between L-attributes and S-attributes:

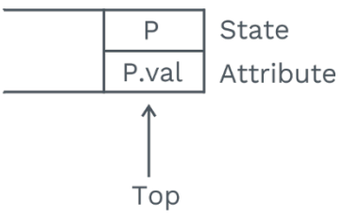
L-attributes		S-attributes	
1)	It consists of both synthesised and inherited attribute, but inherited attribute should not depend on the right sibling.	**1)**	It only consists of synthesised attribute that does not contain inherited attribute.
2)	For evaluation depth-first, the left to right evaluation approach is used.	**2)**	For evaluation bottom up approach is known as post order evaluation is used.
3)	Semantic actions are present anywhere on right-hand side of the production.	**3)**	Semantic action can be present at rightmost place on the right-hand side of the production.
4)	Every L-attributed can not be S-attributed.	**4)**	Every S-attributed is L-attributed.

Table 4.6 Comparison

4.5 BOTTOM-UP EVALUATION OF INHERITED ATTRIBUTES

- Bottom-up evaluation of inherited attributes can be done with the help of stack.
- Attribute of each grammar symbol can be put on the stack where they can be found during the reduction.
- Parser stack contains a record with a field of grammar symbol (or present state) and below it is a field for the attribute.
- For eg: There is a production P → QRS

- Grammar symbols SRQ are on the top of stack, so they are about to be reduced by the production P → QRS.
- After reduction top of the stack will be pointing at position 'Q' and in place of Q, there will be P, and in place of Q.val, there will be P.val.



- In inherited attribute, attribute value depend on the parent as well as on the sibling. We can apply bottom-up evaluation for that inherited attributes where siblings or parents are already present on the stack.
- For eg: P → QR {R.val = Q.val}

Suppose Q is an inherited attribute; its value is present on the stack, so R will use its value, and we can apply the bottom-up approach easily.

- In case if the production P → QR {R.val = Q.val + P.val}

Q.val is present in the stack, but P.val is not present in the stack. Then, in this case bottom-up evaluation can not be applied; we are forced to use depth-first evaluation.

Previous Years' Question

In a bottom-up evaluation of a syntax directed definition of inherited attributes can

a) Always be evaluated
b) Be evaluated only if the definition is L-attributed
c) Be evaluated only if the definition has synthesised attribute
d) Never be evaluated

Sol: c) **[GATE; CS-2003]**

SOLVED EXAMPLES

Q.10 **Given a SDT**

X → YZ {Z.in = Y.type}

Y → int {Y.type= int }

Y → Float {Y.type = float}

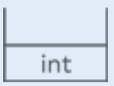
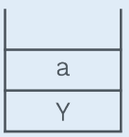
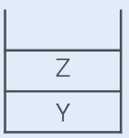
Y → real {Y.type = real}

Z → Z_1, id {Z_1.in = Z.in} {add type (id.entry, Z.in)}

Z → id {add type (id.entry, Z.in}

Input to SDT is "int a,b,c". Then show the implementation of the bottom-up evaluation.

Sol:

Input	Stack	Production
Int a, b, c	–	–
a, b, c		–
a, b, c		Y → int
, b, c		–
, b, c		Z → id

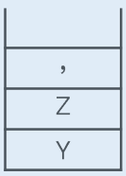
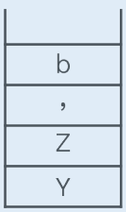
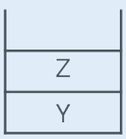
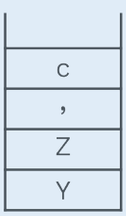
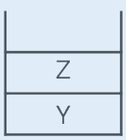
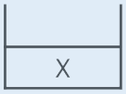
Input	Stack	Production
b, c		–
, c		–
, c		$Z \rightarrow Z_1$, id
c		–
–		–
–		$Z \rightarrow Z_1$, id
–		$X \rightarrow YZ$

Table 4.7 Implementation Using Stack

From the bottom, using productions and reducing them in the stack, we reach to head X.
This shows that "int a, b, c" is accepted by the given SDT.

Note

- During inherited attribute evaluation, semantic action may be present in between the production variables. In such case, semantic action can be replaced by non-terminal pointing to E.
  For e.g: A → B {print '+'}CD
  This can be converted simply as
  A → BMCD
  M → E {print '+'}

Q.11 **Construct SDT to create a syntax tree for the given arithmetic expression.**
I/P : Z = 10 + 2 * 5
Output:

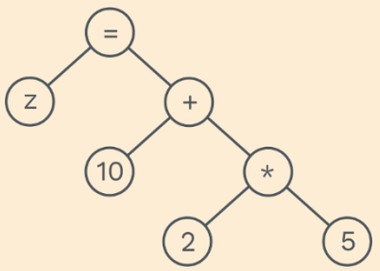


Fig. 4.14

Sol: **To construct SDT, we use a function make-node in the semantic action.**
Make-node (left-pointer, operator, right-pointer) is a function which returns the node address.

P → T = R	{P.ptr = make-node (T.ptr, =, R.ptr) and return(P.ptr)}
R → R + S	{R.ptr = make-node (R_1.ptr, +, S.ptr)}
R → S	{R.ptr = S.ptr}
S → S * Q	{S.ptr = make-node (S_1.ptr, *, Q.ptr)}
S → Q	{S.ptr = Q.ptr}
Q → num	{Q.ptr = make-node (null, num, null}
T → id	{T.ptr = make-node (null, id, null}

P.ptr, R.ptr, S.ptr, T.ptr and Q.ptr are the pointers that store the address of the node.

Parse Tree frame the above SDT

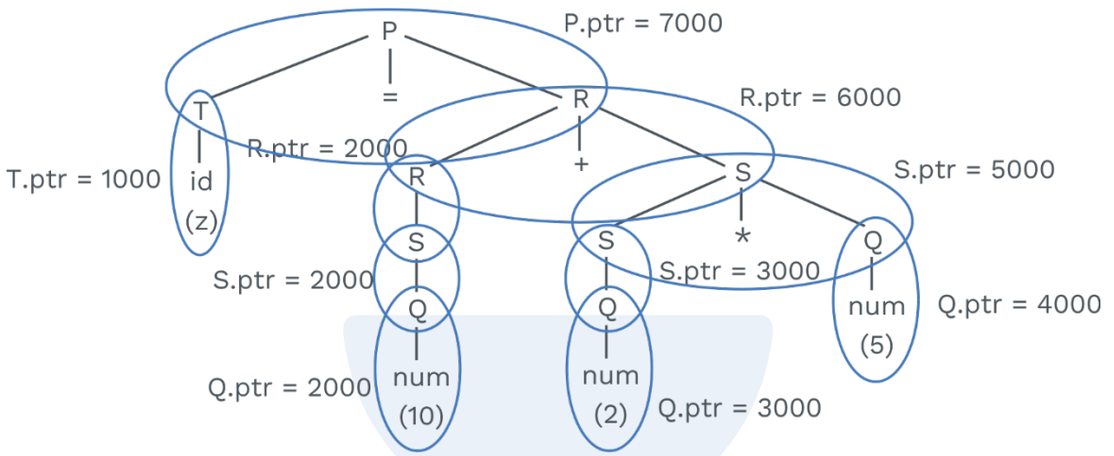


Fig. 4.15 Tree

1000, 2000, 3000, 4000, 5000, 6000, 7000; we have assumed that these are the addresses of the node returned by the make-node function after the formation of the node to the respective pointers.

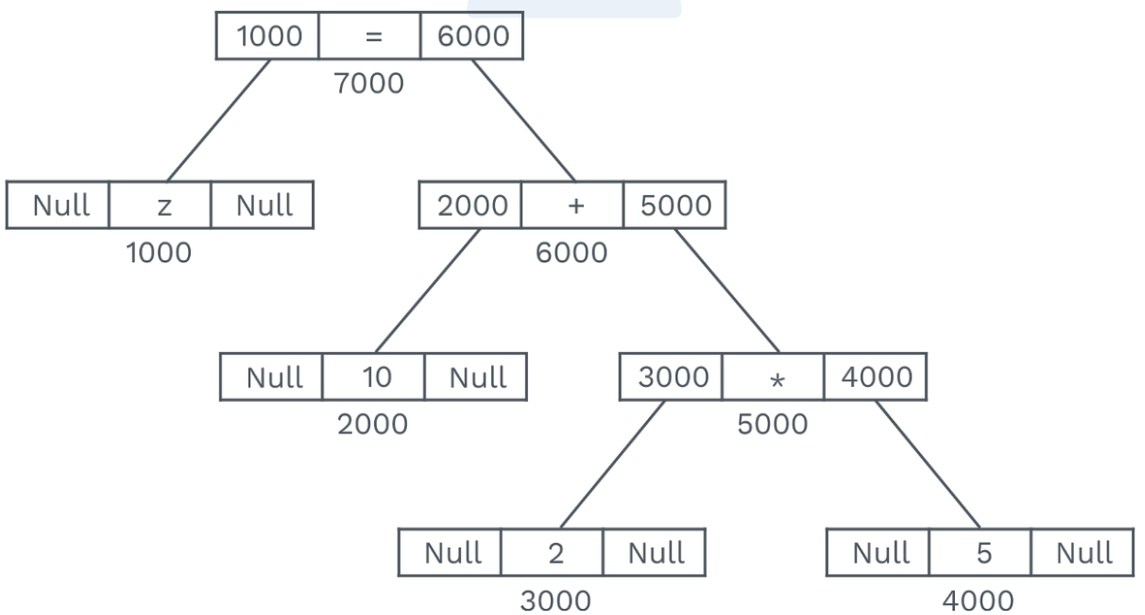


Fig. 4.16 Tree

Q.12 **Create syntax-directed translation for conversion of expression from infix to postfix.**
Input: a = p + q * r – s ÷ t
Output: xpqr*+st÷ – =

Sol:

P → Q = R	**{print (=)}**
R → R + S	{print (+)}
R → R – S	{print (–)}
R → S	{no semantic action}
S → S * Q	{print (*)}
S → S ÷ Q	{print (÷)}
S → Q	{no semantic action}
Q → id	{print (id)}

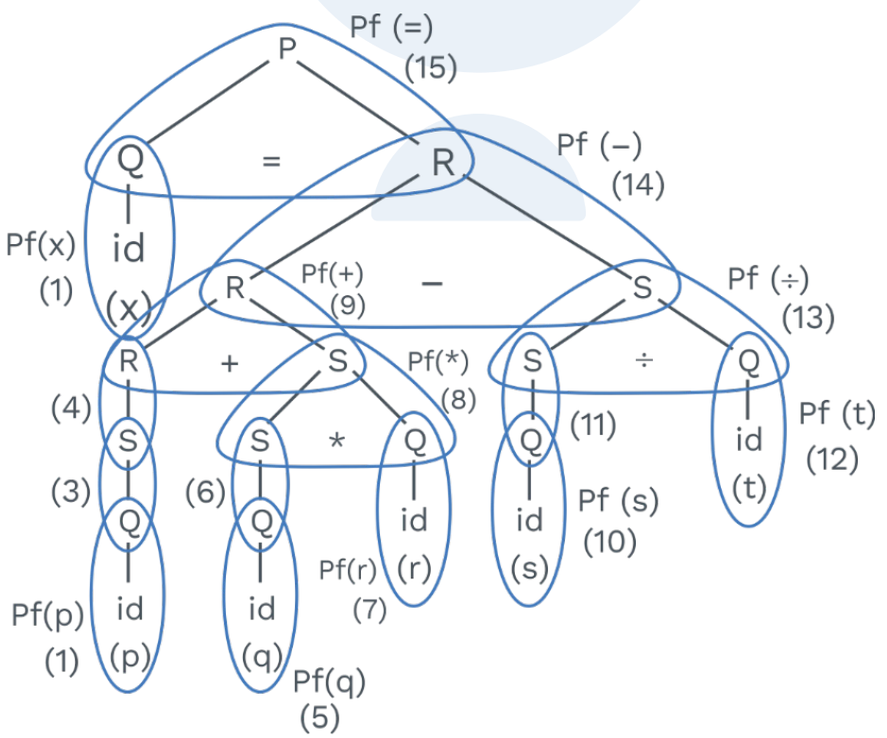


Fig. 4.17 Tree

Previous Years' Question

Consider the syntax directed definition shown below:

$S \rightarrow$ id: = E {gen (id.place = E.place;);}

$E \rightarrow E_1 + E_2$ {t = newtemp ( ); gen(t = E_1.place + E_2.place;); E.place = t}

$E \rightarrow$ id {E.place = id.place;}

Here ; gen is a function that generates the output code, and newtemp is a function that returns the name of a new temporary variable on every call. Assume that is t_1 are the temporary variable name generated by newtemp.

For the statement 'X : = Y + Z' the 3-address code sequence generated by this definition is.

a) X = Y + Z

b) t_1 = Y + Z ; X = t_1

c) t_1 = Y ; t_2 = t_1 + Z ; X = t_2

d) t_1 = Y ; t_2 = Z ; t_3 = t_1 + t_2 ; X = t_3

Sol: b) **[GATE: CS-2003]**

4.6 EVALUATION ORDERS FOR SDD's

- For evaluating the order of the attribute instance in a given parse tree dependency graph is a useful tool.
- While an annotated parse tree shows the values of attribute dependency graph helps in determining how those values can be computed.

Dependency graph:

Definitions

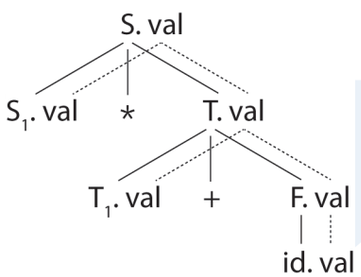
A dependency graph depicts the flow of information among the attribute instances in a particular parse tree, an edge from one attribute instance to another means that the value first is needed to be computed the second.

SOLVED EXAMPLES

Q.13 **Find the grammar and corresponding semantic rules to construct the dependency graph accordingly.**

$S = S_1 * T$; $S.val = S_1 val * T.val$
$T = T_1 + F$; $T.val = T_1.val + F.val$
$F = id$; $F.val = id.val$

Sol:



Note

- As a convention, we shall show the parse tree edges as dotted lines while the edges of the dependency graph with solid lines.

Ordering the evaluation of attributes:

- The order of the evaluation of the attributes at every node in the parse tree is characterized using a dependency graph.
- Topological sort is applied on the dependency graph to get the order of evaluation of attributes.
- In topological sort, first we choose the node whose in-degree is zero and remove it from the dependency graph and again select one node whose in-degree is zero.
- In this manner, we get a sequence of nodes and that sequence tells the order of evaluation of attributes in the SDT's.

SOLVED EXAMPLES

Q.14 **Given a dependency graph:**

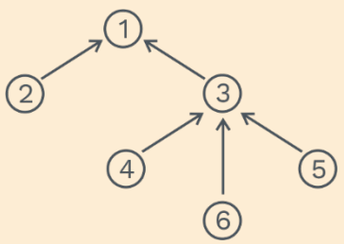


Fig. 4.18 Tree

Which among the following is topological order of given graph.

a) 2, 4, 6, 5, 3, 1 **b) 6, 3, 5, 4, 2, 1**

c) 2, 4, 5, 6, 3, 1 **d) 5, 4, 6, 2, 3, 1**

Sol: **Option a, c, d**

Option b) : 6, 3, 5, 4, 2, 1

We can remove node '6' first because it has a degree as 0.

But after removal of 6, we can not remove node 3 because it has in edge from node 4 and node 5.

Thus option b) is not correct.

Q.15 **Given a dependency graph:**

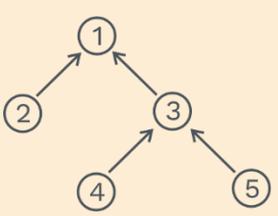


Fig. 4.19 Tree

How many orders of evaluations are possible ?

Sol: **Total number of evaluation orders is equal to the total number of topological order possible on the dependency graph.**

Following are the topological order possible on the dependency graph.

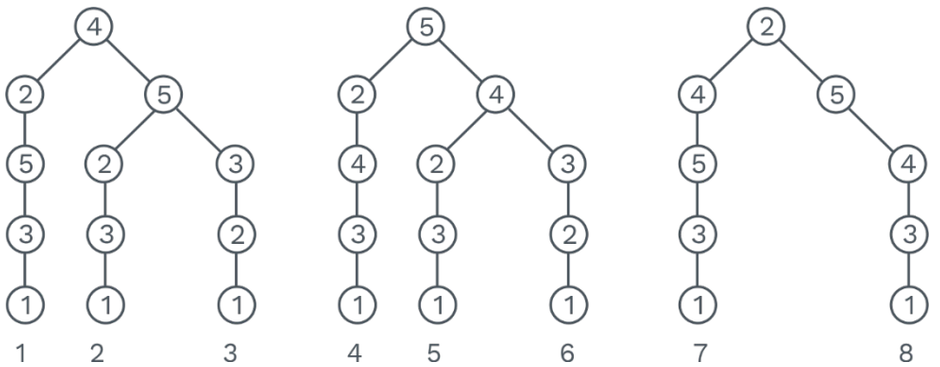


Fig. 4.20 Tree

Therefore total 8 evaluations order are possible.

Q.16 **Given a SDT:**

P → P = Q	**{P.val = Q.val}**
Q → Q + R	**{Q.val = Q_1.val + R.val}**
Q → R	**{Q.val = R.val}**
R → R * P	**{R.val = R.val * P.val}**
R → P	**{R.val = P.val}**
P → id	**{P.val = id}**

Input : x = u * v + w * t
Draw a parse tree and its dependency graph. {Operators have their usual precedence and associativity}

Sol:

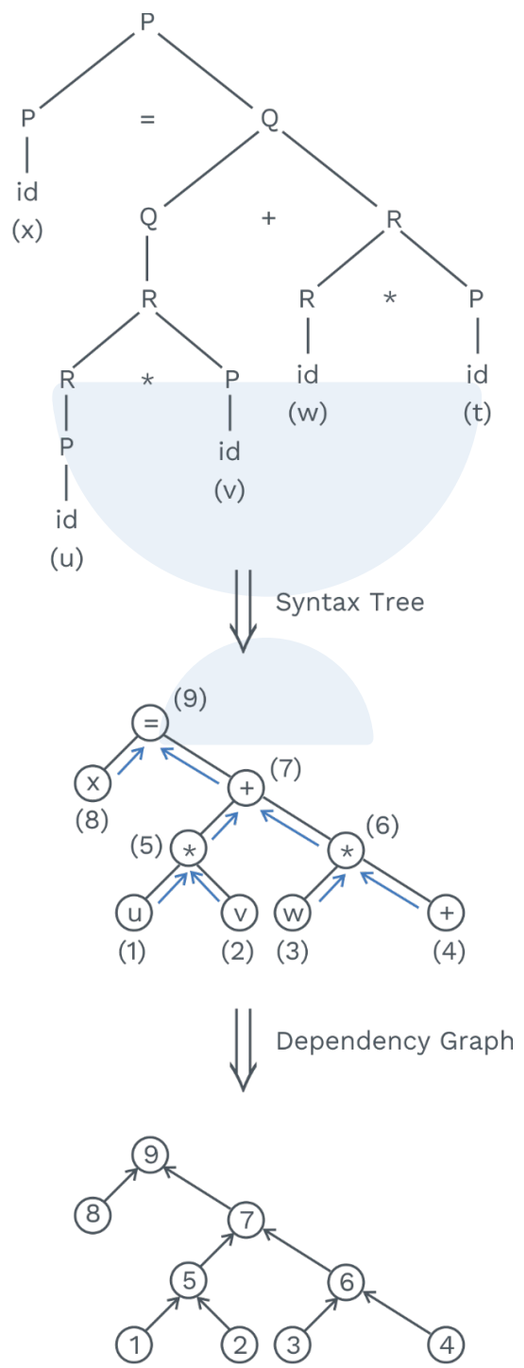


Fig. 4.21 Tree

Order of evaluation = Topological sort on the dependency graph.
= 8 1 2 5 3 4 6 7 9

Note

- There can be more than one topological order thus, the order of evaluation is not unique.
- Topological order equal to post-order traversal is closest to programming.
- If a cycle is present in the dependency graph, then there will be no evaluation order for that SDT.

Chapter Summary

- Syntax directed definition (SDD) is context-free grammar together with attributes and rules.
- Attributes are of two types:
  i) Synthesised attributes: Depend upon children.
  ii) Inherited attributes: Depend on siblings and parents.
- Syntax tree is a tree in which internal nodes represent operators and leaf nodes represent operands.
- Construction of syntax tree requires 3 functions:
  i) make-node (Operator, left-pointer, right-pointer)
  ii) make-leaf (id, entry to symbol table)
  iii) make-leaf (num, value)
- Associativity and precedence of the operators can be known by the analysis of the parse tree.
- Basically, there are two types of SDT:
  i) S-attributed SDT: only synthesised attributes are present in SDT.
  ii) L-attributed SDT: Both synthesised and inherited attributes are present but inherited attributes should not depend on the right sibling.
- Dependency Graph: It depicts the flow of information among attribute instances in a particular parse tree.
- Evaluation order of SDT's depend on the dependency graph.

5 Intermediate Code Generation

5.1 BASIC OF INTERMEDIATE CODE GENERATION

- Compiler may generate sequential intermediate representations to translate a source code to a target machine code language.

Source program → High level intermediate representation → ..... → Low level intermediate representation → Target code

- Source program is associated with "High level intermediate representation", and the target machine code is with "Low level intermediate representation".
- Syntax tree is high level intermediate representation which shows the natural hierarchical structure of the source program. They are well suited for the tasks like static type checking.
- Register allocations and instruction selection these kinds of machine dependent jobs can be done efficiently using "Low level intermediate representation".

5.2 REPRESENTATION OF INTERMEDIATE CODE

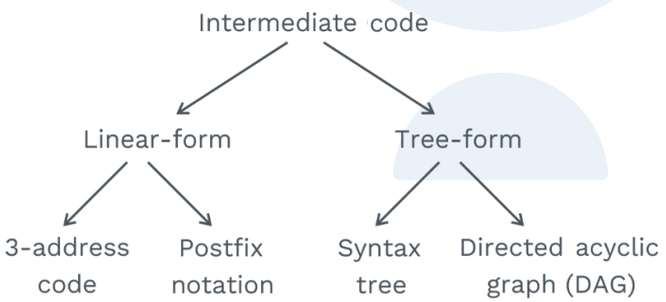


Fig. 5.1 Representation of Intermediate Code

3-Address code:

- Three address code is built from two concepts: address and instructions.
- The implementation of three address codes consists of records as well the address fields
- Three address instructions consist of at most three addresses in a line of code.
- An address can be one of the following:
  - **1)** A name: For convenience, we allow the source program name to appear as in the 3-address code.
  - **2)** A constant: Constants and variables are the mandatory fields with which compiler has to deal with
  - **3)** A compiler-generated temporary variable: Temporary variable is mainly for optimising compiler to store intermediate results each time a temporary variable is needed.

- **Types of 3-address code:**

i)	x = y op z	|| x stores the result of y operator z
ii)	x = op y	|| x stores result of unary operator y
iii)	x = y	|| x stores y
iv)	x = a [i]	|| x stores value of array a at index i
v)	a [i] = x	|| Stores the value of x in an array a at index i
vi)	x = f(a)	|| x stores the value return by function 'f' with parameter 'a'
vii)	if x < y goto z	{conditional goto}
viii)	goto x	{Unconditional goto}
ix)	z = x[i] [j]	|| This is not 3-address code representation because it uses variables z, x, i, j
x)	x = f(a, b)	|| This is not 3-address code representation because it uses 4 variables x, f, a, b
xi)	x = &a	|| address assignment

Types of 3-address code representation:

Basically, there are 3-ways of representing 3-address code

1) Quadruple
2) Triple
3) Indirect triple

Let us consider an expression:

$$a = b * - c + b * - c ;$$

Three address code:

$$t_1 = \text{minus } c$$

$$t_2 = b * t_1$$

$$t_3 = \text{minus } c$$

$$t_4 = b * t_3$$

$$t_5 = t_2 + t_4$$

$$a = t_5$$

1) Quadruples representation of 3-address code:

	Op	arg_1	arg_2	result
0.	minus	c	–	t_1
1.	*	b	t_1	t_2
2.	minus	c	–	t_3
3.	*	b	t_3	t_4
4.	+	t_2	t_4	t_5
5.	=	t_5	–	a

Fig. 5.2 Quadruples Representation of 3-Address Code

For readability, we use actual identifiers like a, b, and c in the field arg_1, arg_2 and result instead of pointers to their symbol table entries.

- Op field contains the internal code for the operator.
- arg_1 and arg_2 contain the operand.
- result field stores the result.

2) Triplet representation of 3-address code:

- A triplet has only tree fields Op, arg_1, arg_2
- Using triplet, we refer to the result of an operation x op y by its position rather than an explicit temporary name.
- Above 3-address code representation in triplet.

	Op	arg_1	arg_2
0.	minus	c	–
1.	*	b	(0)
2.	minus	c	–
3.	*	b	(2)
4.	+	(1)	(3)
5.	=	a	(4)

Fig. 5.3 Triplet Representation of 3-Address Code

Triple representation of a = b * – c + b * – c ;

3) Indirect triple representation of 3-address code:

- Indirect triples do not list the triples, but they list the pointers of triples. For eg: an array of instructions to list pointers to triples in the desired order.
- An optimising compiler is able to reorder the instruction in triples without affecting the triples themselves.

	Instruction
35.	(0)
36.	(1)
37.	(2)
38.	(3)
39.	(4)
40.	(5)
	.....

	Op	arg_1	arg_2
0.	minus	c	–
1.	*	b	(0)
2.	minus	c	–
3.	*	b	(2)
4.	+	(1)	(3)
5.	=	a	(4)

Fig. 5.4 (Indirect Triple Representation of 3-Address Code)

Indirect-triple representation of a = b * (–c) + b * (–c) ;

- Instruction is an array that contains a pointer to triple in the desired order.

Rack Your Brain

What could be the benefit of quadruples over triple?

SOLVED EXAMPLES

Q.1 **Write 3-address code for the following program**

```
x = 2 ;
for (x = 25 ; x ≤ 100 ; x ++)
{
   x = p + q + r ;
}
```

Sol:

1000: x = 2

1001: x = 25

1002: if x ≤ 100 goto 1004.

1003: goto 1009
1004: $t_1 = p + q$
1005: $t_2 = t_1 + r$
1006: $x = t_2$
1007: $x = x + 1$
1008: goto 1002
1009: _____

Q2 **3-address code representation in quadruple form for the expression**
x = b + b + b + b + b

Sol: 3-address code representation:

$t_1 = b + b$
$t_2 = t_1 + b$
$t_3 = t_2 + b$
$t_4 = t_3 + b$
$x = t_4$

Quadruple form representation of 3-address code:

	Op	arg_1	arg_2	result
(0)	+	b	b	t_1
(1)	+	t_1	b	t_2
(2)	+	t_2	b	t_3
(3)	+	t_3	b	t_4
(4)	=	t_4	–	x

Fig. 5.5 (Quadruple Representation)

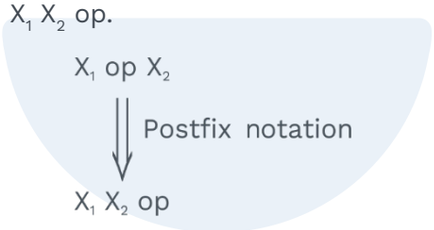
Q3 **3-address code representation for expression**
x = (a + b) * a * b * (c + d) ;

Sol: 3-address code for given expression

$t_1 = a + b$
$t_2 = t_1 * a$
$t_3 = t_2 * b$
$t_4 = c + d$
$t_5 = t_3 * t_4$
$x = t_5$

Postfix notation:

- Let us consider there are two expressions X_1 and X_2 and there is an operand op which are represented as X_1 op X_2 then in postfix notation it will be represented as X_1 X_2 op.



SOLVED EXAMPLES

Q4 **Convert the following expression in postfix notation.**

x = a * b + c * d + d * e

where * has higher precedence, and both are left-associative, and '=' has the least precedence.

Sol: * has higher precedence thus they will be calculated first.

∴ (x) = (a*b) + (c*d) + (d*e)
(x) = (ab*) + (cd*) + (de*)
(x) = ((ab*cd*)+) + (de*)
(x) = (ab*cd*+de*+)
xab*cd* + de* + =

Q5 **Given an expression**
x $ y $ z # x $ b # t ⊕ d
Convert the expression in postfix notation when the precedence order of the operator is $ > # > ⊕ and all the three operators are left-associative.

Sol: $ has highest precedence and ⊕ has the least precedence.

∴ (x y $) $ z # x $ b # t ⊕ d

⇒ (x y $ z $) # (x $ b) # t ⊕ d

⇒ (x y $ z $) # (x b $) # t ⊕ d

⇒ (x y $ z $ x b $ #) # t ⊕ d

⇒ (x y $ z $ x b $ # t #) ⊕ d

⇒ x y $ z $ x b $ # t # d ⊕

Post order notation ⇒ x y $ z $ x b $ # t # d ⊕

Q6 **x = a + uminus b+ uminus d**
All the operators have these usual precedences and associativity then convert given expression in postfix notation uminus represent uniary minus operator.

Sol: As we know, uniary operator has the highest precedence

∴ Order of precedence uminus > + > =

Postfix notation: x = a +(b uminus) + (d uminus)

x = (a b uminus +) + (d uminus)

x = (a b uminus + d uminus +)

x a b uminus + d uminus + =

Syntax tree:

- Syntax tree is the representation in the form of tree in which each internal node represents the operators, and the leaf node represents the operand.
- During the formation of a syntax tree of any code, we should design tree in such a form that lower precedence operator should be on the top level and higher precedence should be on lower levels.

SOLVED EXAMPLES

Q7 **x = a * b + c**
Syntax tree for the above expression when all the operators have their usual associativity and precedence.

Sol: The Precedence order of operators:
* > + > =
Syntax Tree:

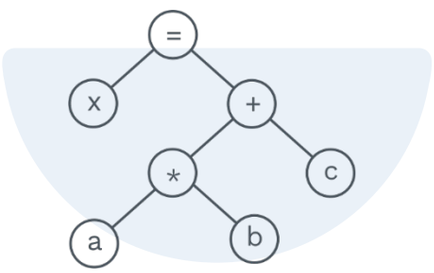


Fig. 5.6 Syntax Tree

Q8 **x = a * b ÷ c * d**
Syntax tree for the above expression when operators have their usual precedence and associativity.

Sol: Both '*' and '÷' have the same precedence. Thus, syntax tree will be formed on the basis of associativity.
Syntax Tree:

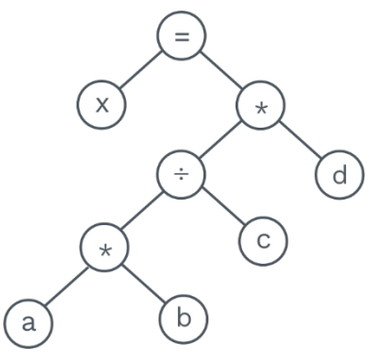


Fig. 5.7 Syntax Tree

Directed acyclic graphs:

- It is similar to a syntax tree except that common sub-expression are represented by a single node.
- Common subexpression is represented with a single node to get the optimised code, i.e., instead of evaluating again and again, we can evaluate common expression only once.

SOLVED EXAMPLES

Q9 **x = t + t + t + t**
For the given expression form DAG.

Sol: Here identifier 't' is used many times in the expression, and 't+t' is evaluated twice so in the graph, we will use only one node for the identifier 't' and 't+t' evaluation.

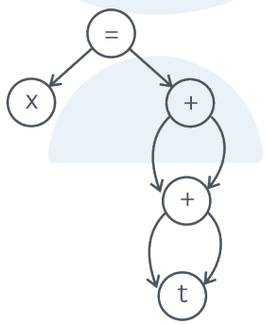


Fig. 5.8 DAG

Q10 **x = a * b * c +b * a**
All operator have their usual precedence and associativity, then form the DAG for the above expression.

Sol: Here (a * b) can be used again as it is repeated in the expression there again forming the node for (a * b), we have reused it.

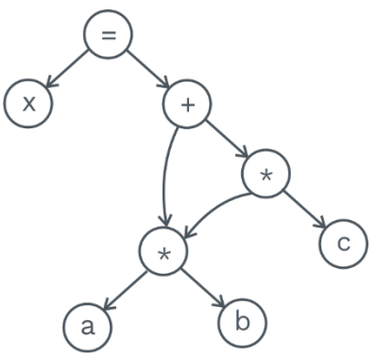


Fig. 5.9 DAG

5.3 STATIC SINGLE ASSIGNMENT

- Static Single Assignment (SSA) is the property of intermediate code representation where each variable should be assigned exactly once and each variable should be defined before its use.
- Each expression must be in 3-address representation only and following the above two conditions.

SOLVED EXAMPLES

Q11 **Check whether given 3-address code in SSA representation or not.**

$T_1 = a * a$
$T_2 = y * a$
$T_2 = z * b$

Sol: Given 3-address code is not in SSA representation because variable T_2 is assigned more than one time.

Q12 **Check whether the given code is in SSA representation or not.**

$T_1 = a * b$
$T_2 = c * T_1$
$T_3 = d + T_2$
$T_4 = T_1 + T_2$

Sol: Given 3-address code is in SSA representation because each variable has been assigned exactly once.

Q13 **Write SSA representation of given expression**
x = b * c + d + a + b

Sol: '*' is computed first because multiplication has higher precedence than addition.

$T_1 = b * c$
$T_2 = T_1 + d$
$T_3 = T_2 + a$
$T_4 = T_3 + b$
$x = T_4$

Q14 **Given a three address code**
x = a + b
y = c + d
x = y + d
c = c + b
z = c + y
Convert the given 3-address code into a single static assignment.

Sol: Here in this 3-address code representation, variables x and c are assigned twice; thus, it is not in SSA.

SSA representation of above code:

$x = a + b$
$y = c + d$
$T_1 = y + d$
$T_2 = c + b$
$z = T_2 + y$

Previous Years' Question

Consider the following code segment

$$x = u - t;$$
$$y = x * v;$$
$$x = y + w;$$
$$y = t - z;$$
$$y = x * y;$$

Minimum number of variables required to convert the above code in single static assignment.

Sol: 10 variables **[GATE: CS-2016]**

Previous Years' Question

Consider the following intermediate program in three address code:

$$p = a - b$$
$$q = p * c$$
$$p = u * v$$
$$q = p + q$$

Which of the following corresponds to a static single assignment from the above code?

a)
$p_1 = a - b$
$q_1 = p_1 * c$
$p_1 = u * v$
$q_1 = p_1 + q_1$

b)
$p_3 = a - b$
$q_4 = p_3 * c$
$p_4 = u * v$
$q_5 = p_4 + q_4$

c)
$p_1 = a - b$
$q_1 = p_2 * c$
$p_2 = u * v$
$q_2 = p_4 + q_3$

d)
$p_1 = a - b$
$q_1 = p * c$
$p_2 = u * v$
$q_2 = p + q$

Sol: b) **[GATE: CS-2016]**

Previous Years' Question

Consider the basic block given below:

a = b + c

c = a + d

d = b + c

e = d – b

a = e + b

The minimum number of nodes and edges present in the DAG representation of the above basic block respectively are.

a) 6 and 6 **b)** 8 and 10

c) 9 and 12 **d)** 4 and 4

Sol: a) **[GATE: CS-2014]**

5.4 CONTROL FLOW GRAPHS

Backpatching:

There is a problem when writing the code for the flow of control statements that match the jump instruction with the target of the jump.

The solution to this problem is back patching, in which when we don't know the target of the jump statement, we leave it unspecified.

When we know the target address, then we fill that unspecified blank space.

For eg:

100: if a < 100 goto ____
101: goto ____
102: if a > 200 goto 104
103: goto ____
104: if a ! = b goto ____
105: goto ____

After backpatching 104 goes to instruction 102.

100: if a < 100 goto ____
101: goto 102
102: if a > 200 goto 104
103: goto ____

104: if a ! = b goto ____
105: goto ____

After backpatching 102 into instruction 101.

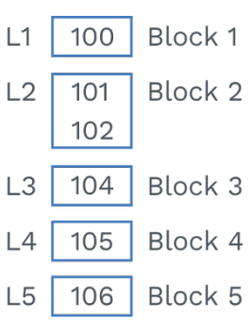
Leader:
1) The first statement of the intermediate code is the leader.
2) Target statement of the goto is the leader.
3) Next statement after goto is the leader.

Basic block:
- Basic block is the set of statements of intermediate code which consist of one entry point and one exit point in between which there can be no halt or jump.
- Each basic block consists of one leader.
- Basic block starts from the leader statement and ends one statement before the next leader.
- Number of Basic blocks = Number of leaders.

SOLVED EXAMPLES

Q15 **Draw the control flow graph of the given code.**
100: if x < 100 goto 104
101: x = a + b ;
102: x = x + 7 ;
104: if x > 200 goto 106
105: x = x + 1
106: z = x + 200

Sol:



Control flow graph:

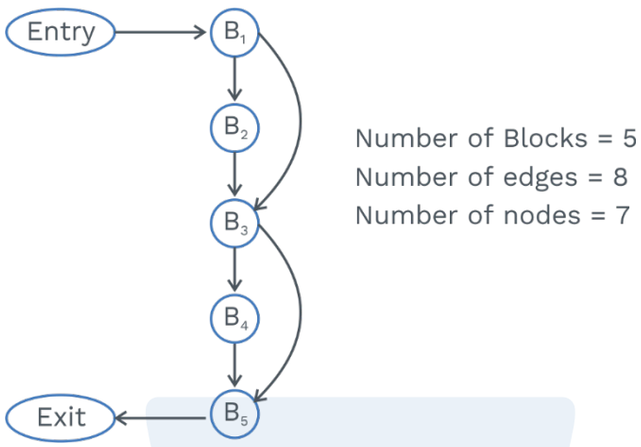


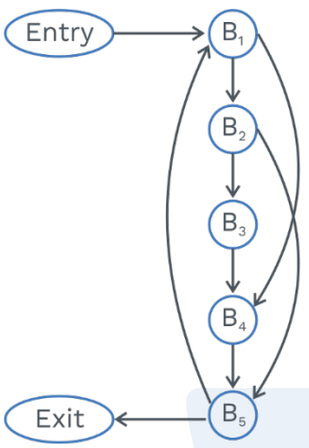
Fig. 5.10 Control Flow Graph

Q16 **100: i = 1 ;**
101: j = 2 ;
102: if i > 100 goto 106
103: i = i + 1 ;
104: if j > 100 goto 107
105: j = j + 1 ;
106: z = z + 200
107: z = z + 200
108: if z < 500 goto 101
The total number of nodes and edges will be there in the control flow graph of above code.

Sol:

L1	100 101 102	Block 1
L2	103 104	Block 2
L3	105	Block 3
L4	106	Block 4
L5	107 108	Block 5

Control flow graph:



Number of nodes = 7
Number of edges = 9

Fig. 5.11 Control Flow Graph

∴ Total number of nodes and edges = 7 + 9 = 16

Previous Years' Question

Consider the intermediated code given below:

1) i = 1
2) j = 1
3) $t_1 = 5 * i$
4) $t_2 = t_1 + j$
5) $t_3 = 4 * t_2$
6) $t_4 = t_3$
7) $a[t_4] = -1$
8) j = j + 1
9) if j < = 5 goto (3)
10) i = i + 1
11) if i < 5 goto (2)

The number of nodes and edges in the control-flow graph constructed for the above code respectively are

a) 5 and 7 **b)** 5 and 5 **c)** 6 and 7 **d)** 7 and 8

Sol: c)

Chapter Summary

- Intermediate code is used to translate the source code into the machine code. It lies between high level language and machine language.
- Intermediate code is present in two forms:
  - **i)** Linear form:
    - **a)** 3-address code
    - **b)** Postfix notation
  - **ii)** Tree form:
    - **a)** Syntax tree
    - **b)** Directed Acyclic Graph (DAG)
- 3-address code can have a maximum of 3-address in its expression and is built on two concept address and instruction.
- 3-address codes are of the following types:
  i) Quadruples
  ii) Tripple
  iii) Indirect tripple
- Postfix notation is the conversion of expression in the postfix form.
- Syntax tree represented in the form of tree where internal nodes represent operators and leaf nodes represent operands.
- Static single assignment (SSA) is the representation of the code in which each variable can only be assigned once, and the variable should be defined before its use.

6 Run Time Environment

6.1 INTRODUCTION

- Compiler must cooperate with the operating system and other system software to support abstraction (scope, binding, data types, the flow of control construct) on the target machine.
- To do so, the compiler creates and manages a run time environment in which it assumes its target programs are being executed.
- Environment deals with a variety of issues such as layout and allocation of the storage location for the object named in the source program, the mechanism used to access variables, the linkage between procedures, the mechanism for parameter passing, and other programs.
- Run time environment act as an interface between higher–level concepts of the programming language and lower concepts supported by the target machine.

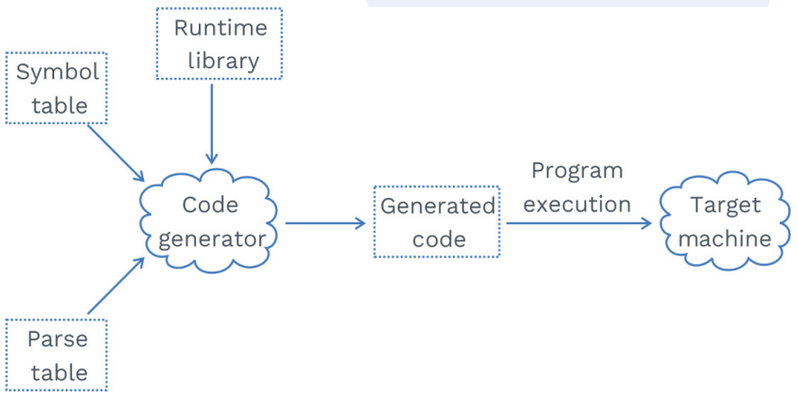


Fig. 6.1 Introduction

Procedures:
Procedures are the functions defined in the simplest form, which associates the identifiers with the statement (body of the procedure)

Activation tree:

- We represent the activation of the procedures during the running of an entire program by a tree called an activation tree.
- Each node corresponds to one activation, and the root is the activation of the main procedure that initiates the execution of the program.

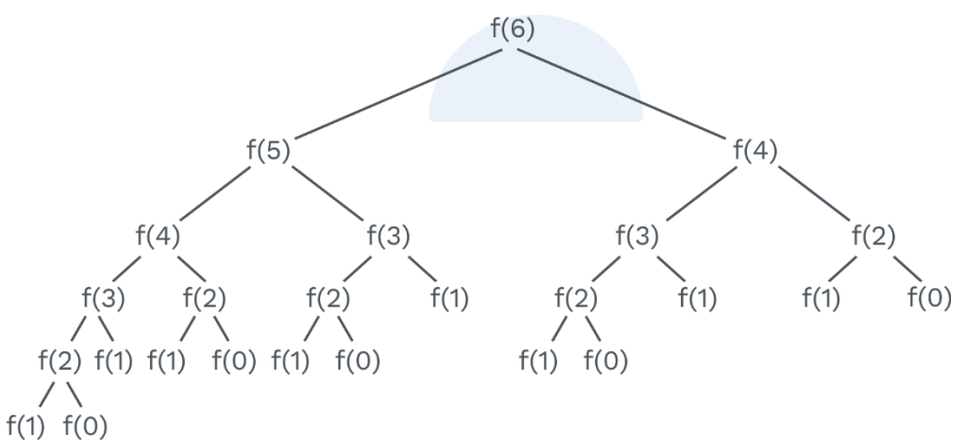
SOLVED EXAMPLES

Q1 **Given a program to print the FIBONACCI series:**

```
    f(n)                    int main ( )
{   if (n= = 0 || n= =1)    {       int n=6;
    return (n) ;            cout << f(n);
else                        getchar( ) ;
{   a=f(n-1);               return 0;
    b=f(n-2);               }
    c=a+b;
    return (c);
    }
}
```

Draw an activation tree for the given program.

Sol:



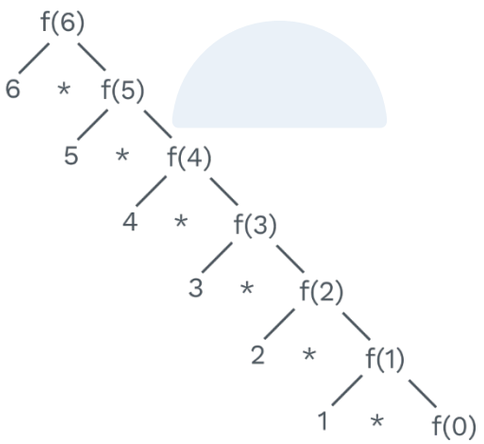
This is the activation tree in which f(n) represents the calling function during the execution of the program.

Q2 **Given a program for finding factorial:**

```
long fact(int n)
{   if (n–1)
    return n*fact(n-1);
    else
    return 1;
}
int main ( )
{   int n=6;
    printf (“Enter a positive integer:”);
    scanf (“%d”, &n);
    printf (“factorial of %d=%d”, n, fact(n));
    return 0;
}
```

Draw an activation tree for the function calls of the above code.

Sol:



Where f(n) equivalent to fact (n).

Control stack:

- Control stack maintains or tracks how the procedures are activated.
- When a procedure is called, it is pushed on the control stack.
- When a procedure completes its execution, it is popped out of the control stack.

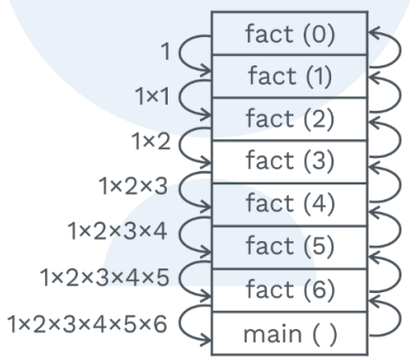
SOLVED EXAMPLES

Q1 **During the execution of the given code what will be the size of stack required.**

```
int fact(int n)
{   if n ≥1
    return n*fact(n-1);
    else
    return 1;
}
```

```
int main ( ) {
int n=6;
int factorial=fact(n);
Printf(" factorial of 6 = %d", factorial);
return 0;
}
```

Sol:



The size of the control stack that is required during the execution of the above program is 8 blocks of the stack.

Scope of declaration:

- Scope of declaration determines the declaration of an identifier when it appears in the text of the program.
- Portion of the program to which the declaration applies is called the scope of that declaration.
- Scope of declaration is of two types
  - **i)** Local scope: Scope which is limited to a function or small segment of the program.
  - **ii)** Global scope: The scope of a variable or identifier which is present in the entire program is known as a global scope.

At compilation time, with the help of the symbol table, we can determine whether the scope of the variable is local or global.

SOLVED EXAMPLES

Q1 **Given a code below:**

```
#include<stdio.h>
    int a ;
    void function1( )
    {       int b ;
            printf ('b');
    }
int main ( )
{   int c ;
    c=a+b;
    printf ("%d",c);
    return 0;
}
```

From the above code, identify variables that have local and global scope separately.

Sol: Variable 'a' has a global scope; its scope is limited to be an entire program Variables b and c have a local scope because the scope of variable b is limited to function 1 ( ) and scope of variable c is limited to function main ( ). Both b and c can not be used outside the main function.

Binding of names:

- Binding of name is a process in which any name present in the code is bound to the storage location.

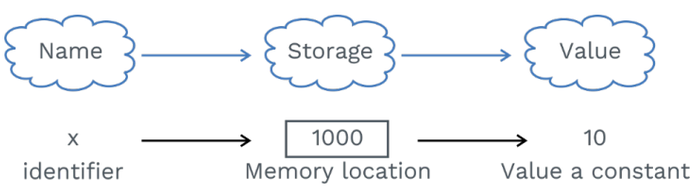


Fig. 6.2 Binding of Names

- A name 'x' an identifier is bound with the memory location referred by the address [1000]. This is known as the binding of names.

6.2 STORAGE ORGANISATION

- According to the perspective of the compiler writer, the executing target program runs in its own logical address space in which each program value has a location.
- The management and organisation of this logical address space are shared between the compiler, operating system, and target machine.
- Operating system maps the logical address into a physical address which is usually spread throughout the memory.

Fig. 6.3 Storage Organisation

Sub-division of runtime memory into code and data areas.

6.3 ACTIVATION RECORD

- Procedure calls and returns are managed by the control stack.
- When the procedure is called, its activation record is pushed into the stack with the root of the activation tree at the bottom of the stack.
- Contents of the activation record are:

Actual parameters
Returned values
Control link
Access link
Saved machine status
Local data
Temporaries

Table 6.1 Activation Record

A general activation record

1) Temporaries:
Temporary values such as those arising from the evaluation of expressions, in the case where those temporaries can not be held in a register.

2) Local data:
Local data is the one that is limited to its procedure only.

3) Save machine status:
A save machine status information about the status of the machine just before the call of the procedures. The information typically includes the return address (value of the program counter to which the called procedure must return) and the content of the register that was used by the calling procedure.

4) Access link:
The non-local data of other activation records can be referred by access link if required.
It mainly helps to access the data that are not local to the activation record.

Code memory:
During compilation time, generated target code size is fixed thus; target code can be placed at the area of the code, which is statically determined. This static area of the code is usually present at the lower end.

- **Static data:**

The size of some program data objects, such as global constants, and the data generated by the compiler, such as information to support garbage collection, may be known at compile-time, and these data can be placed in another statically determined area known as static data.

- **Stack and heap:**
- To maximise the utilisation of space at the run time, two areas stack and heap at opposite ends of the address space.
- These two areas grow towards each other as needed. In usual practice, the stack grows towards lower addresses, and heap grow towards higher addresses.
- Stack is used to store data structures called activation records that get generated during the procedural calls.
- Many programming languages allocate and deallocate data under program control, for eg: C has a function malloc and free. The heap is used to manage this kind of data.

Rack Your Brain

What is the reason for statically allocating as many data objects as possible?

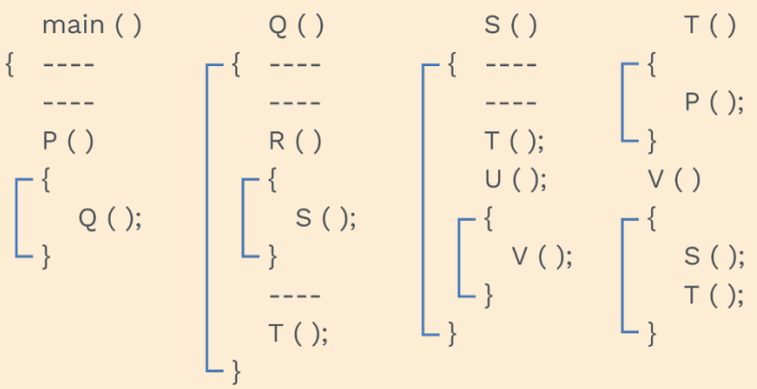
5) Control link:
The control link points to the activation record of the caller.

6) Returned values:
Returned values are the values that are returned by the functions.
Not all procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.

7) Actual parameter:
The actual parameters used by the calling procedure. Commonly, these values are not placed in the activation record but rather in the register, when possible, for greater efficiency.

SOLVED EXAMPLES

Q1



For the above program find the access link and control link for each function.

Sol: Access link of the function is the one in which the function is defined.

Function	Access link
main ( )	NULL
P ( )	main ( )
Q ( )	main( )
R ( )	Q ( )
T ( )	main ( )
S ( )	main ( )

U ( )	S ( )
V ( )	main ( )

The control link of a procedure is the procedure which is called inside it.

Function	**Control link**
main ( )	NULL
P ( )	Q ( )
Q ( )	T ( )
R ( )	S ( )
S ( )	T ( )
T ( )	P ( )
U ( )	V ( )
V ( )	S ( ), T ( )

6.4 STORAGE ALLOCATION STRATEGIES

There are three storage allocation techniques:

1) Static storage allocation
2) Stack storage allocation
3) Heap storage allocation

Static storage allocation:

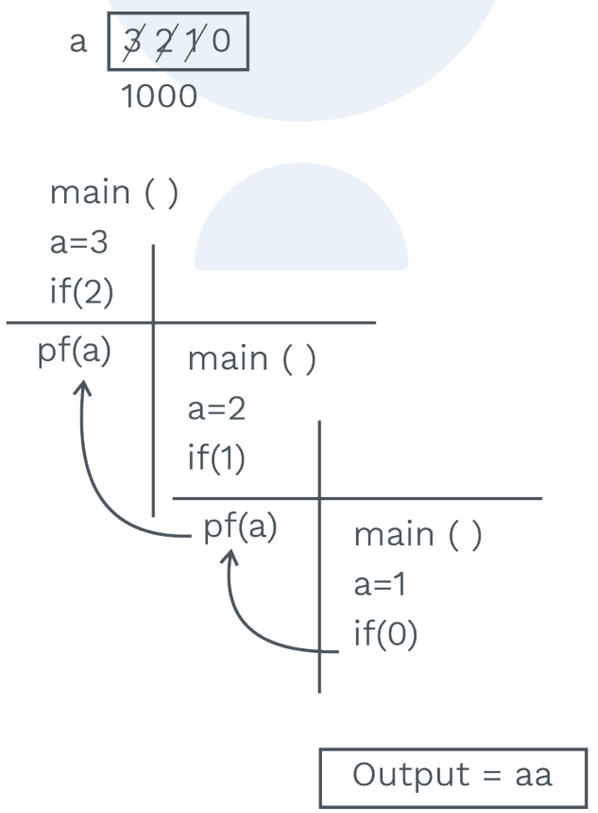
- If we do static allocation of any variable, then we put the keyword static before the data type of the variable.
- For static variables, memory is allocated in the static area, and it will be allocated only once.
- For static variable, memory will be allocated at compilation time only.
- initialisation of static variables can be done only once.
- Recursion does not support static storage allocation.
- Dynamic data structures are also not supported in static storage allocation.
- For static variables, its binding is done at compile-time and it can not be changed at runtime.
- Binding is mapping the variable to its storage location, if it is done at runtime, then it is known as runtime binding.

SOLVED EXAMPLES

Q1 **What will be the output for the given code:**

```
void main ( )
{   static int a=3;
    if (- -a)
    {       main ( );
            printf (a);
    }
}
```

Sol: Memory allocation in static area thus can be initialised only once.



Output = aa

NOTE

If in the above program, variable 'a' is not static, then the given function will never terminate because whenever main ( ) will be called again a new memory space will be allocated to variable 'a', and it will again be initialised to '3' thus if condition will never fail.

Stack storage allocation:

- Whenever the procedure is called, its activation record is pushed into the stack.
- Whenever the procedure finished its execution, its activation record is popped out from the stack.
- One of the most important use of stack storage allocation is that it supports recursion.
- Locals are contained in the activation record, so they have bound a new storage space each time a function is called.
- Drawback of stack storage allocation is that dynamic data structures are not supported.
- One major drawback with stack storage allocation is that once the procedure completed its execution, it popped out of the stack, and we can not get it back if it is required afterwards.

SOLVED EXAMPLES

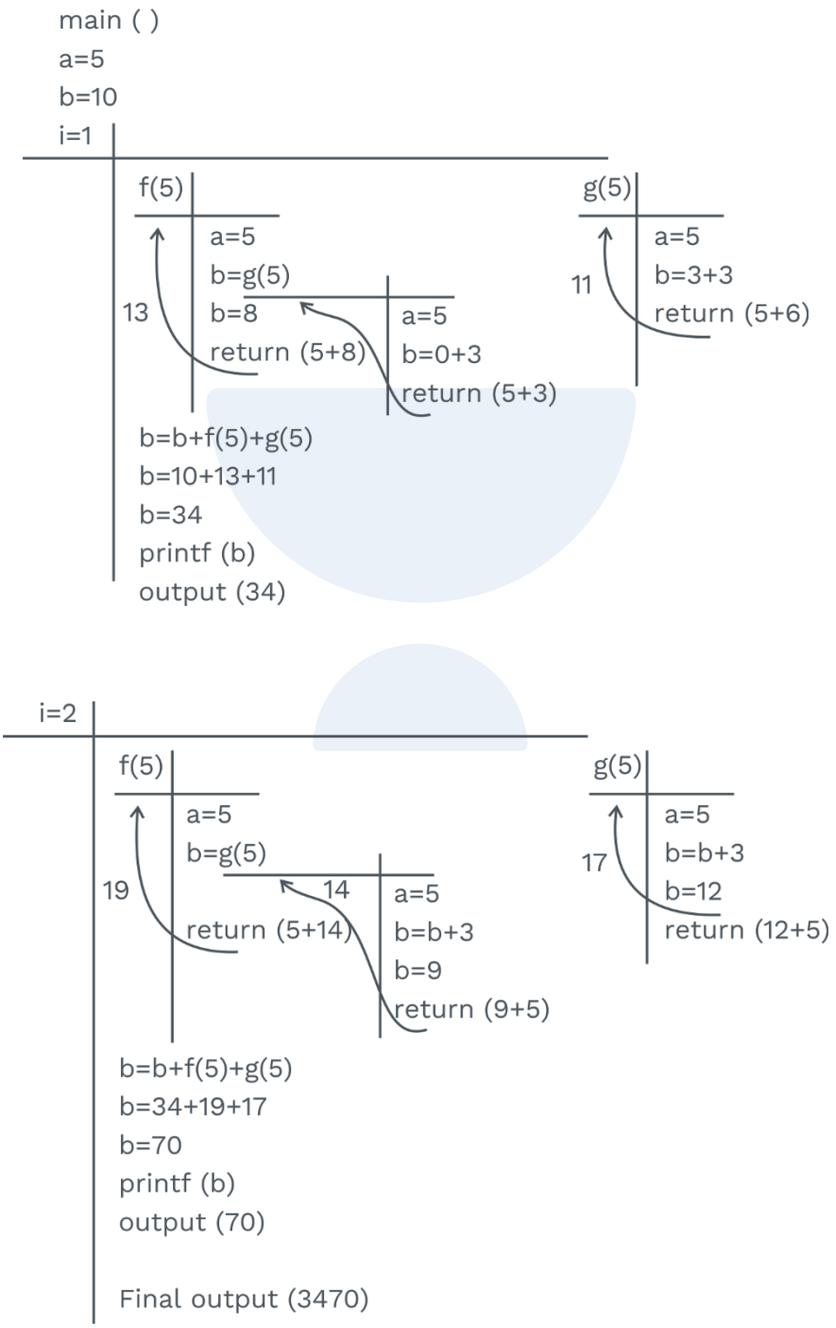
Q1 **Given a code if stack storage allocation is used than what will be the output.**

```
void main ( )
{   int a=5, b==10, i;
    for (i=1, i≤ 2;i++)
    {       b+=f(a) + g(a);
            printf (b);
    }
}
```

```
f (int a)
{       int b;
        b=g(a);
        return (a+b);
}
```

```
g (int a)
{       static int b=0;
        b+=3;
        return (a+b);
}
```

Sol:


main ( )
a=5
b=10
i=1
f(5)
a=5
b=g(5)
b=8
return (5+8)
13
a=5
b=0+3
return (5+3)
g(5)
a=5
b=3+3
return (5+6)
11
b=b+f(5)+g(5)
b=10+13+11
b=34
printf (b)
output (34)
i=2
f(5)
a=5
b=g(5)
19
return (5+14)
14
a=5
b=b+3
b=9
return (9+5)
g(5)
a=5
b=b+3
b=12
return (12+5)
17
b=b+f(5)+g(5)
b=34+19+17
b=70
printf (b)
output (70)
Final output (3470)

Heap storage allocation:

- In some programs, we need to retain the local names even after the activation of the procedure ends; in such cases, we use heap memory allocation of such local names.
- Major advantage of heap memory allocation is that it supports dynamic data structures and recursion.
- Both heap and stack can be allocated and deallocated at runtime.
- Heap memory remains allocated until you explicitly free it.

6.5 SYMBOL TABLE IMPLEMENTATION

Introduction:

- One of the important data structures that the compiler uses to know information about identifiers in the source program.
- Lexical analyzer and parser populate the symbol table with information, and further code generator and optimiser use those information.
- Variable, defined constants, functions, tables, structures, etc., are stored in the symbol table.
- Even for the same language, its symbol table may vary depending upon its implantation.

Information in a symbol table:

- Identifier's name.
- Identifier's type.
- Identifier's offset.
- Scope: The specific program region where the current definition is valid.
- Other attributes : arrays, return value, records,
- parameters, etc.

Operations on the symbol table:

- **Lookup:** Lookup operation is used, whenever an identifier is seen, it is needed to check its type or create a new entry.

Previous Years' Question

Which of the following statements are correct?

1) Static allocation of all data areas by a compiler make, it impossible to implement recursion.
2) Automatic garbage collection is essential to implement recursion.
3) Dynamic allocation of activation records is essential to implement recursion.
4) Both heap and stack are essential to implement recursion.

a) 1 and 2 only
b) 2 and 3 only
c) 3 and 4 only
d) 1 and 3 only

Sol: d) **[GATE: CS 2014]**

Rack Your Brain

Why does static allocation not support recursion?

- **Insert:** Insertion usually occurs in lexical or syntax analysis phases in which we add new names to the table.
- **Modify:** Sometimes, everything about the identifier is not present at the time when it is defined thus, we need to update it later.
- **Delete:** It is needed when the procedure body ends.

Various issues in symbol table design:

- **Format of entries:** From linear arrays to tree-structured tables, various formats of entries are present.
- **Access methodology:** Linear search, binary search, hashing, etc.
- **Location of storage:** Primary memory partial storage in secondary storage.
- **Scope issues:** Identifier whether it has a global scope or local scope.

Commonly used techniques for symbol table:

- Linear table
- Hash table
- Ordered list
- Trees

Linear list implementation of symbol table:

- Simple array of records with each record corresponding to an identifier in the program.
- Eg:

int a, b, c, ;
real z ;

---

procedure abc

L_1 ;

Name	Type	Location
a	integer	offset of a
b	integer	offset of b
c	integer	offset of c
z	real	offset of z
abc	procedure	offset of abc
L_1	Lable	offset of L_1

Table 6.2 Linear List Implementation of Symbol Table

- Lookup, insert, modify can take O(n) where 'n' is the number of entries because it will simply apply linear search on the array.
- Insertion can be done in O(1) time when we know the pointer to the next free location.

Ordered list implementation of the symbol table:

- It is simply a variation of the linear list in which the list is organised either in ascending or descending order.
- Therefore, binary search can be applied, which takes O(log(n)) times for 'n' entries.
- Thus, lookup and modification can be done in O(logn) time, but insertion will take O(nlogn) time because we have to sort the list.

Tree implementation of the symbol table:

- Node of the tree represents each entry of the symbol table.
- Based on the string comparison of names, entries lesser than a reference node are kept in the left subtree, otherwise in the right sub-tree.
- Lookup time for height-balanced tree is O(logn).

Proper height-balanced techniques should be used.

Eg:

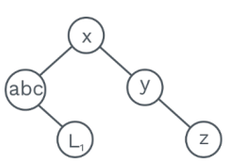


Fig. 6.3 Tree Implementation of the Symbol Table

Hash table implementation of the symbol table:

- Most of the compilers used hash tables for the implementation of the symbol table.
- mapping is done using a hash function that results in a unique location in the table organised as an array.
- Access time for a hash table is O(1)
- Improper hash function results in mapping several symbols to the same location. To overcome this, a proper collision resolution technique is used.
- To keep collisions reasonable, the hash table is chosen to be of size between n and 2n for n keys.

Rack Your Brain

Which is the best data structure for implementation of symbol table and why?

Chapter Summary

- Runtime environment provides proper services to the executing processes by including the software libraries and environment variables.
- Activation of procedure during the running of the entire program represented in the form of the tree is known as the activation tree.
- Control stack keeps track of the activation of the procedures.
- Scope of declaration is of two types:
  i) Global scope of declaration: Identifier is defined for the entire program.
  ii) Local scope of declaration: Identifier is defined over a block or a procedure.
- Binding of name is the process in which we bind the name to a particular storage location.
- Activation record exists for each procedure; when the procedure is called activation record is pushed into the control stack, and when completed its execution pops out of the stack.
- General activation record consists of:
  i) Actual parameter
  ii) Returned values
  iii) Control link
  iv) Access link
  v) Saved machine states
  vi) Local data
  vii) Temporaries

- Subdivision of runtime memory is as follows:

Code memory
Static memory
Stack memory
Heap memory

Table 6.3 Runtime Memory

- Storage allocation techniques are of three types:
  i) Static allocation
  ii) Stack allocation
  iii) Heap allocation
- Symbol table is one of the most important data structures used by the compilers to know information about the identifiers and for error correction.
- Common techniques for the implementation of symbol table:
  i) Linear list
  ii) Ordered list
  iii) Trees
  iv) Hash table
- Hash table is one of the most important used data structure that is widely used by the compiler.