NAME: DISHA JAIN
ENo. : A23052221375
Class/Sec: 6CSE4X.

ASSIGNMENT 2

Q. Explain the different phases of a compiler, showing the output of each phase, using the example of the following statement : position = initial + rate *60.

1.) Lexical Analysis (Scanner):

- Identifier : Position
- Assignment Operator : =
- Identifier : Initial
- Plus Operator : +
- Identifier : rate
- Multiplication : *
- Constant : 60

2.) Syntax Analysis (Parser):

Output : Abstract Syntax Tree (AST)

Identifier +
/ \
Position Identifier
/
Initial *
/
rate 60

3.) Semantic Analysis :

→ Output : Checked for semantic errors
→ No errors found in this example.

4.) Intermediate Code Generation :

- Output : Intermediate Code

Example Intermediate Code.

```
t1 = rate * 60
position = initial + t1
```

3.) Code Optimization

- Output : Optimized Intermediate Code

```
Position = initial + (rate * 60)
```

4.) Code Generation :

Output : Machine Code or Intermediate Code

```
LOAD rate, R1
MUL 60, R1, R2
ADD initial, R2, Position
```

5.) Code Optimization (Machine Code) :

- Output : Optimized Machine Code

```
LOAD rate, R1
MUL 60, R1, Position
ADD initial, Position
```

6.) Code Generation (Machine Code) :

Output : Executable Machine Code

Ex: (binary representation for a specific architecture.

Q. What is ambiguous grammar ? Eliminate ambiguities for the grammar :

$E \rightarrow E+$
$E \mid E*E \mid (E) \mid id.$

Ambiguous grammar is a context - free grammar that produces more than one parse tree for some strings in the language defined by the grammar.

could lead to confusion in parsing, as it creates uncertainty about the correct interpretation of a given input.

E → E + E | E * E | (E) | id.

It is ambiguous because it allows multiple parse trees for certain expressions. To eliminate ambiguity, you can introduce precedence and associativity rules.

E → E + T | T

T → T * F | F

F → (E) | id.

The modification ensures that multiplication has a higher precedence than addition, and it also resolves the left associativity of both operators.

Q. Explain the input buffer scheme for scanning the source program. How the use of sentinels can improve its performance?

Input buffer scheme for scanning involves reading characters from the source program into buffer to be processed by the lexical analyzer. The input buffer helps in efficiently handling input characters and improving the speed of lexical analysis. The basic idea of the input buffer scheme is to read a chunk of characters from the source program into a buffer. The lexical analyzer then processes these characters to identify token.

A sentinel is a special character appended to the

end of the buffer to mark its boundary. When the lexical analyzer reaches the sentinel, it knows that the buffer is about to be exhausted.

Advantages of Sentinels:

1.) Reduced Buffer Overhead: Sentinels help avoid the need for additional checks to determine the buffer's end.

2.) Simplified Buffer Management: Using sentinels simplifies the logic for managing the input buffer.

3.) Improved Efficiency: The use of sentinels reduces the no. of conditional checks during lexical analysis, leading to a more efficient scanning process.

Q. What is LL(1) Grammar? Test whether the following grammar is LL(1) or not.

S: AaAb | BbBa

A: ∈

B: ∈

An LL(1) grammar is a type of context free grammar that can be parsed using a recursive descent parsing method without backtracking, where LL stands for Left to right scan of the input, Leftmost derivation, and 1 symbol of lookahead.

1.) For any pair of production for a non terminal A, $A \to \alpha | \beta$, the sets of first(α) and first(β) must be disjoint.

5) If $A \to \varepsilon$ is a production, then first (B) should not intersect with follow (A), where B is any string of grammar symbols.

$S \to AaAb \mid BbBa$

$A \to \varepsilon$

$B \to \varepsilon$

First Sets:

→ First $(AaAb) = \{a\}$

→ First $(BbBa) = \{b\}$

→ First $(\varepsilon) = \{\varepsilon\}$

Follow Sets:

→ Follow $(S) = \{\$, a, b\}$

→ " $(A) = \{a, b\}$

→ " $(B) = \{a, b\}$

The given grammar is LL(1) since it satisfies the LL(1) conditions. It has disjoint first sets for the alternatives of each non-terminal and follows the necessary rules for LL(1) parsing.

Q. Define tokens, Patterns and lexemes.

1) Tokens: A token is the smallest unit of a programming language that the compiler recognizes. It represents a category of lexical elements in the source code. Eg:- ('if', 'while'). — Keywords, (variable name) — ~~constants, (operators~~ identifiers, etc.

2.) Patterns : It is a description of the form that tokens in a certain category must take. For eg: a pattern for an identifier might be '[a-zA-Z][a-zA-Z0-9]*'.

3.) Lexemes : It is a sequence of characters in the source code that matches the pattern for a particular token.

For eg :- in the statement 'int x = 42;', the lexemes include 'int' (an identifier), 'x' (another identifier). The lexemes are identified based on the patterns specified for the corresponding tokens.

Q. Give the diagrammatic representation of a language processing system.

b. What are the reasons for seperating the analysis phase of compiling into Lexical analysis and parsing?

A language processing system typically involves several phases. Here is a simplied diagrammatic representation:

Source Code

↓

Lexical Analysis → Tokens

↓

Syntax Analysis (Parsing)

↓

Semantic Analysis

↓

Optimization

↓

Code Generation

Optimization

↓

Final Executable

Reasons for Seperating Lexical Analysis and Parsing

1.) Simplifies Design : Seperating lexical analysis and parsing allows for a cleaner and more modular design of the compiler.

2.) Reusability : The separation enables the reuse of lexical analyzers and parsers.

3.) Flexibility : It allows changes to the lexical structure without affecting the parsing phase, and vice versa.

4.) Parallel Processing :- Lexical is often less complex and can be parallelized more easily than parsing.

Q. Explain in detail about the role of Lexical analyzer with the possible error recovery actions?

The primary role of lexical analyzer is to recognize the basic building blocks of a programming language. The lexical analyzer operates on the character stream of the source code and generates a stream of tokens.

Error recovery actions :-

1.) Skip and Continue : If a lexical error is detected, the lexer skips the problematic portion of the input and continues scanning.

2.) Insertion or deletion : The lexer may insert or delete characters to synchronize with the expected token structure.

3.) Token substition : The lexer may replace an erroneous token with the analysis.

4) Flagging Errors: The lexer can flag errors and continue processing, reporting all detected errors after completing the lexical analysis phase.

Q. What are the algebraic properties of regular expression? What is Left Recursion? Give an example for eliminating the same.

Properties :-

1) Concatenation (Associative): If r, s and t are regular expressions, then $(r.s).t = r.(s.t)$.

2) Union (Associative and Commutative): If r, s and t are regular expressions, then $(r+s)+t = r+(s+t)$ (associative) and $r+s = s+r$ (commutative). Union is both associative and commutative.

3) Concatenation with the Empty String: For any regular expression r, $r.\varepsilon = r = \varepsilon.r$, where ε represents the empty string.

4) Concatenation with the Empty Set: For any regular expression r, $r.\emptyset = \emptyset = \emptyset.r$, where $\emptyset$ represents the empty set.

5) Distributive Laws: For regular expressions r, s, and t, $r.(s+t) = r.s + r.t$ and $(r+s).t = r.t + s.t$

Left Recursion: It occurs in a grammar when a non-terminal A produces a string that begins with itself. In other words, A directly or indirectly derives a production that starts with A.

Eg :-

$A \rightarrow A+B \mid B$

$B \rightarrow num$

Eliminating Left Recursion:

$A \rightarrow BA'$

$A' \rightarrow +BA' \mid \varepsilon$

$B \rightarrow num$

Q. What is first and follow? Explain in detail with an example. Write down the necessary algorithm.

These sets are used in the analysis of context-free grammars. They help determine the possible initial and subsequent symbols of a production, respectively, and are crucial in constructing predictive parsing tables.

1.) First Set: It is the set of terminals that can begin the strings derivable from that symbol.

Notation: First(X), where X is the non-terminal or terminal symbol.

Eg :- $A \rightarrow BC$

The first set of A is the set of terminals that can appear immediately to the right

2.) Follow Set: It is the set of terminals that can appear immediately to the right of that non-terminal in some derivation.

Notation: Follow(X), where X is a non-terminal symbol.

Eg: If A is a non-terminal and there is a production $B \rightarrow \alpha A \beta$

then First(β) (excluding ε) are in Follow(A).

Algorithms:

First Set :-

- For each production $A \to \alpha$:
  - If α is a terminal, add α to First(A).
  - If α is a non-terminal, add First(α) to First(A).
  - If α can derive ϵ, add ϵ to First(A) (and continue checking for next symbol).

2.) Follow Set :-

- Initialize Follow(S) to {$} where S is the start symbol.
- For each production $A \to \beta B \gamma$:
  - Add First(γ) \ {ϵ} to Follow(B).
  - If ϵ is in First(γ), add Follow(A) to Follow(B).
  - If B is at the end of a production or $\beta B \gamma$ and ϵ is in First(γ), add Follow(A) to Follow(B).

Q. Construct Predictive Parsing table for the following grammar:

$S \to (L) / a$

$L \to L, S / S$

and check whether the following sentences belong to that grammar or not.

(i.) (a,a)

(ii.) (a,(a,a))

(iii.) (a,((a,a),(a,a)))

From the given grammar:

$S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

First and Follow Sets:

1) First (S) = {(, a}
2) First (L) = {(, a} (since both L and S can derive ϵ)
3) Follow (S) = {), $}
4) Follow (L) = {), $, ,}

Predictive Parsing Table:

	(	)	a	,	$
S	(L)		a		
L	L, S		S		

i) (a, a) → Belongs to grammar

ii) (a, (a, a)) → ! ! ! . ! !

iii) (a, ((a, a), (a, a))) → ! ! . . .

Q. Write YACC specification for arithmetic expression. Explain in brief about YACC.

- It is a tool used in the construction of compilers. It generates parsers for CFGs specified in BNF.
- YACC takes a set of production rules and generates C code for parser.

- The %{ ... %} section allows including C code within the YACC file.
- %token NUMBER declares a token type.
- 'expression' rule specifies how expressions can be built using operators '+', '-', '*', and '/'.
- The 'term' rule defines the basic building block of an expression, which can be a no. or an expression.
- The 'yylex' function is responsible for lexical analysis. It returns the token value or 0 when the end of the input is reached.
- The 'yyerror' function is called when an error occurs during parsing. It prints an error message.
- Main Function: It calls 'yyparse' to start the parsing process.

Q. What is a shift reduce parser? Explain the conflicts in SR Parser?

Shift reduce parser is a type of bottom-up parser that performs a series of shift and reduce operations to match the input string against the production rules of a grammar. It builds a parse tree from the leaves (tokens) to the roots (start symbol) by shifting input symbols onto a stack and reducing them to non-terminals when a match with a production rule is found. This process continues until the entire input is reduced to the start symbol.

Reduce Operation: Replaces a sequence of sequence of symbols on the top of the stack with a non-terminal, based on a production rule.

Conflicts in SR Parser:

1.) Shift-Reduce:- It occurs when the parser faces a choice between shifting the next input symbol onto the stack or reducing the symbols on the top.

2.) Reduce-Reduce Conflict: It occurs when the parser has multiple choices for reducing the symbols on top of the stack to different non-terminals..

Q. Discuss differences between top down and bottom up Parsing techniques?

Top Down	Bottom Up.
→ The goal is to find a leftmost derivation of the input string.	The goal is to find a rightmost derivation of the input string.
→ May exhibit backtracking which can lead to inefficiencies especially in cases of left recursion or ambiguity.	Generally more efficient and can handle a broader class of grammars, including left-recursive grammars.
→ The parser is called a recursive descent parser or a predictive parser.	The parser is called a shift-reduce parser.
→ May require additional disambiguation rules or adjustments to the grammar.	Can handle a broader range of grammars, including ambiguous ones, without modification.

Eg:- Recursive Descent Parsing, LL Parsing.

Eg: LR Parsing, SLR, LALR.

Q. Discuss the commonly occuring compiler errors? Mention the differences between compiler and interpreter?

1) Syntax Errors: Violations of the language's syntax rules. The compiler can't understand the source code.

2) Semantic Errors:- Error in meaning or logic of the program. The code may compile, but it won't produce the intended result.

3) Type errors: Mismatches between data types. Operations or assignments may involve incompatible types.

4) Declaration Errors:- Issues related to variable or function declarations.

5) Runtime Errors: Occur during program execution and may result

Differences between Compiler and Interpreter.

Compiler	Interpreter
Translates the entire source code into machine code or an intermediate code before execution. The resulting executable file is independent of the source code.	Translates and executes the source code line by line. No separate compilation step is involved.

→ Produces an executable file that can be run independently.

→ Generally produces more efficient code in terms of memory usage.

→ Requires a separate compilation phase before execution

Eg :- GCC (GNU), Microsoft Visual C++ Compiler.

Executes the code directly, producing output during runtime.

May consume more memory as the source code needs to be kept in memory during execution.

No separate compilation phase.

Python, Ruby, JavaScript.