CA - I.

Page No.
Date

Q. 1. What is Compiler ? Compare the Compiler and interpreter.

- A compiler is a computer program that helps in translating the computer code from one programming language into another language.
- Compilers are essential for making human-readable code executable on machines, enabling the development and deployment of software applications.

	Compiler	Interpreter.
1.	Compiler is a language processor which converts high level language.	1. Interpreter is a language processor which converts high level language.
2.	It takes the entire program at a time.	2. It takes a single line of code or instruction at a time.
3.	Error detection is difficult.	3. Error detection is easier.
4.	It consumes less time.	4. It consumes more time.

5. Compiler is more efficient.	5. Interpreter is less efficient.
6. Speed is comparatively faster.	6. Speed is slower
7. programming language like :- C, C++.	7. programming language like :- PHP, python.

8. Define the terms :- Tokens, patterns and lexemes.

→

- Tokens :-

A sequence of characters treated as the fundamental unit of a programming language that is not further broken down.

- patterns :-

It specifies a set of rules that a scanner follows to create a token.

- Lexemes :-

A lexemes is an actual character sequence forming a specific instance of a token.

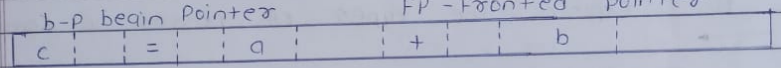
Q.2

Page No.
Date

Q.2. What is input buffering ? Explain is types.

→

- The input buffering is used to store the source program statement for lexical analysis phase to seprate the tokens.
- The memory slots which can accuphy single character at time in sequential manner by using the pointers.
- The begin pointer and forward pointer initially point to the 1st character into the string in input buffer.



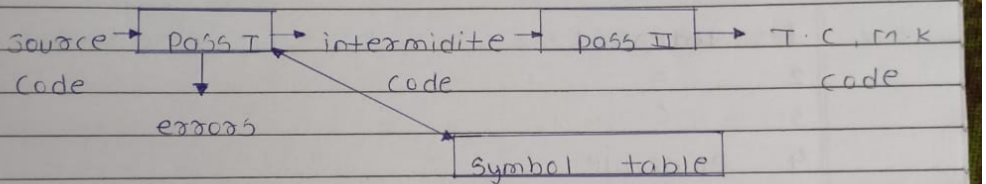
- It is incremented until it's find the white space. once the white space is find out, til NO. of characteriters are grouped and considered as token & lexeme.
- There are 2 types of input buffering mechanism :-
  1. One buffer and
  2. Two buffer.

- Both works as stated about but in a Two buffer mechanism the memory is available is double as compared to one buffer mechanism.

2. Two pass Compiler :-

In this type of compiler the phases of compiler are devided into two parts. The pass one contain mechanism analysis, Sentaanalysis and Semantic.

- In pass Two the intermidiate code is optimize and the machine code or program is generated why impliimenting pass Two it refers the Symbol table.

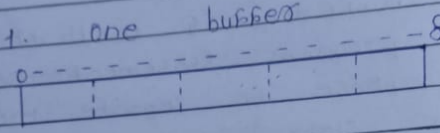


3. Multi pass Compiler :-

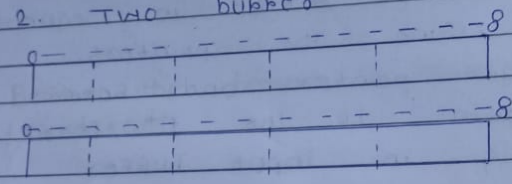
- In this type of compiler pass 1 has lexical, Sentax, Semantics phases then the intermidiate code generation has seprate phase.
- Finaly the code optimization and code generation are impliimented in pass Two the pass 1 also known as front end. pass 2 known as backend.
- The machine dependent code is converted from intermidiate code generation phase.

- which helps for language statement present in source code.

1. One buffer



2. Two buffer

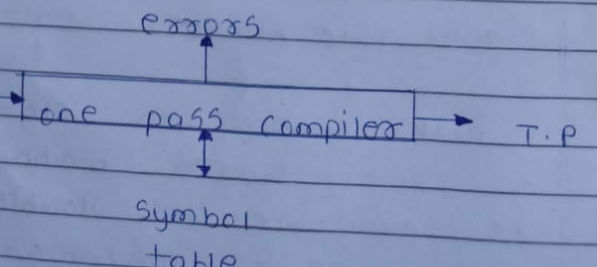


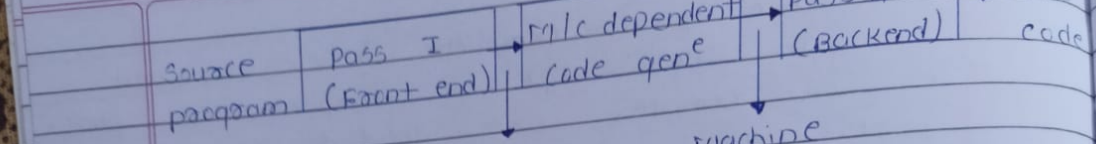
B. Explain the types of passes in Compiler.

→

- The pages of Compiler are grouped together or either seprated into following types.

1. One pass or single pass compiler in this type all the phases of compiler are implimented in sequential manner in single pass.





Q.3. Separate the tokens for Interest = (Principle * N * Rate) / 100 and construct symbol table.

→

1. Interest.
2. =
3. (
4. Principle
5. *
6. N
7. *
8. Rate
9.)
10. /
11. 100

Symbol table :-

Identifier :- Interest, principle, N, Rate
operator :- =, *, *, /
value :- 100.

Page No.
Date

8. List out and explain the phases in front end of Compiler.

→ The front end of a compiler is responsible for analyzing the source code and preparing it for the later stages of compilation. It generally includes the following phases:

1. Lexical Analysis :-
- To read the source code and convert it into tokens, which are sequences of characters that represent the smallest unit of meaning.
- output :- A stream of tokens.

2. Syntax Analysis :-
- To analyze the token stream according to the grammatical structure of the programming language. It checks whether the tokens form valid sentences and constructs a parse tree or syntax tree.
- output :- A parse tree or abstract syntax tree (AST).

3. Semantic Analysis :-
- To ensure the syntactically correct code makes sense according to the language rules. It involves type checking, scope resolution, & checking for other semantic errors.
- output :- An annotated syntax tree (AST) with semantic information.

B. List out and explain the phases in front end of Compiler.

→

- The front end of a compiler is responsible for analyzing the source code and preparing it for the later stages of compilation. It generally includes the following phases :-

1. Lexical Analysis :-
- To read the source code and convert it into tokens, which are sequences of characters that represent the smallest unit of meaning.
- output :- A stream of tokens.

2. Syntax Analysis :-
- To analyze the token stream according to the grammatical structure of the programming language. It checks whether the tokens form valid sentences and constructs a parse tree or syntax tree.
- output :- A parse tree or abstract syntax tree (AST).

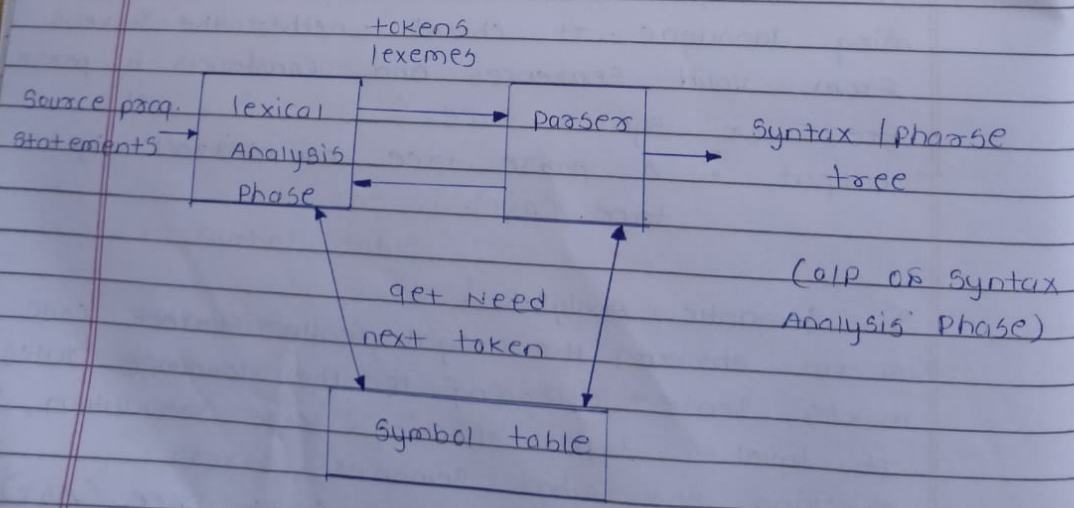
3. Semantic Analysis :-
- To ensure the syntactically correct code makes sense according to the language rules. It involves type checking, scope resolution, & checking for other semantic errors.
- output :- An annotated syntax tree (AST) with semantic information.

4. Intermediate Code generation :-
- To translate the annotated syntax tree into an intermediate representation (IR) that is easier for the back end of the compiler to optimize and translate into machine code.
- output :- Intermediate code or intermediate representation (IR)

5. optimization :-
- To improve the intermidiate code by making it more effcient. These are typically machine-independent optimization.
- output :- Optimized intermediate code.

Q.4. Explain the role of lexical Analyzer with diagram.

→



Date

- In lexical Analysis phase the source statement are scaned and generates the takens or lexemes present in respective statement this process is done by reading char, by character data from source code.
- The token or lexeme is variable which is use the in expression or calculation.
- The characteritics of token such as name, type, value, memory location (add) etc. is also reflected into symbol table.
- Consider the example of c language in which if any statement or program is return then the variable or identifier is other than 32 keywords.
- Consider the following code.

```
int a;
float b, c;
a = b+c;
```

- After lexical Analysis phase for above statement the takens are a, b, c and symbol table is as follows.
- The generated takens or lexeme by lexical Analysis phase given input to the parser.
- Now parser generates the syntax tree or phrase tree to validate each statement is correct form by sentatically.

3. List and explain the different Compiler Construction tools.

1. Lexical Analyzers :-
- To generate lexical analyzers, which are programs that convert a sequence of characters from source code into a sequence of tokens. These tools take regular expressions as input and produce C code that performs lexical analysis.

2. Parser generators :-
- To generate parsers that analyze the token sequences produced by lexical analyzers and construct parse trees or abstract syntax trees (ASTs). These tools take context-free grammers as input and produce code for syntax analysis.

3. Code optimizers :-
- To improve the intermidiate code by making it more efficient in terms of execution speed, memory usage, and other performance metrics.

4. Code generators :-
- To translate intermediate code into machine code specific to the target architecture. These tools handle the details of instruction selection, register allocation.

Page No.
Date

5. Assemblers :-

- To convert assembly language code into machine code. Assemblers are used after the code generation phase to produce executable programs.

6. Build Automation Tools :-

- To automate the process of compiling and linking programs. These tools manage dependencies and ensure that the compilation process is executed correctly and efficiently.

11.

MSE - 2023-24

(19/4/24)

Page No.
Date

Q. 2. How left Recursion is removed from grammar.

→

- Left recursion in a grammar can cause problems for certain types of parsers, particularly recursive descent parsers, which can enter into infite recursion. To make a grammar suitable for these parsers, left recursion needs to be eliminated. Here's how you can remove left recursion from a context-free grammar :-

* Direct Left Recursion :-

- For a production of the form :-

  $A \rightarrow A\alpha | \beta$

- where A is a non-terminal, α is a sequence of grammar symbols, and β does not start with A, direct left recursion can be eliminated as follows :-

1. Identify the direct left recursive and nonleft recursive productions.
2. Rewrite the productions using new non-terminals to eliminate the left recursion.

- Steps :-

1. original production :-

   $A \rightarrow A\alpha_1 | A\alpha_2 | \cdots | A\alpha_n | \beta_1 | \beta_2 | \cdots | \beta_m$

   where each β_i does not start with A

2. Transform to :

$A \rightarrow B_1 A' \mid B_2 A' \mid \cdots \mid B_m A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_n A' \mid \epsilon$

where A' is a new non-terminal and ϵ denotes the empty string.

* Indirect left recursion :-

- Indirect left recursion occurs when a non-terminal A indirectly leads to itself via other non-terminals.

- Steps :-

1. Identify and Eliminate Direct left recursion : Apply the direct left recursion elimation method to each non-terminal.

2. Substitue Non-terminals :- Substitue non-terminals in productions where indirect left recursion occurs.

Q2 A Solve the following.

1) Implement shift reduce parsing for following grammar.

$S \rightarrow S+S$

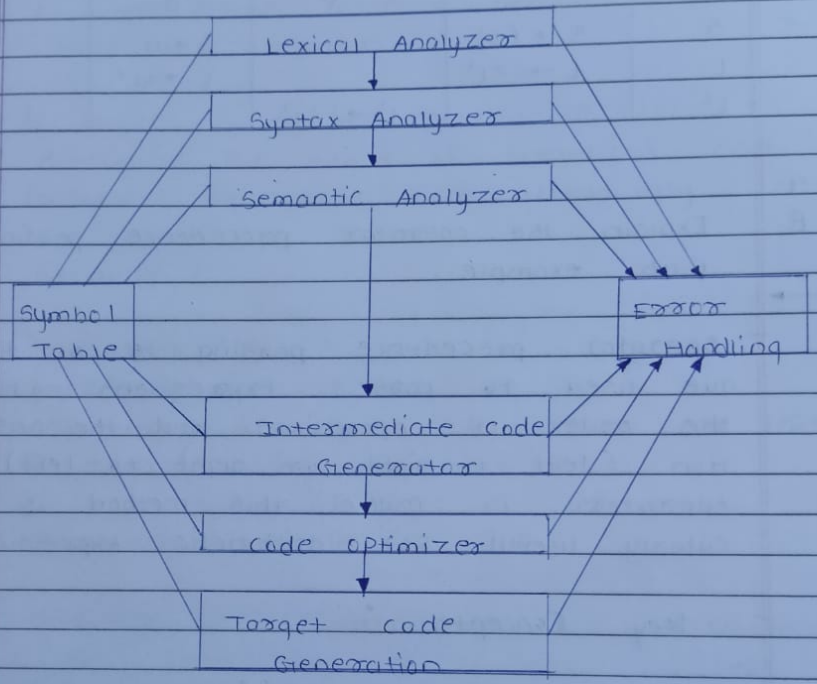
$S \rightarrow S*S$

$S \rightarrow id$

perform shift reduce parsing for following input string "id + id + id".

Page No.
Date

code that can be executed by the computer hardware.



B. Implement predictive parsing for grammar G1 given by

$S \rightarrow (L) \mid a$

$L \rightarrow SL'$

$L' \rightarrow SL' \mid \epsilon$

	First	Follow
S	{ (, a }	{ $,), }
L	{ (, a }	{) }
L'	{ ε,) }	{) }

- operator precedence parsing steps :-

1. precedence Table :- Define a precedence table for operators.

2. Parsing :- use the precedence table to determine the order of operations and construct the parse tree accordingly.

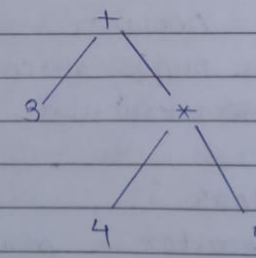
Example :-

- operators :- + , *
- precedence :- * > +
- Associativity :- Both + and * are left-associative.

- precedence Table :-

operator	precedence	Associativity
*	2	Left
+	1	Left

- The final parse tree :-



11.

Page No.
Date

Parse Table :-

	(	)	a	$
S	S → (L)		S → a	
L	L → SL'		L → SL'	
L'		L' →) SL'		L' → ϵ

Q.4.
B. Explain the operator precedence parsing with example.

→

- operator precedence parsing is a technique used to parse expressions where the order of operations and the association (left-to-right or right-to-left) of operators is crucial. this method is particularly useful for arithmetic expressions.

- Key Concepts :-

1. operator precedence :-
Each operator has a precedence level which determines the order in which operations are performed.

2. operator Associativity :-
- This determines the order of operations when two operators of the same precedence level appear in an expression. operators can be left-associative or right-associative.

- Framework, responsible for generating machine specific code from the intermediate representation.

5. Debuggers and profilers :-

- GDB :- A powerful debugger used to test and debug programs written in C, C++ and other languages.

C. What are the contents of a symbol table? Explain in detail the symbol table organization for Block-structured languages.

→

- Symbol table is used to store the information about the occurance of various entities such as objects, classes, variable name, interface function name etc.

- It is used by both the analyses and synthesis phase.

- Symbol table is an important data structured created and maintained by the compiler is order to keep track by cp semantic of variable it source information about instances of various entity sure as various & function such as variable & function names, classes, object etc.

Regular End Semester
Summer - 2022

(11/08/2022)

Page No.
Date

Q.1. Define Compiler ? State some commonly used compiler - construction tools.

A. →

- A Compiler is a specialized program that translates source code written in a high-level programming language into machine code, bytecode, or another programming language.

* Commonly used compiler-construction Tools :-

1. Lexical Analyzes :-
- A fast lexical analyzer generator used to produce lexers, which convert a sequence of characters into a sequence of tokens.

2. Syntax Analyzers :-
- Yet Another Compiler, used to produce a parser that converts tokens into a parse tree.

3. Semantic Analyzers :-
- Clang Static Analyzer :- used for analyzing code for bugs, often part of the Clang compiler infrastructure.

4. Code Generators :-
- LLVM Code Generator :- part of the LLVM

- Symbol table organization for Block - structure languages.

1. Global Symbol table :-
- The global symbol is a centralized data structure that store information.

2. Local Symbol table :-
- when local symbol tables are associated with individual blocks or scope.

3. Scope Resolution :-
- During semantic analysers the compiler resolves identifiers by searching through the symbol.

4. Handling Nested Blocks :-
- As the compiler traverse nested blocks during paring.

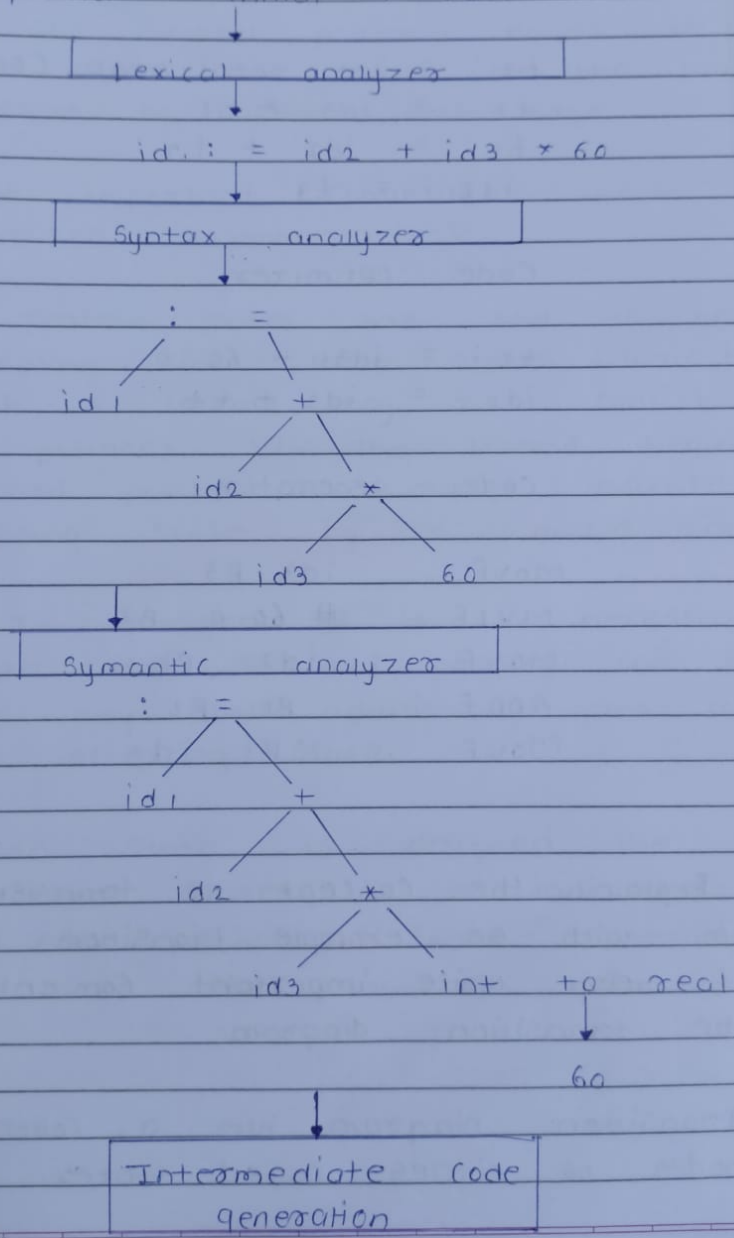
5. memory management :-
- Symbol table also pias a role in memory management during code generation.

B. Explain how the assignment statements "position = initial + rate * 60" is grouped into the lexemes and mapped into the tokens passed on the syntax analyzer.

Page No.
Date

phase result for position = initial + rate * 60

position = initial + rate * 60



represents a condition that could occur during the process scanning the input looking for a lexeme that matches one of the several patterns edges are directed from one state of the transition diagram to another each edge.

- Some important conventions about transition diagram are :-

1. Certain states are said to be accepting final these states indicates that a lexeme may not consist of all positions b/w the lexeme begin & forward pointer we always indicate as accepting state by a double circle.

2. In addition, if it is necessary to return the forward pointer one position then way shall additionally plan a near that accepting state.

3. one state is designed the state or initial states it is indicated by an edge labeled "start" entering from now nere the transition diagram.

- First operand represents source & second operand represents destination in the machine code.

```
t1 : = int total real (60)
t2 : = id3 * t1
t3 : = id2 + t2
id1 : = t3
```

Code optimizer

```
t1 : = id3 * 60.0
id1 : = id2 + t1
```

Code generator

```
MOVF    id3, R3
MVLF    # 60.0, R2
MOVF    id2, R1
ADDF    R1, R2
MOVF    R1, id1
```

Q.2.

A. Explain the concept of transistion diagram with an example transition diagram of relop write important convention about the transition diagram.

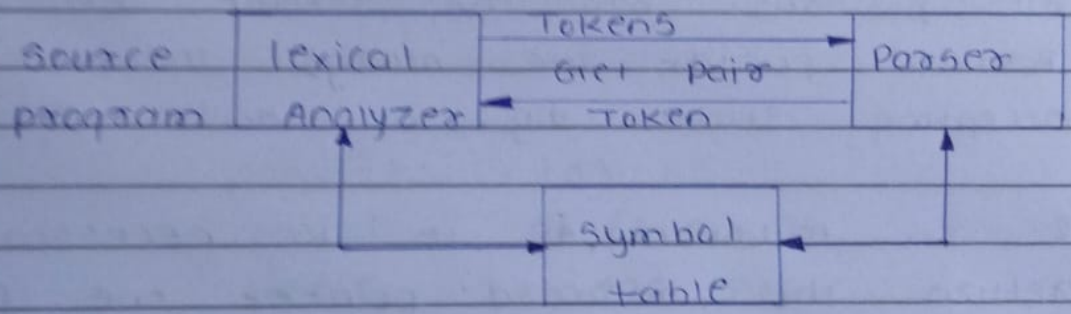
→

- Transition diagram has a collection of nodes or circles called states each state

Page No.
Date

6. In lexical analyzer, explain for example how tokens, patterns & lexemes are related.

→ The LA is the first phase of a compiler lexical analyzed is called as linear analysis or scanning in this phase the stream of characters making up the source program is read from the right and grouped into tokens that are sequence of character having a collective meaning.

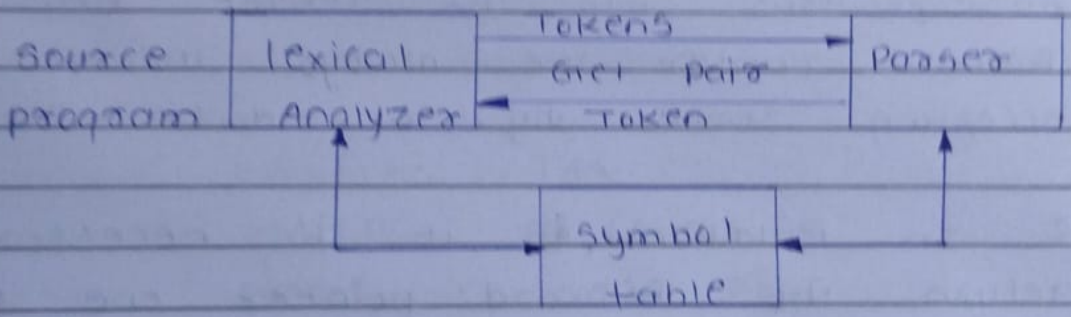


Token :-

- Token is a sequence of characters that can be treated as a single logical entity typically token are
1. keywords
2. Indentifier
3. operator
4. special Symbol
5. Constant

6. In lexical analyzer, explain for example how tokens, patterns & lexemes are related.

→ The LA is the first phase of a compiler lexical analyzed is called as linear analysis or scanning in this phase the stream of characters making up the source program is read from the right and grouped into tokens that are sequese of character having a collective meaning.



Token :-

Token is a sequence of characters that can be treated as a single logical entity typically token are

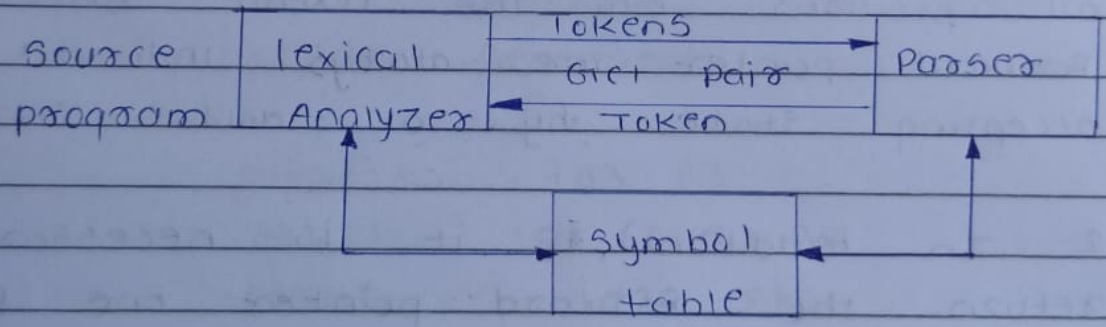
1. keywords
2. Indentifier
3. operator
4. speal symbol
5. Constantsxce

Page No.
Date

B. In lexical analyzes, explain for example how tokens, patterns & lexemes are related.

→

- The LA is the first phase of a compiler lexical analyzed is called as linear analysis or scanning in this phase the stream of characters making up the source program is read from the right and grouped into tokens that are sequere of character having a collective meaning.



Token :-

- Token is a sequence of characters that can be treated as a single logical entity typically token are
1. keywords
2. Indentifier
3. operator
4. speal Symbol
5. Constantarce

- Patterns :-

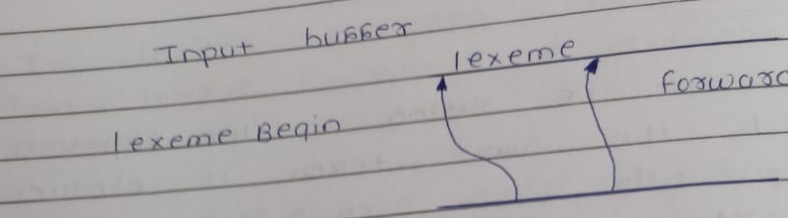
- A set of string in the input for which the same token is produced as output this set of string is described by the rule called pattern

- Lexemes :-

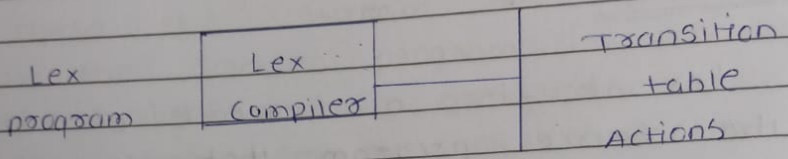
- A lexeme is a sequence of character in the source program that is matched by the pattern for a token

C. Explain the structure of the lexical-analyzer generator. Show the construction of an NFA from a lex program.

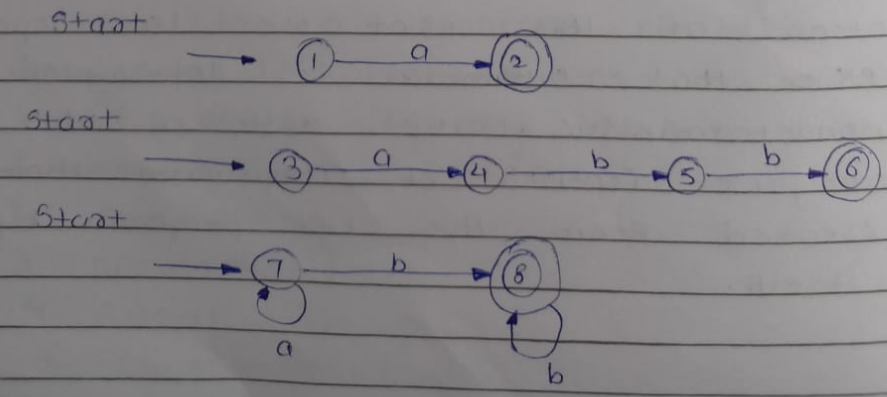
- 1. The structure of the Generated Analyzer
- overviews the architecture of a lexical analyzer generated by Lex. The program that serves as the lexical analyzer includes a fixed program that simulates an automaton : at this point we leave open whether that automaton is deterministic or nondeterministic. The rest of the lexical analyzer consists of components that are created from the Lex program by lex itself.



Automaton simulator



- Eventually, the NFA simulation reaches a point on the input where there are no next states. At that point, there is no hope that any longer prefix of the input would ever get the NFA to an accepting state; rather, the set of states will always be empty. Thus, we are ready to decide on the longest prefix that is a lexeme matching some pattern



Q.3.
A. How left recursion is eliminated ? Explain with algorithms & Example.

→

1. Indentify left recursive productions :-
- Look for production of the form
$A \rightarrow Aa$

2. Create New non-terminals :-
For each left-recursive non-terminal introduce a new non-terminals.

3. Separate production :-
- Rewrite the production for A into two groups one for non-left recursive.

4. Replace left Recursive production :-
- Replace the left-recursive production $A \rightarrow Aa$ with $A \rightarrow \beta A'$ & $A' \rightarrow \alpha A' | \epsilon$, where β is a string of terminals and for non-terminals that does not start A, & ϵ represents an empty string.

5. update other production :-
- update other production involving A where neccessary.

B. What is meant by shift-reduce parsing? Explain the configuration of a shift-reduce parser on input "id * d2".

- Shift Reduce parsing used a stack a hold grammer symbols & input buffer to hold string to be used parsed, because handled always appear at the top of the stack i.e. theres no need to look deper into the stack.

- A shift reduce parser has just four actions :-

1. Shift - next word is shifited code the stack (input symbol) until a handle on format.

2. Reduce Right end of handle in at top of stack locate left end of handle within the stack pop handles off stack & pasn appropriate use.

3. Accept - stack parsin an sucessful completion of parse & repet successes.

4. Shift - Reduce :-
- Both a shift action and a reduce action are possible in the time state (should are shift or reduce).

5. Reduces Reduce :-
- Two or more distint reduce action are possible in the same state which production should we reduce with.

6. Construct a predictive parsing table for the Grammer [E → E+T | T, T → T*F | F, F → (E) | id.

- Step 1 :-
- Suppose is the given grammer in left Recursive then convert the given grammer C and ϵ into non-left Recursive grammer.

E → TE'
E' → +TE' | E
T → FT'
T' → *|FT'|E
F → (E) | id

Step 2 :-
- The Variable are :- { E, E', T, T', F }
- Terminal are :- { +, *, (,), id } &
- Computation of First () s;

FIRST (F) = FIRST ((E)) ∪ FIRST (id) = { (, id }
FIRST (T') = FIRST (*FT') ∪ FIRST (E) = { *, ϵ }
FIRST (T) = FIRST (F T') = FIRST (F) = { (, id }
FIRST (E') = FIRST (T TE') ∪ FIRST (ϵ) = { +, ϵ }
FIRST (E) = FIRST (TE') = FIRST (T) = { (, id }

Page No.
Date

Computation of Follow () sets

Follow (E) = {$} ∪ FIRST ()) = {$,)}, F → (E) E → TE'
Follow (E') = Follow (E) = {$,)}
Follow (T) = (FIRST (E') - {ε}) ∪
Follow (E) ∪ Follow (E') = {T}
Follow (T') = Follow (T) = {+,), $}
Follow (F) = (FIRST (T') - {ε}) ∪
Follow (T) ∪ Follow (T') =
{*, +, $}

Step 3 :-
Construction of parsing table :-

Terminal variables	+	*	(	)	id	$
E		E→TE		E→TE'		
E'	E'→		E'→ε			E'→ε
T		T→FT'		T→FT'		
T'	T→ε	T'→		T'→ε		T'→ε
F		E→(E)		F→id		

Page No.
Date

Q.4.
A. Differentiate between synthesized and inherited attributes. Also, define what is meant by annotated parse tree.

Synthesized Attributes	inherited attributes
1. use values from children & from constants.	1. use values from parent, constant, & siblings.
2. S-attributes grammars.	2. Directly express context
3. Evaluate in a single bottom-up pass	3. Can rewrite to avoid them
4. Good match to LR parsing	4. Thought to be more natural Not easily done at parse time.
5. The production must have non-terminal as its head.	5. The production must have non-terminal as a symbol in its body.

- Annotated parse tree :-
- An annotated parse tree is a parse tree in which each node is augmented with attribute values as defined by the grammar's attribute rules.

B. Explain Constructing Syntax trees for simple expressions involving only binary operators + & -, State the use of leaf & node in the Syntax tree.

→

1. Identify operators :-
- identify the operators in the expression in this case, we have only addition (+) & substraction (-)

2. Identify operands :-
- Identify the operands which are the values are sub-expression the operators act upon the can be number or other expression.

3. Establish Hierarchi :-
- Determine the Hierarchi of operations based the preproduce of the operaters.

4. Build the tree :-
- Start with the root of the tree which corresponds to the entire expression then recursively construct subtrees for each

Page No.
Date

C. Explain in brief about Type checking and type Conversion.

→

- Type Checking :-

- Type checking is the process of verifying and enforcing the constraints of types in a programming language. It ensures that the operations in a program are performed on compatible data types.

- Static Type checking :-

- This is performed at compile-time. The compiler checks the type correctness of the code before execution. Languages like Java, C, and C++ use static type checking.

• Advantages :-

- Errors are caught early in the development process, which can prevent many runtime errors.

- Type Conversion :-

- Type conversion, also known as type casting, is the process of converting a value from ~~the~~ one data type to another. This can be either implicit (automatic) or explicit (manual).

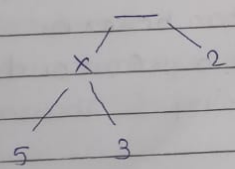
operation and it's operand.

- For example, Consider the expression
  5 + 3 - 2

- The root of the tree could be the subtraction operator (-).

- The left child root could be the addition operator (+) which as it to left operand and 3 it 1, right operand.

- So, the syntax tree would look like this,



- use of leaf node is syntax tree. each leaf node represents an operand where as each inner node represents the operator tree is considered inte, the syntax tree in a syntax tree each leaf node stands ~~to~~ in for an operand & each inside node for an operator.

3. Improving Scalability :-
- Ensuring that the code can handle increased load or data sizes effectively, which is crucial for applications that need to scale.

- Directed Acyclic Graph (DAG) Representation of Basic Blocks :-

- A Directed Acyclic graph (DAG) is a useful tool in code optimization, particularly for representing basic blocks in a way that highlights dependencies between instructions and allows for optimizers like common subexpression elimination, instruction recording and dead code elimation.

- Steps to Construct a DAG for a Basic Block :-

1. Indentify Statements :- Extract individual statements or instructions in the basic block.

2. Determine Dependencies :- Identify dependncies between these instructions based on variable usage.

3. Create Nodes :- Create a node for each unique operation and value.

4. Create Edges :- Draw directed edges from operand nodes to the operation nodes that use them.

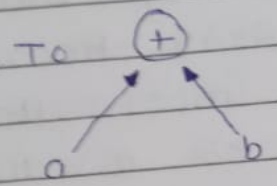
Example :-

$T_0 = a + b$

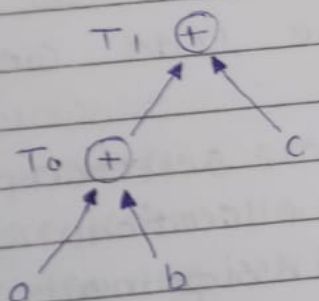
$T_1 = T_0 + c$

$d = T_0 + T_1$

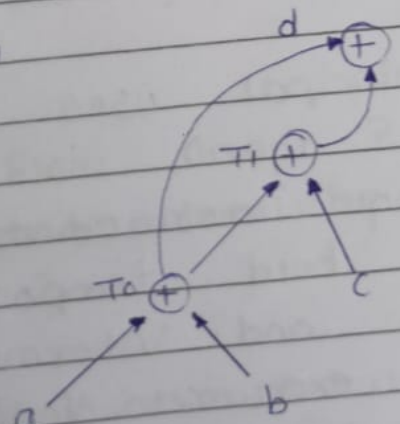
$T_0 = a + b$



$T_1 = T_0 + c$



$d = T_0 + T_1$



Page No.
Date

B. Explain the code generation algorithm with three - address instructions. State the four principal uses of registers.

→

- Code generation is a critical phase in a compiler where intermediate code is translated into machine code or assembly code. Three address instructions are a common intermidate representation used in this phase. They are called "three - address" instructions because each instruction can have at most three operands. The general form of a three - address instruction.

- Steps in the Code Generation Algorithm :-

1. Instruction selection
2. Register allocation
3. Register Assignment
4. Instruction scheduling

- Four principal uses of Registers

1. Storing intermediate values :-
- Register hold temporary results of calculations and intermediate data during instruction execution, significantly speeding up computations compared to accessing data from main memory.

Page No.
Date

B. Explain the code generation algorithm with three-address instructions. State the four principal uses of registers.

→

- Code generation is a critical phase in a compiler where intermediate code is translated into machine code or assembly code. Three address instructions are a common intermidate representation used in this phase. They are called "Three-address" instructions because each instruction can have at most three operands. The general form of a three-address instruction.

- Steps in the Code Generation Algorithm :-

1. Instruction selection
2. Register allocation
3. Register Assignment
4. Instruction scheduling

- Four principal uses of Registers

1. Storing intermediate values :-
- Register hold temporary results of calculations and intermediate data during instruction execution, significantly speeding up computations compared to accessing data from main memory.

Page No.
Date

B. Explain the code generation algorithm with three - address instructions. State the four principal uses of registers.

→

- Code generation is a critical phase in a compiler where intermediate code is translated into machine code or assembly code. Three address instructions are a common intermidate representation used in this phase. They are called "three - address" instructions because each instruction can have at most three operands. The general form of a three - address instruction :

- Steps in the Code Generation Algorithm :-

1. Instruction selection
2. Register allocation
3. Register Assignment
4. Instruction scheduling

- Four principal uses of Registers

1. Storing intermediate values :-
- Register hold temporary results of calculations and intermediate data during instruction execution, significantly speeding up computations compared to accessing data from main memory.

Page No.
Date

B. Explain the code generation algorithm with three - address instructions. State the four principal uses of registers.

→

- Code generation is a critical phase in a compiler where intermediate code is translated into machine code or assembly code. Three address instructions are a common intermidate representation used in this phase. They are called "three - address" instructions because each instruction can have at most three operands. The general form of a three - address instruction :

- Steps in the Code Generation Algorithm :-

1. Instruction selection
2. Register allocation
3. Register Assignment
4. Instruction Scheduling

- Four principal uses of Registers

1. Storing intermediate values :-

- Register hold temporary results of calculations and intermediate data during instruction execution, significantly speeding up computations compared to accessing data from main memory.

2. Holding Addresses :-
- Registers can hold memory addresses for direct and indirect addressing modes, facilitating access to data in memory. This includes base addresses, pointers, & index registers used in array and pointer calculations.

3. Instruction Execution :-
- Certain registers are dedicated to specific functions within the CPU, such as the instruction register (IR) which hold the current instruction being executed, and the program counter (PC) which holds the address of the next instruction.

4. Special purpose :-
- Some registers serve special purpose like status registers, which store flags indicating the status of the processor, including zero, carry, overflow and sign flags.

C. What is a flow Graph ? Explain how a given program can be converted into a flow graph ?

→
- A flow graph, also known as a control flow graph (CFG), is a representation of all paths that might be traversed throu-

gh a program during its execution. Each node in the graph represents a basic block, which is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit. Directed edges between nodes represent control flow paths.

- Components of a flow graph :-
  1. Basic Blocks.
  2. Directed Edges.

- Converting a program into a flow graph :-

1. Identify Basic Blocks :-
- A basic block starts with the first instruction or an instruction that is the target of a branch instruction
- A basic block ends at a branch instruction or when the next instruction is a target of a branch.

2. Construct Nodes for Each Basic Block :-
- Each basic block identified in the previous step becomes a node in the flow graph.

Page No.
Date

3. Identify Control Flow :-
- Determine the possible transitions from one basic block to another. This involves looking at the branch and jump instructions to see where control might flow next.

4. Draw Edges :-
- For each possible transition, draw a directed edge from the source basic block to the target basic block.

24/05/24

11.

- Implicit Conversion :-
- The language automatically converts one data type to another without explicit instruction from the programmer. This is often done to ensure type compatibility in operations.

Q.5.
A. What is the purpose of code optimization ? Explain the DAG representation of basic blocks with example.

→

- purpose of code optimization :-
- Code optimization is a crucial phase in the compilication process that aims to improve the performance and efficiency of a program. The primary objectives of code optimization include :-

1. Improving Execution speed :-
- Reducing the runtime of a program by eliminating unnecessary instructions, minimizing memory access latency.

2. Reducing Resource Consumption :-
- Decreasing the usage of CPU, memory and other system resources to make the program more efficient and less power consuming.

Page No.
Date

Step 1 :- Grammer

S → S+S
S → S×S
S → id

Step 2 :- "id + id +id"

Step 3 :- Shift reduce parsing steps.

Stack	Input	Action
$	id + id + id	Shift id
$ id	+id + id	Shift +
$ id +	id + id	Shift id
$ id+id	+id	reduce S → id
$ id +S	+id	reduce S → id
$ S+S	+id	reduce S → S+S
$ S	+id	Shift +
$ S+	id	Shift id
$ S+id	id	reduce S → id
$ S+S	id	reduce S → id
$ S	id	reduce S → S+S
$ S	$	Accept

Q.3.A Draw and explain how analysis of source program is done

→

A compiler is a software program that converts the high-level source code in a program language into low-level machine