**Rajasthan Institute of Engineering & Technology, Jaipur  
Ist Midterm Examination, Sept-18  
Session: 2018-19  
Branch: - Computer Science and Engineering  
Semester:-B.Tech VII Semester  
Solution: Compiler Construction**

Q1(a). Explain all the phases of compiler with diagram

### Lexical Analysis

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

<token-name, attribute-value>

### Syntax Analysis

The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

### Semantic Analysis

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

### Intermediate Code Generation

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

### Code Optimization

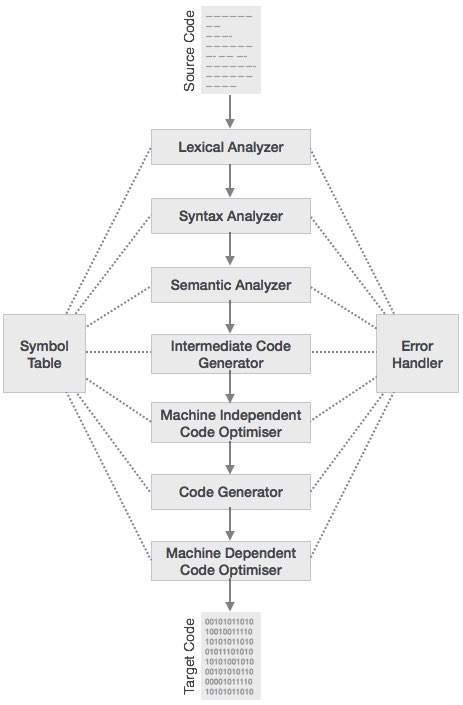
The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

### Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

### Symbol Table

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.



Q1(b) What is the difference between compiler and interpreter?

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

Q2. What is the role of lexical analyzer? How does lexical analyzer remove white spaces from a source file

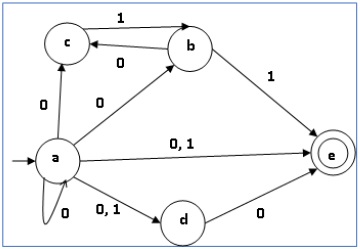
process of taking an input string of characters (such as the source code of a computer program) and producing a sequence of symbols called lexical tokens , or just tokens , which may be handled more easily by a parser.

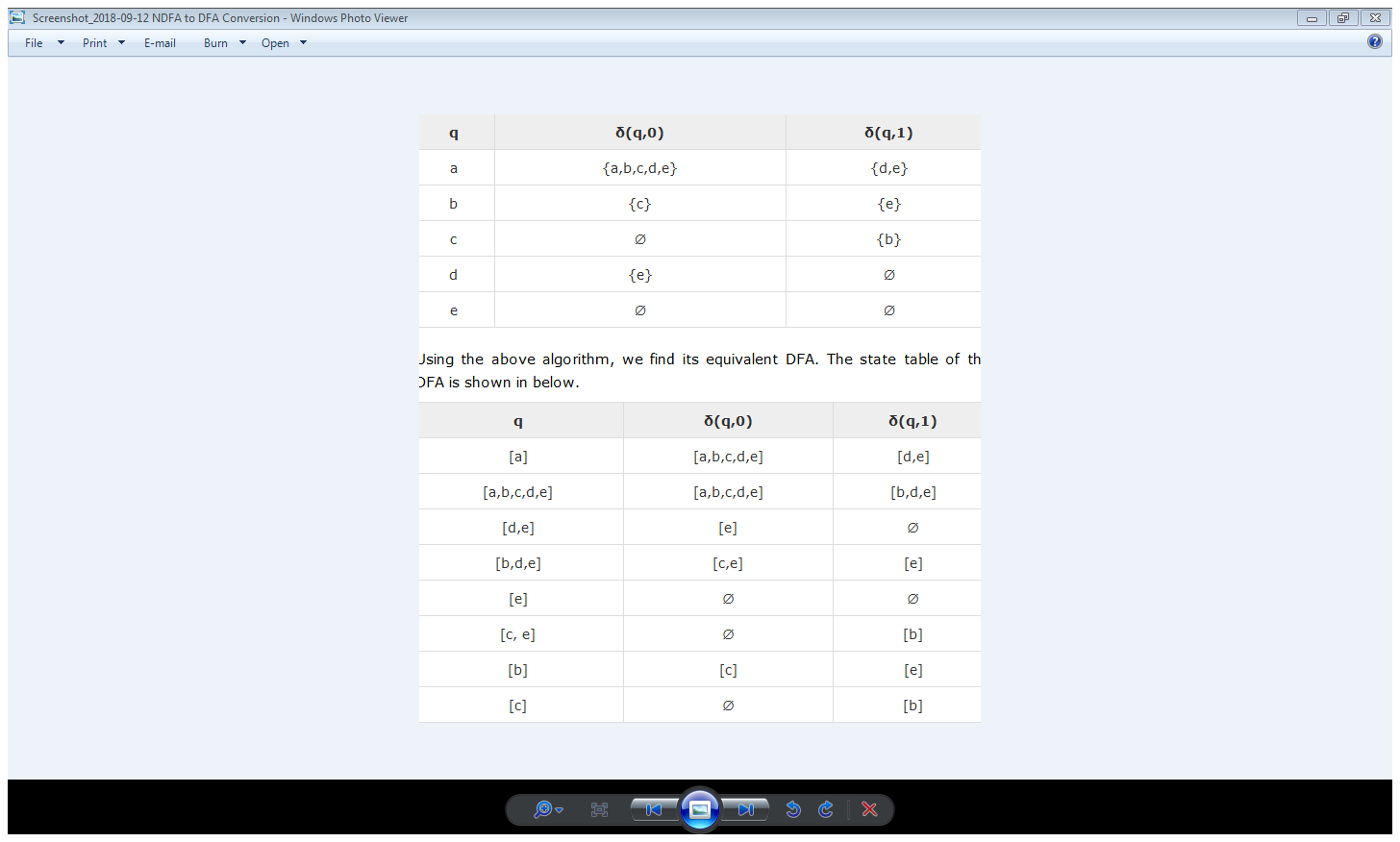
* The lexical analyzer reads the source text and, thus, it may perform certain secondary tasks:
* Eliminate comments and white spaces in the form of blanks, tab and newline characters.
* Correlate errors messages from the compiler with the sourceprogram (eg,keep track of the number of lines), ...
* The interaction with the parser is usually done by making thelexical analyzer bea sub-routine of the p

With the help of state diagram while appearing a whitespace or tab no action would be taken by the lexical analyzer that means state will not change.

Q3. How to convert NFA into DFA? Explain with example.

**Conversion from NFA to DFA**  
Suppose there is an NFA N < Q, ∑, q0, δ, F > which recognizes a language L. Then the DFA D < Q’, ∑, q0, δ’, F’ > can be constructed for language L as:  
Step 1: Initially Q’ = ɸ.  
Step 2: Add q0 to Q’.  
Step 3: For each state in Q’, find the possible set of states for each input symbol using transition function of NFA. If this set of states is not in Q’, add it to Q’.  
Step 4: Final state of DFA will be all states with contain F (final states of NFA)





Q4. Write short Notes on Following:

1. Difference between LL and LR parsing
2. Passes of compiler
3. Left factoring a grammar
4. Error Handler
5. **Difference between LL and LR parsing**

At a high level, the difference between LL parsing and LR parsing is that LL parsers begin at the start symbol and try to apply productions to arrive at the target string, whereas LR parsers begin at the target string and try to arrive back at the start symbol.

An LL parse is a left-to-right, leftmost derivation. That is, we consider the input symbols from the left to the right and attempt to construct a leftmost derivation. This is done by beginning at the start symbol and repeatedly expanding out the leftmost nonterminal until we arrive at the target string. An LR parse is a left-to-right, rightmost derivation, meaning that we scan from the left to right and attempt to construct a rightmost derivation. The parser continuously picks a substring of the input and attempts to reverse it back to a nonterminal.

During an LL parse, the parser continuously chooses between two actions:

1. **Predict**: Based on the leftmost nonterminal and some number of lookahead tokens, choose which production ought to be applied to get closer to the input string.
2. **Match**: Match the leftmost guessed terminal symbol with the leftmost unconsumed symbol of input.
3. **Passes of compiler**
4. A one-pass compiler is a compiler that passes through the source code of each compilation unit only once. A multi-pass compiler is a type of compiler that processes the source code or abstract syntax tree of a program several times.
5. A one-pass compilers is faster than multi-pass compilers
6. A one-pass compiler has limited scope of passes but multi-pass compiler has wide scope of passes.
7. Multi-pass compilers are sometimes called wide compilers where as one-pass compiler are sometimes called narrow compiler.
8. Many programming languages cannot be represented with a single pass compilers, for example Pascal can be implemented with a single pass compiler where as languages like **Java** require a multi-pass compiler.
9. **Left factoring a grammar**

Left factoring is removing the common left factor that appears in two productions of the same non-terminal. It is done to avoid back-tracing by the parser. Suppose the parser has a look-ahead ,consider this example-

A -> qB | qC

where A,B,C are non-terminals and q is a sentence. In this case, the parser will be confused as to which of the two productions to choose and it might have to back-trace. After left factoring, the grammar is converted to-

A -> qD

D -> B | C

In this case, a parser with a look-ahead will always choose the right production.

Left recursion is a case when the left-most non-terminal in a production of a non-terminal is the non-terminal itself( direct left recursion ) or through some other non-terminal definitions, rewrites to the non-terminal again(indirect left recursion). Consider these examples -

(1) A -> Aq (direct)

(2) A -> Bq B -> Ar (indirect)

Left recursion has to be removed if the parser performs top-down parsing

**d. Error Handler**

The tasks of the **Error Handling** process are to detect each error, report it to the user, and then make some recover strategy and implement them to handle error. During this whole process processing time of program should not be slow. An **Error** is the blank entries in the symbol table.

**Types or Sources of Error –** There are two types of error: run-time and compile-time error:

1. A **run-time error** is an error which takes place during the execution of a program, and usually happens because of adverse system parameters or invalid input data. The lack of sufficient memory to run an application or a memory conflict with another program and logical error are example of this. Logic errors, occur when executed code does not produce the expected result. Logic errors are best handled by meticulous program debugging.
2. **Compile-time errors** rises at compile time, before execution of the program. Syntax error or missing file reference that prevents the program from successfully compiling is the example of this.

**Classification of Compile-time error –**

1. **Lexical** : This includes misspellings of identifiers, keywords or operators
2. **Syntactical** : missing semicolon or unbalanced parenthesis
3. **Semantical** : incompatible value assignment or type mismatches between operator and operand
4. **Logical** : code not reachable, infinite loop.

**Finding error or reporting an error –** Viable-prefix is the property of a parser which allows early detection of syntax errors.

* **Goal:** detection of an error as soon as possible without further consuming unnecessary input
* **How:** detect an error as soon as the prefix of the input does not match a prefix of any string in the  
  language.
* **Example:** for(**;**), this will report an error as for have two semicolons inside braces.

**Error Recovery –**  
The basic requirement for the compiler is to simply stop and issue a message, and cease compilation. There are some common recovery methods that are follows.

1. **Panic mode recovery:** This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops while recovering error. The parser discards the input symbol one at a time until one of the designated (like end, semicolon) set of synchronizing tokens (are typically the statement or expression terminators) is found. This is adequate when the presence of multiple errors in same statement is rare. Example: Consider the erroneous expression- (1 + + 2) + 3. Panic-mode recovery: Skip ahead to next integer and then continue. Bison: use the special terminal **error** to describe how much input to skip.

E->int|E+E|(E)|error int|(error)

1. **Phase level recovery:** Perform local correction on the input to repair the error. But error correction is difficult in this strategy.
2. **Error productions:** Some common errors are known to the compiler designers that may occur in the code. Augmented grammars can also be used, as productions that generate erroneous constructs when these errors are encountered. Example: write 5x instead of 5\*x
3. **Global correction:** Its aim is to make as few changes as possible while converting an incorrect input string to a valid string. This strategy is costly to implement.

Q5. Write short Notes on Following:

1. **Left recursion and elimination :**

**Left Recursion:** A grammar is left recursive if it has a nonterminal A such that there is a derivation **A -> Aα | β** where α and β are sequences of terminals and nonterminals that do not start with A.

While designing a top down-parser, if the left recursion exist in the grammar then the parser falls in an infinite loop, here because A is trying to match A itself, which is not possible. We can eliminate the above left recursion by rewriting the offending production. As-

**A -> βA'**

**A' -> αA' | epsilon**

**Left Factoring:** Left factoring is required to eliminate non-determinism of a grammar. Suppose a grammar, **S -> abS | aSb**

Here, S is deriving the same terminal a in the production rule(two alternative choices for S), which follows non-determinism. We can rewrite the production to defer the decision of S as-

**S -> aS'**

**S' -> bS | Sb**

Thus, S' can be replaced for bS or Sb

1. **CFG**  : A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple **(N, T, P, S)** where

* **N** is a set of non-terminal symbols.
* **T** is a set of terminals where **N ∩ T = NULL.**
* **P** is a set of rules, **P: N → (N ∪ T)\***, i.e., the left-hand side of the production rule **P** does have any right context or left context.
* **S** is the start symbol.

**Example**

* The grammar ({A}, {a, b, c}, P, A), P : A → aA, A → abc.
* The grammar ({S, a, b}, {a, b}, P, S), P: S → aSa, S → bSb, S → ε
* The grammar ({S, F}, {0, 1}, P, S), P: S → 00S | 11F, F → 00F | ε

## Generation of Derivation Tree

A derivation tree or parse tree is an ordered rooted tree that graphically represents the semantic information a string derived from a context-free grammar.

### Representation Technique

* **Root vertex** − Must be labeled by the start symbol.
* **Vertex** − Labeled by a non-terminal symbol.
* **Leaves** − Labeled by a terminal symbol or ε.

If S → x1x2 …… xn is a production rule in a CFG, then the parse tree / derivation tree will be as follows

1. **Input Buffering :**

To ensure that a right lexeme is found, one or more characters have to be looked up beyond the next lexeme.

• Hence a two-buffer scheme is introduced to handle large lookaheads safely.

• Techniques for speeding up the process of lexical analyzer such as the use of sentinels to mark the buffer end have been adopted.

There are three general approaches for the implementation of a lexical analyzer:

(i) By using a lexical-analyzer generator, such as lex compiler to produce the lexical analyzer from a regular expression based specification. In this, the generator provides routines for reading and buffering the input.

(ii) By writing the lexical analyzer in a conventional systems-programming language, using I/O facilities of that language to read the input.

(iii) By writing the lexical analyzer in assembly language and explicitly managing the reading of input.

Buffer Pairs

Because of large amount of time consumption in moving characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.

• Consists of two buffers, each consists of N-character size which are reloaded alternatively.

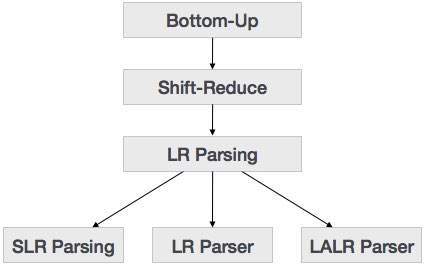
• N-Number of characters on one disk block, e.g., 4096.

• N characters are read from the input file to the buffer using one system read command.

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

1. **Bottom up parsing :**

Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. Here, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol. The image given below depicts the bottom-up parsers available.



**Shift-Reduce Parsing**

Shift-reduce parsing uses two unique steps for bottom-up parsing. These steps are known as shift-step and reduce-step.

* **Shift step**: The shift step refers to the advancement of the input pointer to the next input symbol, which is called the shifted symbol. This symbol is pushed onto the stack. The shifted symbol is treated as a single node of the parse tree.
* **Reduce step** : When the parser finds a complete grammar rule (RHS) and replaces it to (LHS), it is known as reduce-step. This occurs when the top of the stack contains a handle. To reduce, a POP function is performed on the stack which pops off the handle and replaces it with LHS non-terminal symbol.

Q6 . Consider the following grammar :

**E → TE’**

**E ‘→ +TE’ | ϵ**

**T → FT’**

**T’→ \*FT’ | ϵ**

**F → (E) | id**

Where ϵ denoted the empty string of symbol

* 1. Compute FIRST and FOLLOW for each non- terminal of the grammar G.
  2. Construct a predictive parsing table for a grammar G.

**FIRST and FOLLOW sets of the modified grammar**

Start with FIRST sets. The computation completes in these 3 phases:

Initially we have   
these sets:

FIRST(E) = {}

FIRST(E′) = {ε}

FIRST(T) = {}

FIRST(T′) = {ε}

FIRST(F) = {}

 Use the right sides   
of E', T', F to get:

FIRST(E) = {}

FIRST(E′) = {+,ε}

FIRST(T) = {}

FIRST(T′) = {\*,ε}

FIRST(F) = {(,id}

 Use the right sides   
of T, then E to get:

FIRST(E) = {(,id}

FIRST(E′) = {+,ε}

FIRST(T) = {(,id}

FIRST(T′) = {\*,ε}

FIRST(F) = {(,id}

 Next compute FOLLOW sets. Start here:

FOLLOW(E) = {$}

FOLLOW(E′) = {}

FOLLOW(T) = {}

FOLLOW(T′) = {}

FOLLOW(F) = {}

Then consider effects of these rules:   
   E → TE′: FOLLOW(E) ⊆ FOLLOW(E′)   
   E' → +TE′: FIRST(E′) - {ε} = {+} ⊆ FOLLOW(T)   
   T → FT′: FOLLOW(T) ⊆ FOLLOW(T′)

FOLLOW(E) = {$}

FOLLOW(E′) = {$}

FOLLOW(T) = {+}

FOLLOW(T′) = {+}

FOLLOW(F) = {}

Continuing,   
   F → (E): FIRST(')') ⊆ FOLLOW(E)   
   T′ → \*FT′: FIRST(T′) - {ε} = {\*} ⊆ FOLLOW(F)

FOLLOW(E) = {$,)}

FOLLOW(E′) = {$}

FOLLOW(T) = {+}

FOLLOW(T′) = {+}

FOLLOW(F) = {\*}

Continuing,   
   E → TE′: FOLLOW(E) ⊆ FOLLOW(E′).   
   E′ → +TE′: FOLLOW(E′) ⊆ FOLLOW(T) since ε ∈ FIRST(E′)

FOLLOW(E) = {$,)}

FOLLOW(E') = {$,)}

FOLLOW(T) = {+,$,)}

FOLLOW(T') = {+}

FOLLOW(F) = {\*}

Continuing,   
   T → FT': FOLLOW(T) ⊆ FOLLOW(T′):   
   T' → \*FT': FOLLOW(T') ⊆ FOLLOW(F), since ε ∈ FIRST(T')

FOLLOW(E) = {$,)}

FOLLOW(E') = {$,)}

FOLLOW(T) = {+,$,)}

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

FOLLOW(F) = {\*,+,$,)}

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | + | \* | Id | ( | ) | $ |
| E |  |  | E->TE’ | E->TE’ |  |  |
| E’ | E’->+TE’ |  |  |  | E’-> ε | E’-> ε |
| T |  |  | T->FT’ | T->FT’ |  |  |
| T’ | T’-> ε | T’->\*FT’ |  |  | T’-> ε | T’-> ε |
| F |  |  | F-> id | F->( E ) |  |  |

Q7. Consider the following grammer:

**E->E\*E | E-E| E/E| E+E | (E) | -E | id**

Input string id\*(id+id)-id/id

What is operator precedence parsing ? explain with the help of above example.

Ans : As a general parsing technique, operator-precedence parsing has a number of disadvantages: 1) it is hard to handle tokens like the minus sign (-), which has two different precedence (depending on whether it is unary or binary); 2) the relationship between a grammar for the language being parsed and the operator-precedence pararser itself is tenuous, one cannot always be sure the parser accepts exactly the desired language. 3) only a small class of grammars can be parsed using operator-precedence techniques

In operator-precedence parsing, we define three disjoint precednese relations: , between certain pairs of terminals. These precedence relations guide the selection of handles and have the following meanings:

* a<·b - a yields precedence to b;
* a=b – a has the same precedence as b;
* a·>b – a takes precedence over b.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Id | \* | + | - | / | ( | ) | $ |
| Id | - | > | > | > | > |  | > | > |
| \* | < | > | > | > | > | < | > | > |
| + | < | < | > | > | < | < | > | > |
| - | < | < | > | > | < | < | > | > |
| / | < | > | > | > | > | < | > | > |
| ( | < | < | < | > | < | < | = |  |
| ) |  | > | > | > | > |  | > | > |
| $ | < | < | < | < | < | < |  |  |

Q 8. What is LR Parsing? Explain algorithm of LR parser.

LR parsers are used to parse the large class of context free grammars. This technique is called LR(k) parsing.

• L is left-to-right scanning of the input.

• R is for constructing a right most derivation in reverse.

• k is the number of input symbols of lookahead that are used in making parsing decisions.

There are three widely used algorithms available for constructing an LR parser:

• SLR(l) - Simple LR

o Works on smallest class of grammar.

o Few number of states, hence very small table.

o Simple and fast construction.

• LR parsers can handle a large class of context-free grammars.

• The LR parsing method is a most general non-back tracking shift-reduce parsing method.

• An LR parser can detect the syntax errors as soon as they can occur.

• LR grammars can describe more languages than LL grammars.

Model of LR Parser

LR parser consists of an input, an output, a stack, a driver program and a parsing table that has two functions

1. Action

2. Goto

The driver program is same for all LR parsers. Only the parsing table changes from one parser to another.

The parsing program reads character from an input buffer one at a time, where a shift reduces parser would shift a symbol; an LR parser shifts a state. Each state summarizes the information contained in the stack.

The stack holds a sequence of states, so, s1, · ·· , Sm, where Sm is on the top.

Action This function takes as arguments a state i and a terminal a (or $, the input end marker). The value of ACTION [i, a] can have one of the four forms:

i) Shift j, where j is a state.

ii) Reduce by a grammar production A---> β.

iii) Accept.

iv) Error.

Goto This function takes a state and grammar symbol as arguments and produces a state.

If GOTO [Ii ,A] = Ij, the GOTO also maps a state i and non terminal A to state j.

**LR Parsing Algorithm :**

1. If ACTION[sm, ai] = shift s. The parser executes the shift move, it shifts the next state s onto the stack, entering the configuration

a) Sm - the state on top of the stack.

b) ai- the current input symbol.

2. If ACTION[sm, ai] =reduce A---> β, then the parser executes a reduce move, entering the configuration

(s0s1 ... S(m-r)S, ai+l ... an$)

a) where r is the length of β and s= GOTO[sm - r, A].

b) First popped r state symbols off the stack, exposing state Sm-r·

c) Then pushed s, the entry for GOTO[sm-r, A], onto the stack.

3. If ACTION[sm, ai] = accept, parsing is completed.

4. If ACTION[sm, ai] = error, the parser has discovered an error and calls an error recovery routine.

