Compiler Construction
Assignment - I

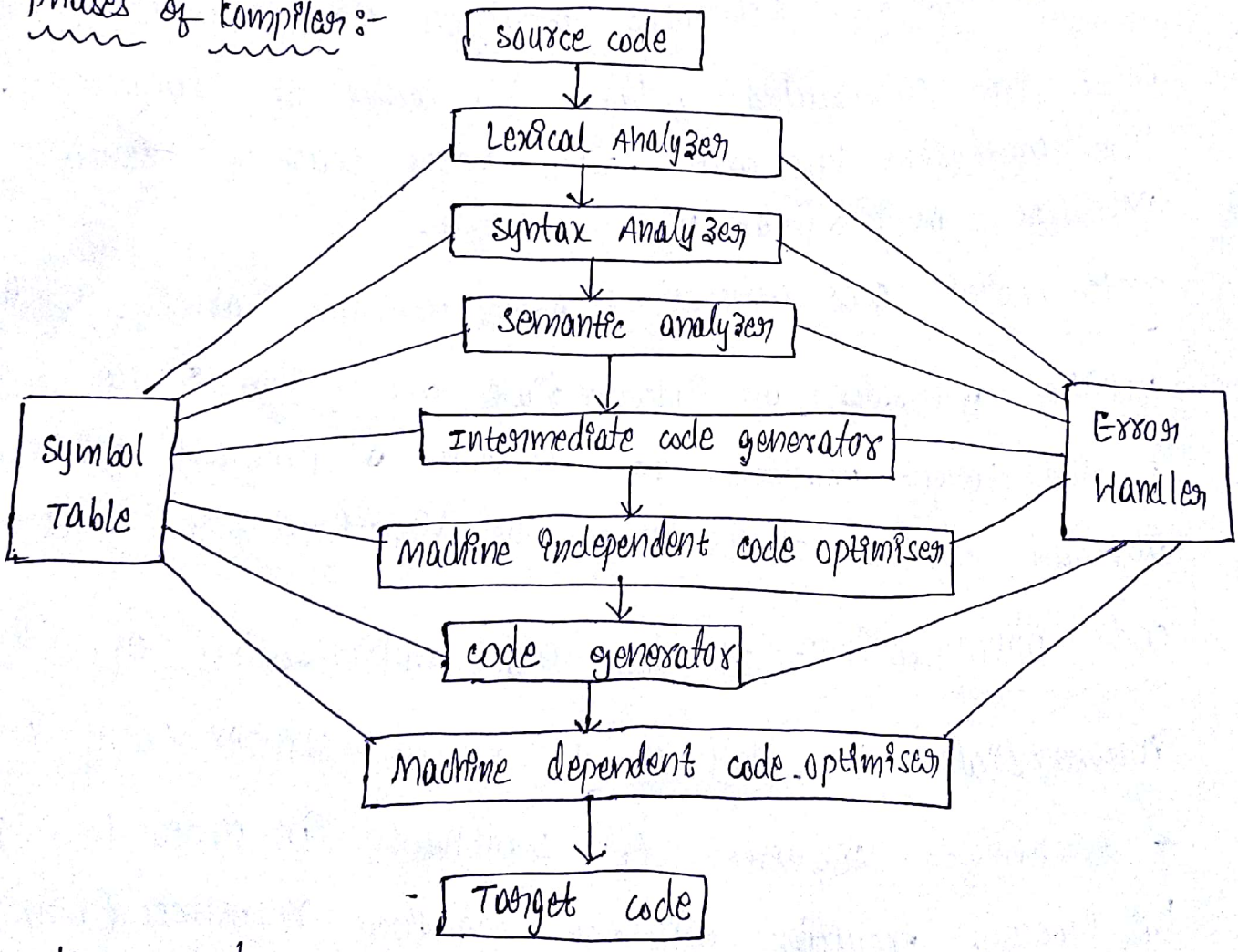
S. Rajeshwari
1602-16-737-092
IT-B

1) Draw a block diagram of each phases of a compiler & explain main functions of each phase of the compiler. Show the output generated by each phase for the following expression.

$$x = a[i]+2$$

Ans:- The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler.

phases of compiler:-




Functions of phases:-

Lexical Analyser:- The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters & converts it into meaningful lexemes.

Lexical analyzer represents these lexemes in the form of tokens as: <token-name, attribute-value>

Syntax Analysis:- The syntax analysis also called parsing. It takes the token produced by lexical analysis as input and generates a parse tree. Here token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by tokens is syntactically correct.

Semantic Analysis:- semantic analyser checks whether the parse tree constructed follows the rules of language. type conversions, type casting. also keeps track of identifiers, produces an annotated syntax tree as output.

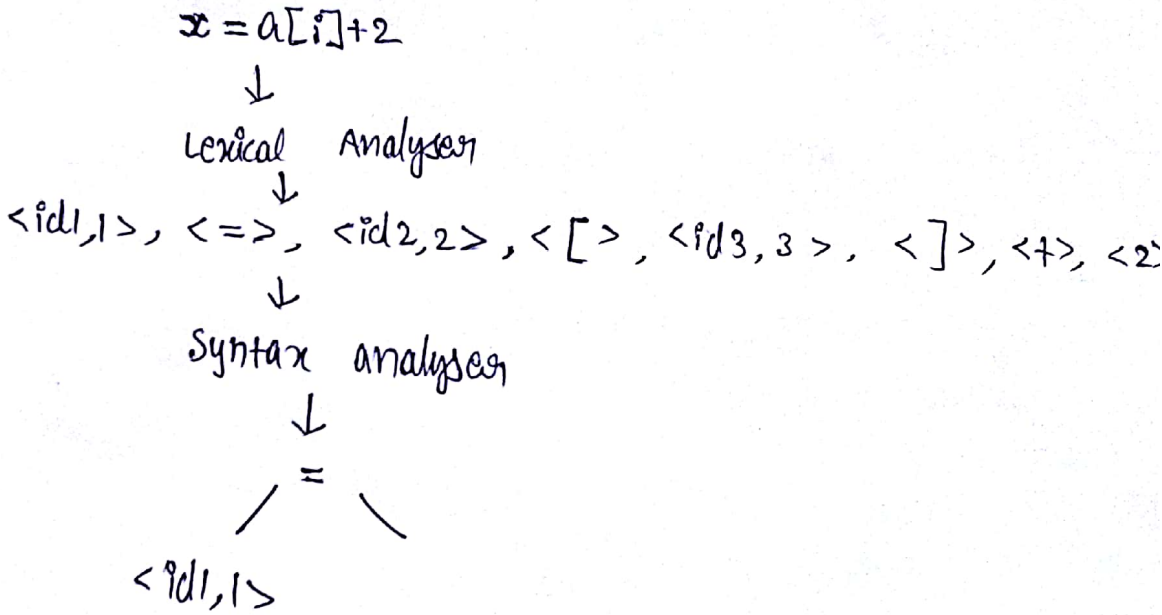
Intermediate code generation:- After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in b/w the high level & low level language.

code optimization:- It does code optimization of the intermediate code. that is it removes unnecessary code lines, & arranges sequence of statements in order to speed up the program execution without wasting resources (CPU).

code generation:- In this phase, the code generator takes the optimized representation of intermediate code & maps it to the target machine language.

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

Error Handler :- It is used to handle errors at every phase. Anything is not according to expected it will suggest saying it is error.



2) Discuss about LEX tool. Write LEX ~~app~~ specification for replacing a string with another string in the given input file and count the no. of replacements.
(ex:- vce should be replaced by vasavi college of engineering).

Ans LEX tool:- the input notation for the lex tool is referred to as the Lex language and the tool itself is the Lex compiler. Behind the scenes, the lex compiler transforms the input patterns into a transition diagram and generates code in a file called lex.yy.c, that simulates this transition diagram. this tools is designed for lexical analysers.

→ we have to create lex specification file (any file with .l extension), once .l file given to lexical analyser, it creates c program as lex.yy.c.

Structure of lex program:-

```
declarations
%%
 translation rules
 %%
auxillary functions.
```

Definition section:- entire code between %{ and %} copied into resulting c file. It contains initialisation, declaration, comments etc., lex specification file exists without this section also.
%{ declarations %}
Rule section:- Here we use writing the regular expressions & the subsequent actions are placed as follows

%%

{pattern} {action}

%%

subroutine section:- It holds additional functions that are used in the actions. These functions can be compiled separately & loaded, with the lexical analyser. this also contains main program, if we want to compile the lex output as standalone program.

```
void main()
{
    yylex();
}
```

lex specification file for replacing one string with another:-

```
%{
  #include <stdio.h>
%}
%%
"ENGG" { printf("ENGINEERING"); }
"RAJI" { printf("RAJESHWARI"); }
"VCE " { printf("VASAVI COLLEGE OF ENGINEERING"); }
%%
main()
{
    yylex();
    return 0;
    yywrap();
    return 0;
}
```

output:- lex Replace.l

./a.out

VCE

VASAVI COLLEGE OF ENGINEERI

3) Define Left Recursion & Left Factoring grammar. Discuss about the reasons for eliminating the left recursion & left factoring. Write the rules for eliminating left recursion & left factoring. Apply these rules for the following CFG.

a) $X \rightarrow Xsb / sa / b$

$S \rightarrow Sb / Xa / a$

b) $S \rightarrow a / ab / abc / abcd$

ns:- **Left Recursion:-** the production is left-recursive if the leftmost symbol on the right side is the same as the non-terminal on the left side

Ex:- $E \rightarrow E+T$

**Reasons for eliminating left recursion:-**

→ If one were to code this production in a recursive-descent parser, the parser would go in an infinite loop.

**Rules for eliminating left recursion:-**

→ we can eliminate the left recursion by introducing new non-terminals and new production rules.

→ If grammar is of form $A \rightarrow A\alpha / \beta$, introduce $A'$ such that

$\Rightarrow A \rightarrow \beta A'$

$\Rightarrow A' \rightarrow \alpha A' / \varepsilon$

Left factoring :- the production is left factoring if there is a common left factor (terminal / terminals) that appears in all productions of same non-terminal.

ex:- $A \rightarrow aB | aC$

Reasons for eliminating left factoring:-

→ this grammar creates problematic situation for top down parsers.

→ top down parsers can not decide which production must be chosen to parse the string in hand. It creates confusion.

Rules for eliminating left factoring:-

→ we can eliminate the left factoring by introducing new non terminal for each production of left factoring.

→ If the grammar is of the form $A \rightarrow \alpha\beta_1 | \alpha\beta_2$ ; introduce non-terminal $A'$ such that $\Rightarrow A \rightarrow \alpha A'$

$\Rightarrow A' \rightarrow \beta_1 | \beta_2$

a) $X \rightarrow XSb | Sa | b$

$S \rightarrow Sb | Xa | a$

eliminating left recursion

$X \rightarrow SaX' | bX'$

$X' \rightarrow SbX' | \varepsilon$

$S \rightarrow XaS' | aS'$

$S' \rightarrow bS' | \varepsilon$

No left factoring.

b) $S \rightarrow a|ab|abc|abcd$

eliminating left factoring

$S \rightarrow aS'$

$S' \rightarrow \varepsilon|b|bc|bcd$

Again left factoring

$\Rightarrow S' \rightarrow bA/\varepsilon$

$A \rightarrow c|cd|\varepsilon$

Again left factoring

$A \rightarrow cB/\varepsilon$

$B \rightarrow d|\varepsilon$

∴ After left factoring

$S \rightarrow aS'$

~~$S' \rightarrow b|bc|bcd|\varepsilon$~~

$S' \rightarrow bA|\varepsilon$

$A \rightarrow cB|\varepsilon$

$B \rightarrow d|\varepsilon$

4) write an algorithm for the Recursive Descent parsing. construct Recursive Descent parser for the following grammar & Input string is "read"

$S \rightarrow rXd$

$X \rightarrow oa|ea$

Ans Algorithm for Recursive Descent parsing:-

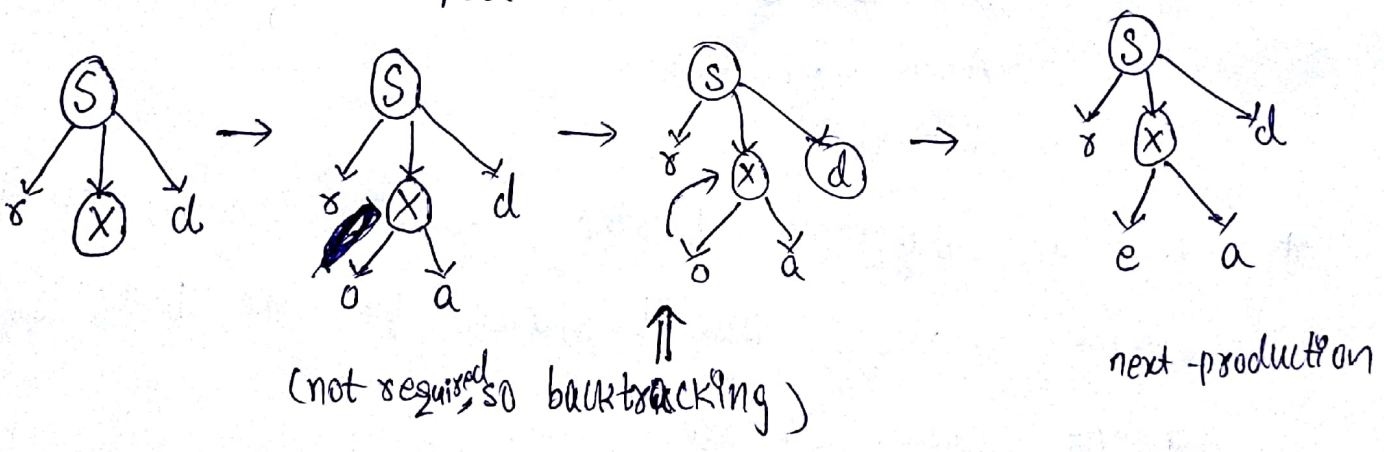
```
void A
{
    choose an A production, A → X1X2X3 ---- Xk
```

```
for (i=1 to k)
{
    if (Xi is a nonterminal)
        call procedure Xi();
    else if (Xi equals the current i/p symbol a)
        advance input to the next symbol;
    else
        error;
}
}
```

Recursive descent parser (backtracking):- input :- read

given $S \rightarrow \gamma X d$

$X \rightarrow oa / ea$




(not required, so backtracking)

next-production

5) Write the rules to compute FIRST and FOLLOW. Compute the FIRST and FOLLOW sets for the following grammar.

Ans:- Rules to compute FIRST:-

FIRST(X) for all grammar symbols X

1) If X is terminal, FIRST(X) = {X}.

2) If $X \to \varepsilon$ is a production, then add $\varepsilon$ to FIRST(X).

3) If X is a non-terminal, and $X \to Y_1 Y_2 \cdots Y_k$ is a production, and $\varepsilon$ is in all of $FIRST(Y_1), \cdots FIRST(Y_k)$, then add $\varepsilon$ to FIRST(X).

4) If X is a non-terminal, and $X \to Y_1 Y_2 \cdots Y_k$ is a production, then add a to FIRST(X) if for some i, a is in $FIRST(Y_i)$, and $\varepsilon$ is in all of $FIRST(Y_1), \ldots, FIRST(Y_{i-1})$.

Rules for writing FOLLOW:-

For all non-terminals a

1) If \$ is the input end-marker, and S is the start symbol,

$\$ \in FOLLOW(S)$.

2) If there is a production, $A \to \alpha B \beta$, then $(FIRST(\beta) - \varepsilon) \subseteq FOLLOW(B)$

3) If there is a production, $A \to \alpha B$, or a production $A \to \alpha B \beta$, where $\varepsilon \in FIRST(\beta)$, then $FOLLOW(A) \subseteq FOLLOW(B)$.

given productions

exp → exp addop term | term // left recursive

addop → + | -

term → term mulop factor | factor // left recursive

mulop → *

factor → (exp) | number

eliminating left recursion

First (exp) =

exp → term exp'

exp' → addop term exp' | ε

term → factor term'

term' → mulop factor term' / ε

so After elimination of Ambiguity, grammar is

exp → term exp'

exp' → addop term exp' / ε

addop → + / −

term → factor term'

term' → mulop factor term' / ε

mulop → *

factor → (exp) / number

First(exp) = first(term) = first(factor) = {(, number}

First(exp') = first(addop), ε = {+, −, ε}

First(addop) = {+, −}

First(term) = first(factor) = {(, number}

First(term') = first(mulop), ε = {*, ε}

First(factor) = {(, number}

First(mulop) = {*}

Follow(exp) = {$, )}

Follow(exp') = {$, )} = follow(exp)

Follow(addop) = first(term) = {(, number}

Follow(term) = first(exp') & follow(exp') ∵ ε = {+, −, $, ), $}

Follow(term') = follow(term) = {+, −, $, ), $}

Follow(mulop) = first(factor) = {(, number}

Follow(factor) = first(term') = {*, $} & follow of (term') ∵ ε

= {*, +, −, ), $}

6) What is predictive parsing? explain the model of predictive parser. write the algorithm to construct predictive parser. Construct LL(1) parsing table for the following grammar.

exp → exp addop term/term

addop → +/−

term → term mulop factor/factor

mulop → *

factor → (exp)/number

Ans:- predictive parsing:- It is constructing a top-down parser that never backtracks. To do so, we must remove ambiguity in grammar

→ eliminate left recursion &

→ perform left factoring.

i) Find First & Follow for given grammar

ii) construct predictive parsing table

iii) parsing Action

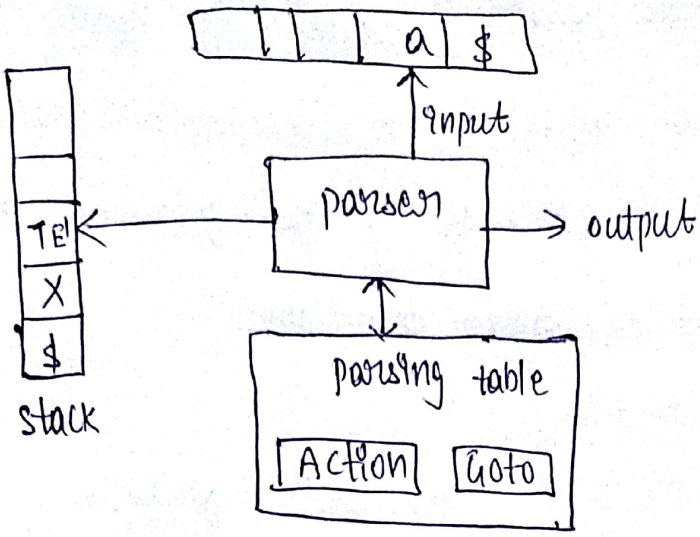
model of predictive parser:- It is a recursive descent parser, which has capability to predict which production is to be used to replace the input string. No backtracking exists.

→ It uses look ahead pointer, which points to next input symbols.

→ It uses LL1 grammar → Look ahead symbol

↓ ↓

left to right — Left most derivation




Algorithm to construct predictive parser:-

Input: Grammar G

output: parsing table M

Method: For each production $A \to \alpha$ of the grammar do the following.

1) For each terminal 'a' in First($\alpha$), add $A \to \alpha$ to M[A, a]

2) If $\varepsilon$ is in FIRST($\alpha$), then for each terminal 'b' in FOLLOW(A), add $A \to \alpha$ to M[A, b]. If $\varepsilon$ is in FIRST($\alpha$), $ is in FOLLOW(A), add $A \to \alpha$ to M[A, $] as well.

given grammar is :-

exp → exp addop term / term

addop → +/−

term → term mulop factor / factor

mulop → *

factor → (exp) / number

ambiguity exists

∴ exp → term exp'
exp' → addop term exp' | ε
addop → + | −
term → factor term'
term' → mulop factor term' | ε
mulop → *
factor → (exp) | number

| | First | Follow |
|---|---|---|
| exp | { (, number } | { $, ) } |
| exp' | { +, −, ε } | { $, ) } |
| addop | { +, − } | { (, number } |
| term | { (, number } | { +, −, ), $ } |
| term' | { *, ε } | { +, −, ), $ } |
| mulop | { * } | { (, number } |
| factor | { (, number } | { *, +, −, ), $ } |

predictive parsing table:-

e :- error

| | I/p symbols | | | | | | |
|---|---|---|---|---|---|---|---|
| | ( | ) | + | − | * | number | $ |
| exp | exp → term exp' | e | e | e | e | exp → term exp' | e |
| exp' | e | exp' → ε | exp' → addop term exp' | exp' → addop term exp' | e | e | exp' → ε |
| addop | e | e | addop → + | addop → − | e | e | e |
| term | term → factor term' | e | e | e | e | term → factor term' | e |

| | ( | ) | + | − | * | number | $ |
|---|---|---|---|---|---|---|---|
| term' | | term'→ ε | term'→ ε | term'→ ε | term'→ mulop factor term' | | term' → ε |
| mulop | | | | | mulop→ * | | |
| factor | factor→ (exp) | | | | | factor→ number | |

7) consider the following grammar:-

$S \rightarrow AS/b$

$A \rightarrow SA/a$

parse the input string "abab" using shift-reduce parser;

Ans:- shift Reduce parser:- It has shift, reduce, accept, error.

| Stack | Input Buffer | Action |
|---|---|---|
| $ | abab $ | shift |
| $a | bab $ | Reduce |
| $A | bab$ | shift |
| $Ab | ab $ | Reduce [∵ S→Ab |
| $S | ab $ | shift |
| $Sa | b $ | Reduce [∵ A→Sa] |
| $A | b $ | shift |
| $Ab | $ | Reduce [∵ S→Ab] |
| $S | $ | Accept |

8) Explain error recovery techniques in LR parsers.

Ans:- An LR parser will detect an error when it consults the parsing action table & find a blank or error entry. Errors are never detected by consulting goto table. An LR parser will detect as soon as there is no valid continuation for input scanned.

panic-mode error Recovery:-

We can implement panic-mode error recovery by scanning down the stack s with a goto on a particular nonterminals A & found zero or more input symbols are then discarded until a symbol a is found that can legitimately follow A. The parser then stacks the state GOTO(S,A). and resumes normal parsing. situation might exist where there is more than one choice for nonterminal A.

phrase-level Recovery:-

It is implemented by examining each error entry in LR action table & deciding on basis of language usage the most likely programmer error that would give rise to that error. An appropriate recovery procedure can then be constructed. presumbly the top of the stack and/or first input symbol would be modified in a way deemed appropriate for each error entry.