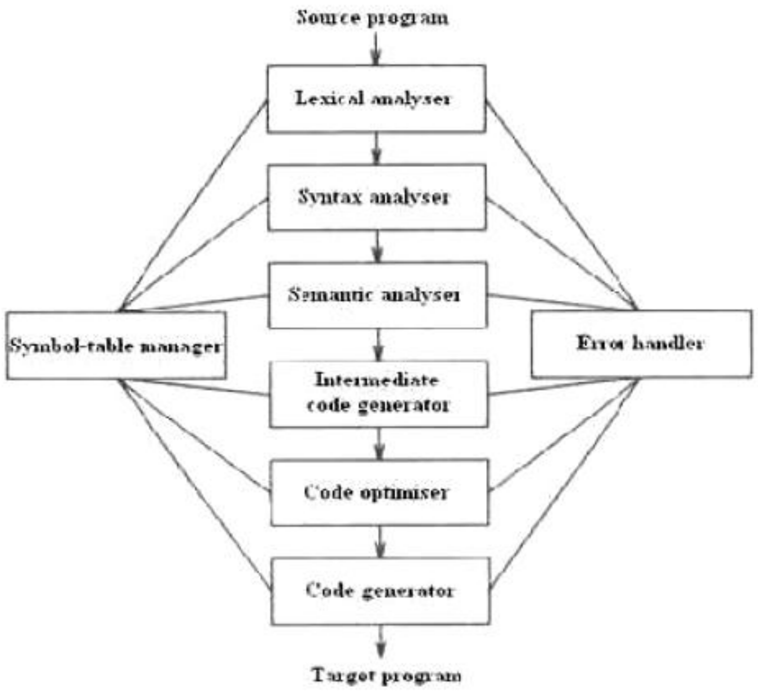
CS VII sem Compiler Construction

Solution

Q1 What are different phases of compiler? Explain them with help of suitable example.
Solu

Phases of Compiler



LEXICAL ANALYSIS:

- ✓ It is the first phase of the compiler. It gets input from the source program and produces tokens as output.
- ✓ It reads the characters one by one, starting from left to right and forms the tokens.
- ✓ **Token** : It represents a logically cohesive sequence of characters such as keywords,
- ✓ operators, identifiers, special symbols etc.

Example: a + b = 20
Here, a,b,+,=,20 are all separate tokens.
Group of characters forming a token is called the Lexeme.

- ✓ The lexical analyser not only generates a token but also enters the lexeme into the symbol table if it is not already there.

SYNTAX ANALYSIS:

- ✓ It is the second phase of the compiler. It is also known as parser.
- ✓ It gets the token stream as input from the lexical analyser of the compiler and generates syntax tree as the output.

Syntax tree:
It is a tree in which interior nodes are operators and exterior nodes are operands.

SEMANTIC ANALYSIS:

- ✓ It is the third phase of the compiler.
- ✓ It gets input from the syntax analysis as parse tree and checks whether the given syntax is correct or not.
- ✓ It performs type conversion of all the data types into real data types.

INTERMEDIATE CODE GENERATION:

- ✓ It is the fourth phase of the compiler.
- ✓ It gets input from the semantic analysis and converts the input into output as intermediate code such as three-address code.
- ✓ The three-address code consists of a sequence of instructions, each of which has atmost three operands.
  Example: t1=t2+t3

CODE OPTIMIZATION:

- ✓ It is the fifth phase of the compiler.
- ✓ It gets the intermediate code as input and produces optimized intermediate code as output.
- ✓ This phase reduces the redundant code and attempts to improve the intermediate code so that faster-running machine code will result.
- ✓ During the code optimization, the result of the program is not affected.
- ✓ To improve the code generation, the optimization involves
  - deduction and removal of dead code (unreachable code).
  - calculation of constants in expressions and terms.
  - collapsing of repeated expression into temporary string.
  - loop unrolling.
  - moving code outside the loop.
  - removal of unwanted temporary variables.

CODE GENERATION:

- ✓ It is the final phase of the compiler.
- ✓ It gets input from code optimization phase and produces the target code or object code as result.
- ✓ Intermediate instructions are translated into a sequence of machine instructions that perform the same task.
- ✓ The code generation involves
  - allocation of register and memory
  - generation of correct references
  - generation of correct data types
  - generation of missing code

SYMBOL TABLE MANAGEMENT:

- ✓ Symbol table is used to store all the information about identifiers used in the program.

- ✓ It is a data structure containing a record for each identifier, with fields for the attributes of the identifier.
- ✓ It allows to find the record for each identifier quickly and to store or retrieve data from that record.
- ✓ Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

ERROR HANDLING:

- ✓ Each phase can encounter errors. After detecting an error, a phase must handle the error so that compilation can proceed.
- ✓ In lexical analysis, errors occur in separation of tokens.
- ✓ In syntax analysis, errors occur during construction of syntax tree.
- ✓ In semantic analysis, errors occur when the compiler detects constructs with right syntactic structure but no meaning and during type conversion.
- ✓ In code optimization, errors occur when the result is affected by the optimization.
- ✓ In code generation, it shows error when code is missing etc.

OR

Q1 Define the terms tokens, pattern & lexemes with the help of examples.
Solu

TOKENS
A token is a string of characters, categorized according to the rules as a symbol (e.g., IDENTIFIER, NUMBER, COMMA). The process of forming tokens from an input stream of characters is called tokenization.

A token can look like anything that is useful for processing an input text stream or text file. Consider this expression in the C programming language: sum=3+2;

Lexeme	Token type
sum	Identifier
=	Assignment operator
3	Number
+	Addition operator
2	Number
;	End of statement

LEXEME:
Collection or group of characters forming tokens is called Lexeme.

PATTERN:
A pattern is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For

identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

Attributes for Tokens
Some tokens have attributes that can be passed back to the parser. The lexical analyzer collects information about tokens into their associated attributes. The attributes influence the translation of tokens.
i) Constant : value of the constant
ii) Identifiers: pointer to the corresponding symbol table entry.

Q2 Explain the following term

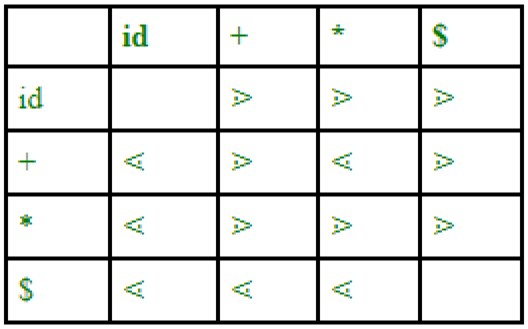
1. Operator Precedence parser for regular expression
2. Difference between bottom up and top down parsing with example
3. Context free grammer

Solu

Operator precedence parser –
An operator precedence parser is a one of the bottom-up parser that interprets an operator-precedence grammar. This parser is only used for operator grammars. *Ambiguous grammars are not allowed* in case of any parser except operator precedence parser.
There are two methods for determining what precedence relations should hold between a pair of terminals:

1. Use the conventional associativity and precedence of operator.
2. The second method of selecting operator-precedence relations is first to construct an unambiguous grammar for the language, a grammar that reflects the correct associativity and precedence in its parse trees.

This parser relies on the following three precedence relations: **⋖, ≐, ⋗**
a ⋖ b This means a “yields precedence to” b.
a ⋗ b This means a “takes precedence over” b.
a ≐ b This means a “has precedence as” b.



	id	+	*	$
id		⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	

Figure – Operator precedence relation table for grammar E->E+E/E*E/id
There is not given any relation between id and id as id will not be compared and two variables can not come side by side. There is also a disadvantage of this table as if we have n operators

than size of table will be n*n and complexity will be $0(n^2)$. In order to increase the size of table, use **operator function table**.
The operator precedence parsers usually do not store the precedence table with the relations; rather they are implemented in a special way. Operator precedence parsers use **precedence functions** that map terminal symbols to integers, and so the precedence relations between the symbols are implemented by numerical comparison. The parsing table can be encoded by two precedence functions **f** and **g** that map terminal symbols to integers. We select f and g such that:

1. $f(a) < g(b)$ whenever a is precedence to b
2. $f(a) = g(b)$ whenever a and b having precedence
3. $f(a) > g(b)$ whenever a takes precedence over b

Top-down vs Bottom-up Parsing

BASIS FOR COMPARISON	TOP-DOWN PARSING	BOTTOM-UP PARSING
Initiates from	Root	Leaves
Working	Production is used to derive and check the similarity in the string.	Starts from the token and then go to the start symbol.
Uses	Backtracking (sometimes)	Handling
Strength	Moderate	More powerful
Producing a parser	Simple	Hard
Type of derivation	Leftmost derivation	Rightmost derivation

Context-Free Grammars

A context-free grammar (CFG) is a set of recursive rewriting rules (or productions) used to generate patterns of strings.
A CFG consists of the following components:

- a set of *terminal symbols*, which are the characters of the alphabet that appear in the strings generated by the grammar.
- a set of *nonterminal symbols*, which are placeholders for patterns of terminal symbols that can be generated by the nonterminal symbols.
- a set of *productions*, which are rules for replacing (or rewriting) nonterminal symbols (on the left side of the production) in a string with other nonterminal or terminal symbols (on the right side of the production).
- a *start symbol*, which is a special nonterminal symbol that appears in the initial string generated by the grammar.

To generate a string of terminal symbols from a CFG, we:

- Begin with a string consisting of the start symbol;

- Apply one of the productions with the start symbol on the left hand size, replacing the start symbol with the right hand side of the production;
- Repeat the process of selecting nonterminal symbols in the string, and replacing them with the right hand side of some corresponding production, until all nonterminals have been replaced by terminal symbols.

Classification of Context Free Grammars

Context **F**ree **G**rammars (CFG) can be classified on the basis of following two properties:

1) Based on number of strings it generates.

- If CFG is generating finite number of strings, then CFG is **Non-Recursive** (or the grammar is said to be Non-recursive grammar)
- If CFG can generate infinite number of strings then the grammar is said to be **Recursive** grammar

During Compilation, the parser uses the grammar of the language to make a parse tree(or derivation tree) out of the source code. The grammar used must be unambiguous. An ambiguous grammar must not be used for parsing.

2) Based on number of derivation trees.

- If there is only 1 derivation tree then the CFG is unambiguous.
- If there are more than 1 derivation tree, then the CFG is ambiguous.

Example:

L= {wcw^R | w € (a, b)*}

Production rules:

1. S → aSa
2. S → bSb
3. S → c

Now check that abbcbba string can be derived from the given CFG.

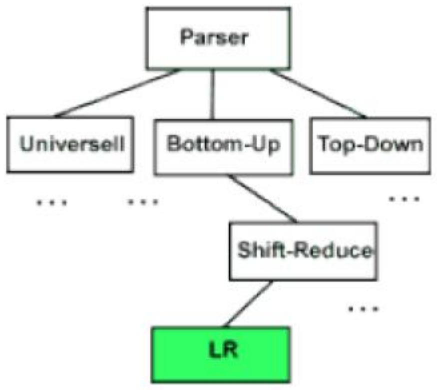
4. S ⇒ aSa
5. S ⇒ abSba
6. S ⇒ abbSbba
7. S ⇒ abbcbba

OR

Q2 What do you mean by LR parser ? What is the model of an LR parser ? Explain.
Solu

LR PARSING

The "L" is for left-to-right scanning of the input and the "R" is for constructing a rightmost derivation in reverse.

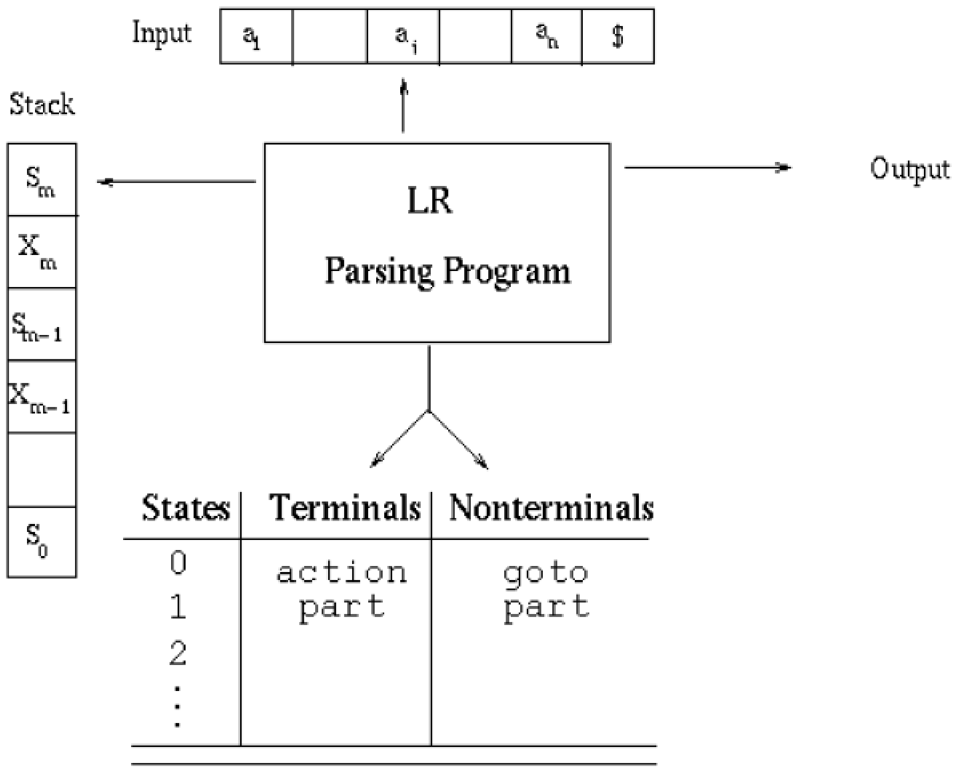


WHY LR PARSING:

- ✓ LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written.
- ✓ The LR parsing method is the most general non-backtracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.
- ✓ The class of grammars that can be parsed using LR methods is a proper subset of the class of grammars that can be parsed with predictive parsers.
- ✓ An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.

Model of LR Parser

The schematic form of an LR parser is shown below.



LR Parser Table

The program uses a stack to store a string of the form s0X1s1X2...Xmsm where sm is on top. Each Xi is a grammar symbol and each si is a symbol representing a state. Each state symbol summarizes the information contained in the stack below it. The combination of the state symbol on top of the stack and the current input symbol are used to index the parsing table and determine the shiftreduce parsing decision. The parsing table consists of two parts: a parsing action function action and a goto function goto. The program driving the LR parser behaves as follows: It determines sm the state currently on top of the stack and ai the current input symbol. It then consults action[sm,ai], which can have one of four values:

1. shift s, where s is a state
2. reduce by a grammar production A -> b
3. accept
4. error

The function goto takes a state and grammar symbol as arguments and produces a state. For a parsing table constructed for a grammar G, the goto table is the transition function of a deterministic finite automaton that recognizes the viable prefixes of G. Recall that the viable prefixes of G are those prefixes of right-sentential forms that can appear on the stack of a shiftreduce parser because they do not extend past the rightmost handle. A configuration of an LR parser is a pair whose first component is the stack contents and whose second component is the unexpended input: (s0 X1 s1 X2 s2... Xm sm, ai ai+1... an$)
This configuration represents the right-sentential form
X1 X1 ... Xm ai ai+1 ...an

in essentially the same way a shift-reduce parser would; only the presence of the states on the stack is new. Recall the sample parse we did (see Example 1: Sample bottom-up parse) in which we assembled the right-sentential form by concatenating the remainder of the input buffer to the top of the stack. The next move of the parser is determined by reading ai and sm, and consulting the parsing action table entry action[sm, ai]. Note that we are just looking at the state here and no symbol below it. We'll see how this actually works later. The configurations resulting after each of the four types of move are as follows:
If action[sm, ai] = shift s, the parser executes a shift move entering the configuration
(s0 X1 s1 X2 s2... Xm sm ai s, ai+1... an$)
Here the parser has shifted both the current input symbol ai and the next symbol. If action[sm, ai] = reduce A -> b, then the parser executes a reduce move, entering the configuration,
(s0 X1 s1 X2 s2... Xm-r sm-r A s, ai ai+1... an$)
where s = goto[sm-r, A] and r is the length of b, the right side of the production. The parser first popped 2r symbols off the stack (r state symbols and r grammar symbols), exposing state sm-r. The parser then pushed both A, the left side of the production, and s, the entry for goto[sm-r, A], onto the stack. The current input symbol is not changed in a reduce move. The output of an LR parser is generated after a reduce move by executing the semantic action associated with the reducing production. For example, we might just print out the production reduced.
If action[sm, ai] = accept, parsing is completed.