Rohan Nyati
500075940
R177219148
Batch-5 (Ai & Ml)

Class Test -1

Q.1

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. Let us understand the phases of a compiler.

Lexical Analysis

The first phase of the scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes.

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).

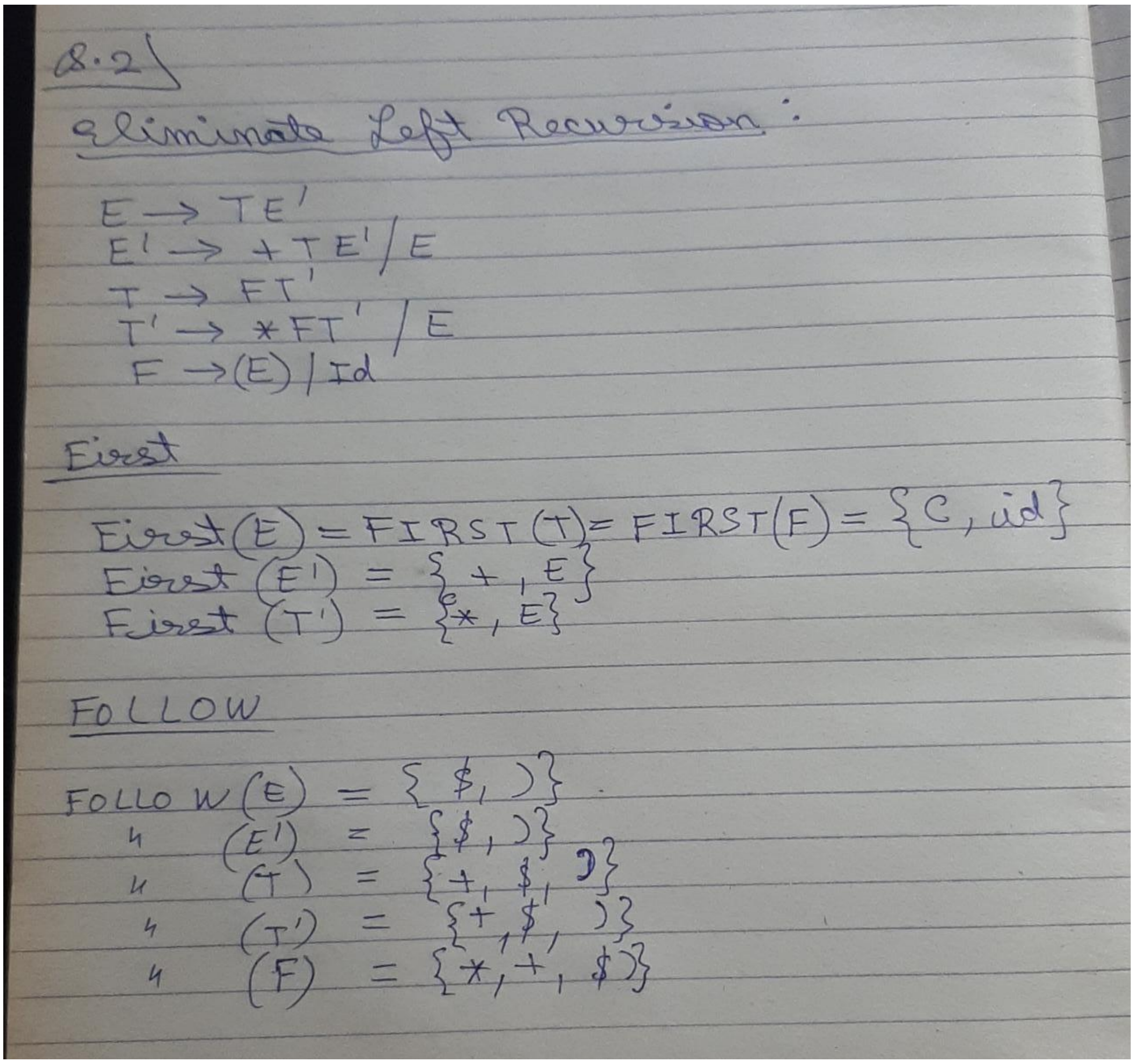
Code Generation

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.

Q.2



Q.2)

Eliminate Left Recursion:

$E \rightarrow TE'$
$E' \rightarrow +TE' / \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' / \epsilon$
$F \rightarrow (E) / Id$

First

First(E) = FIRST(T) = FIRST(F) = $\{(, id\}$
First(E') = $\{+, \epsilon\}$
First(T') = $\{*, \epsilon\}$

FOLLOW

FOLLOW(E) = $\{\$,)\}$
" (E') = $\{\$,)\}$
" (T) = $\{+, \$,)\}$
" (T') = $\{+, \$,)\}$
" (F) = $\{*, +, \$\}$

Parsing Table

NTs	+	*	(	)	id	$
E			E→TE'		E→TE'	
E'	E'→+TE'			E'→e		E'→e
T			T→FT'		T→FT'	
T'	T'→E	T'→*FT'		T'→e		T'→e
F			F→(E)		F→id	

W= id * id + id

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'T' F *	* id + id $	T' → *FT'
$ E'T' F	id + id $	F → id
$ E'T' id	id + id $	
$ E'T'	+ id $	T' → e
$ E'	+ id $	E' → +TE'
$ E'T +	+ id $	
$ E'T	id $	T → FT'
$ E'T'F	id $	F → id
$ E'T' id	id $	
$ E'T'	$	
$ E'	$	