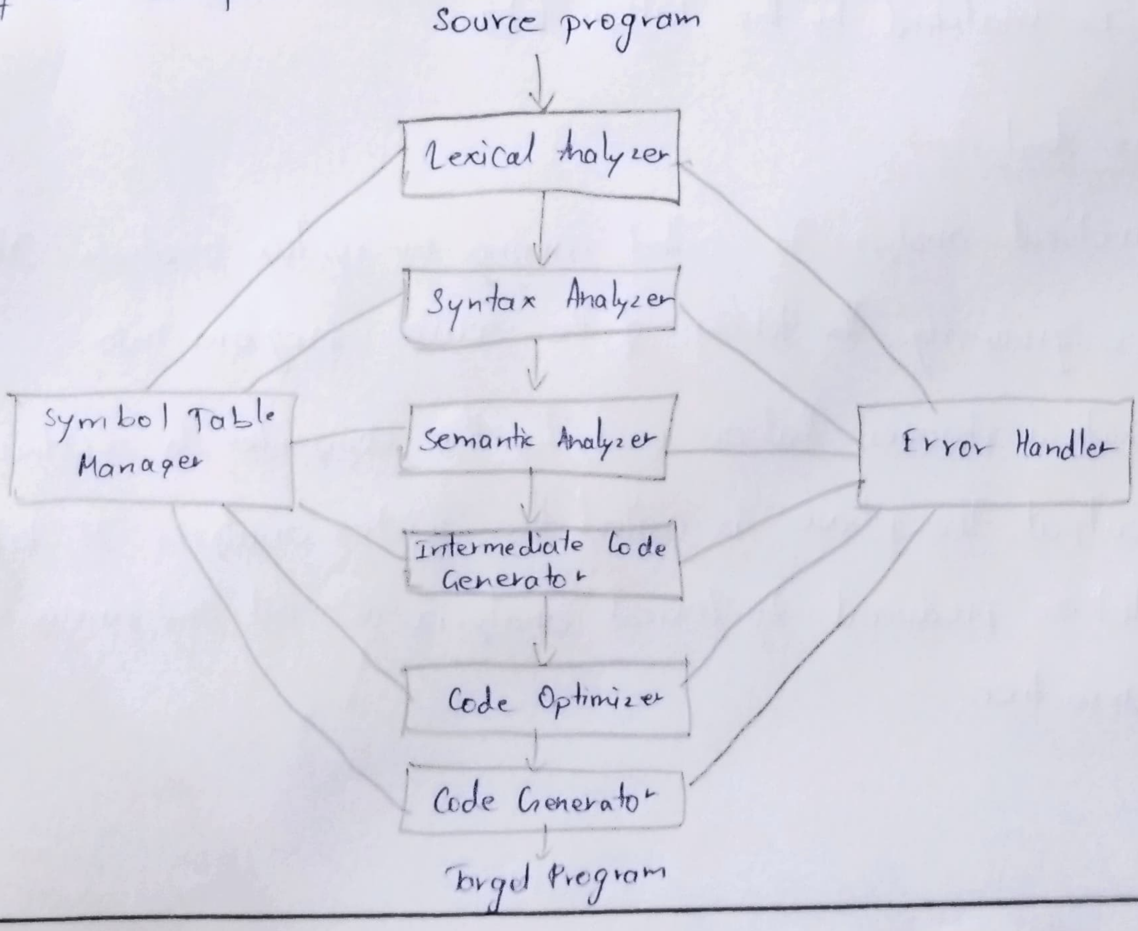
CAE - 1

Name: Mohnish Devaraj	Subject Name: Compiler Design
Reg No: 39110636	Subject code: SCSA1604
Roll No: 19S115398	Total pages: 8
Date: 03 Feb 2022	

PART - B

7 A compiler operates, each of which transforms the source program from one representation to another. The structure of compiler. The first three phases form the analysis portion of the compiler



Lexical Analysis

In a compiler, linear analysis is called lexical analysis or scanning.

- It is a first phase of a compiler
- Lexical Analyzer is also known as scanner
- Reads the characters in the source program from left to right and groups the characters into stream of Tokens.
- Such identifier, keyword, Punctuation character, Operator.
- Pattern: Rule for a set of strings in the input for which a token is output.
- A Lexeme is a sequence of characters in the source code that is matched by the Pattern token.

Syntax Analysis

Hierarchical analysis is called parsing or syntax analysis. It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize the output. This phase is called the syntax analysis. It takes the token produced by lexical analysis as input and generates a parse tree.

Sematic Analysis

The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phase. An important component of semantic analysis is type checking i.e., whether the operands are type compatible. For example, a real number used to index an array.

Intermediate Code Generation:

After semantic analysis, some compilers generate an explicit intermediate representation of the source program. The represent-ation should be easy to produce and easy to translate into the target program. These are varieties of forms, three address code, Postfix notation, Syntax tree.

Code Optimization

This phase attempts to improve the intermediate code, so that faster running machine code will result. There is a way to perform the same calculation for the address code. There are various techniques such as Common sub-expression elimination, Dead code elimination, Constant folding.

Code Generation:

The final phase of the compiler is the generation of target code, Consisting of relocatable machine code or assembly code

The intermediate instructions are each translated into sequence of machine instrutions that perform the same task. A crucial aspect is the assignment of variables to register.

Symbol-table management

An essential function of a compiler is to record the identifiers used in the source program and collect its information. A symbol table is a data structure containing a record for each identifier with fields for attributes.

Error-Handling and Reporting:

Each phase can encounter errors. After detecting on error a phase must deal that error, so that compilation can proceed, allowing fur ther errors to be detected. Lexical phase Can detect error when the characters remaining in the input do not form any token.

PART - A

(1) The lexical Analyzer scans the input from left to right one character at a time. It uses two pointers begin ptr (bp) and forward to keep track of the pointer of the input scanned. A block of data is first read into a buffer, and the second by lexical analyzer.

(5) Handling prunning is the general approach used in shift-and-reduce parsing. A Handle is a substring that matches the body of a production. Handle reduction is a step in the reverse of rightmost derviation. A rightmost derivation in reverse can be obtained by handle pruning.

(6)
(4) Ambiguous grammer

-> In ambiguous grommer, the leftmost and rightmost derivation are not same.

-> Amount of non-terminals in ambiguous grammer is less than in unambiguous.

-> Length of the parse tree is ambiguous grammer is comparitively short.

-> Ambiguous grammer generates more than one parse tree.

Unambiguous grammer

-> In unambiguous grammer, the leftmost and rightmost derivation are same.

-> Amount of non-terminal in ambigvous grammer is more than in ambiguous grammer.

-> Length of the parse tree is unambiguous grammer is comparatively large

-> Unambiguous grammer generates only one parse tree

39110636 (6)

PART - B

(8) S -> (L) | a
L -> L, S | S

S -> (L)

S -> a

L -> SL'

L' -> S

L' -> ε

L -> S

<u>step 1</u>

FIRST (s) = {ϵ, a)

FIRST (L) = {ε FIRST (s)} = {c, a}

FIRST (L') = {, , ε)

<u>step 2</u>:

FOLLOW (s) = {$, FIRST (L') = (s', $)

FOLLOW (L) = {) }

FOLLOW (L') = FOLLOW (L) = {) }

	a	c	,	)	$
S	S -> a	S -> (L)			
L	L -> S	L -> L'			
L'					
			L' -> , s	L' -> s	

Parse string ((a,a),a)

stack	I/P string	Actions
$ ε	((a,a),a)$	S→(L) (push)
$) L $	((a,a), a)$	pop
$) L	(a,a),a) $	L→L' (push)
$) L' $	(a, a) , a) $	S→(L) (push)
$) L') L $	(a ,a),a) $	pop
$) L') L	a, a) , a) $	L→S (push)
$) L') S	a , a) , a) $	~~L~~ S→a (push)
$) L') $	~~a~~ , a) , a) $	pop
$) L')) L'	~~,~~ a) , a) $	L'→S (push)
$)) L'	, a) , a) $	pop
$)) S	a), a) $	S→a (push)
$)) a	~~a~~) , a) $	pop
$))	) , a) $	pop
$)	, a) $	L'→S (push)
$) S	, a) $	pop
$) S	a) $	S→a (push)
$) ~~a~~	a) $	pop
$)	) $	pop
$	$	Accept

PART - A

(5) LR(0) ~~items~~ of the grammer:

$S \to AS \mid b$

$A \to SA \mid a$

$S' \to \cdot S$

$S \to \cdot AS$

$S \to \cdot b$

$A \to \cdot SA$

$A \to \cdot a$

(3) A reduce/reduce conflict occurs if there are two or more rule that apply to the same ~~eg~~ sequence of input. For example, here is an erroneous attemp ~~to~~ to define a sequence of zero or more word grouping

Eg: $A \to a$
$B \to b$

shift reduce parser attens for the construction of parse in a similar manner as done in bottom-up ~~parsing~~ parsing

$A \to b$

$X \to X_{r} a$