RIPHAH INTERNATIONAL UNIVERSITY,

LAHORE CAMPUS

Riphah School of Computing and Innovation BS-CS

8th (Spring -2023)

ASSIGNMENT – I

Course: Compiler Construction

Total Marks: 30

NAME:	Muhmmad Sameer Sohail
SAP:	15000

Q # 1

a) What is meant by compiler? How it is different from other translators?

A compiler is a sort of software program that converts high-level programming languages into machine code that the CPU of a computer can comprehend and execute. Lexical analysis, syntax analysis, semantic analysis, code optimization, and code creation are some of the processes that make up the compilation process. An executable program that can be run on a computer is the outcome**.**

- Speed: Because the entire program is converted into machine code in advance rather than on-the-fly as it is executed, compiled programs often run quicker than interpreted programs.
- A program may be run on any computer that has the required hardware architecture and operating system after it has been compiled into machine code.
- Security: Because the source code is not present in the executable file, compiled programs are often more secure than interpreted ones.

However, compilers also have some disadvantages compared to other types of translators.

- Compilers, for instance, need numerous steps of analysis and optimization, making them more sophisticated than interpreters and assemblers.
- Because the machine code produced by the compiler is not understandable by humans, debugging compiled programmers can be more challenging than debugging interpreted ones.
- Compiler development can require a lot of time and resources, which might make it more challenging for new programming languages to become popular.

Compilers have a number of benefits over alternative language translators, such as interpreters and assemblers. As an illustration:

b) Explain the different phases of compiler.

A compiler is a piece of software that converts high-level programming languages into executable code that the CPU of a computer can use. Compilation is often broken down into a few stages, each of which completes a particular task to convert the input program into a machine-readable format. Following are the phases of a compiler:

- **Lexical analysis:** It is often known as scanning, is the initial stage of compilation. Character by character, the input program is read, and tokens, such as keywords, identifiers, operators, and literals, are recognised as groupings of characters based on the specified rules. To be used in later stages, the tokens are kept in a data structure called a symbol table.

- **Syntax Analysis:** Syntax analysis, commonly referred to as parsing, is the second stage of compilation. The token stream produced by the lexical analysis step is examined using a programming language-specified grammar. The rules for statements, expressions, and program structure are all defined by the grammar, which also specifies the syntax and structure of the language. An error is produced if the grammar cannot be applied to the token stream.

- **Semantic Analysis**: Semantic analysis is the third stage of compilation. This step examines the input program for any semantic issues, such as type mismatches, undefined variables, and inaccessible code, that the syntax analysis phase was unable to find. An abstract syntax tree (AST), which depicts the program's structure in a way that can be utilised for following stages, is also produced during the semantic analysis phase.

- **Optimization:** The compilation process's fourth stage is optimization. The performance and efficacy of the produced code are enhanced at this step using a variety of ways. Instruction scheduling, loop unrolling, and constant folding are examples of optimization strategies. The optimization step can be carried out at many levels of granularity, such as the program-wide level or the fundamental block level.

- **Code Generation:** Code generation is the last step in the compilation process. It is during this stage that the target CPU's executable machine code is produced. Assembler language or binary code are examples of low-level code that is produced during the code generation phase using the AST created

during the semantic analysis phase. In most cases, the code is produced in an object file format, which may be linked with other object files to produce an executable program.

Q # 2

a) Explain the Symbol Table and Reserved Word Table.

b) Also explain their roles in each phase of compiler.

Compilers maintain track of the symbols, such as variables, functions, and classes, used in a programme using a data structure called a symbol table. The scope, data type, name, and position of each symbol are all stored in the symbol table. The symbol table is normally filled out during the lexical analysis and syntactic analysis stages of the compiler. It is then utilised in the semantic analysis, optimisation, and code generation stages.

Identifiers and other symbols that are found in the input program are stored in the symbol table during the lexical analysis stage. To find out if an identifier has previously been defined in the current scope or an outside scope, the lexical analyser consults the symbol table. The lexical analyser adds the identifier to the symbol table if it hasn't already been declared.

Reserved words:

Compilers employ a data structure called a reserved word table to keep track of the reserved words, including if, while, and return, that are used in programming languages. It is used to identify reserved words and to separate them from identifiers. The reserved word table is normally filled up during the lexical analysis phase of the compiler. The reserved word table is utilised during the lexical analysis stage to determine if a token is a reserved word or an identifier. To see if a token corresponds to a reserved word, the lexical analyser consults the reserved word table. The token is identified as a reserved word and its matching token code is returned if it matches a reserved word.

Q # 3

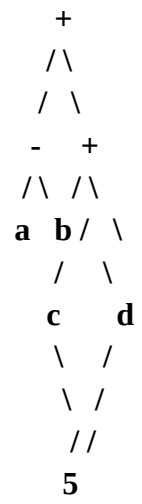
a) Explain the Pass the given statement from all the phases of compiler and mention output of each phase.

Result = (a – b) + (c / d) + 5

The following steps would be required to pass the provided statement through all the stages of a typical compiler:

Lexical analysis: Tokens from the input program, such as identifiers, operators, and literals, are separated out into several categories. The tokens would be "a", "-", "b", "+", "(", "c", "/", "d", ")", "+," and "5" in this instance. A stream of tokens is the phase's output.

Syntax analysis: The flow of tokens is examined to see whether the syntax of the programming language is being followed. An abstract syntax tree, which depicts the structure of the program in a tree-like fashion, is the result of this step. The AST for the provided statement would seem as follows:



Semantic analysis: The AST is examined to make sure that it is semantically sound, which means that the expressions' types and values are consistent. This phase produces a symbol table that contains details on the identifiers used in the programme, including their types and scopes.

Intermediate Code Generation: The AST is converted into an intermediate representation, which is typically a low-level language or bytecode. The output of this phase is a sequence of intermediate code instructions. For example, the intermediate code for the given statement might look like this:

t1 = a - b
t2 = c / d
t3 = t1 + t2
t4 = t3 + 5

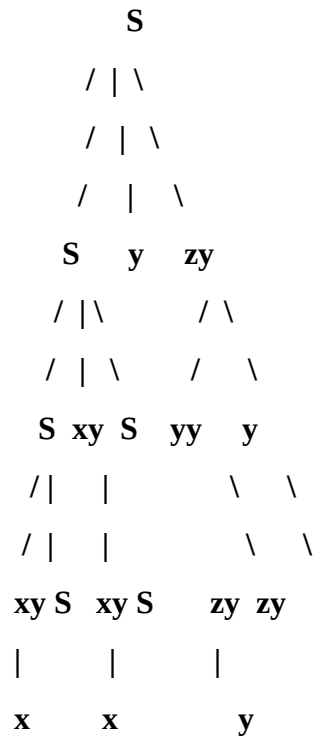
Optimization: To increase the efficiency of the programme, the intermediate code is optimised. Redundant code may be removed, and complex statements may be made simpler. This stage results in optimised intermediate code.

Code generation: The optimised intermediate code is converted to machine code or bytecode for the target platform. The executable programme that may be used on the target platform is the phase's output.

b) Let

S→ xyS
S→ yzS
S→ yyS
S→ x
S→ zy

Construct a parse tree for the following string "xyxyxyyzyyzy



Q # 4

a) What is meant by parsing? Explain top down and bottom up parsing with examples.

To determine a given input sequence of tokens or characters' grammatical structure, a process known as parsing involves analyzing them in accordance with a grammar's rules. It calls for the usage of a parser, which may be either a top-down or bottom-up parser.

The method of parsing known as "top-down parsing" entails the parser beginning at the parse tree's root and working its way down to the leaves. It generates the input

string by applying the production rules after the start symbol. Recursive Descent Parsing, LL(1) Parsing, and LL(k) Parsing are the three most popular top-down parsing algorithms.

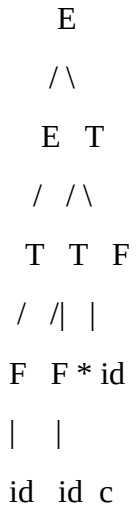
Example:
Consider the following grammar:

E → E + T | T

T → T * F | F

F → (E) | id

Top Down Parsing:



Bottom-up parsing, on the other hand, is a type of parsing where the parser starts from the leaves of the parse tree and works its way up to the root. It starts with the input string and applies the production rules in reverse to generate the start symbol. The most commonly used bottom-up parsing algorithms are Shift-Reduce Parsing and LR(k) Parsing.

Example:

begin by inserting the input string aabbb and an empty stack. From left to right, we scan the input string, adding each symbol as a new terminal to the stack. We determine if the right-hand side of any production in the grammar matches the contents of the current stack at each stage. If they do, we apply the corresponding production and swap the matching non-terminal for the matched symbols on the stack.

I. Push a onto the stack: a
II. Push a onto the stack: aa
III. Match aSb on the right-hand side of the grammar, replace with S: S
IV. Push b onto the stack: Sb
V. Push b onto the stack: Sbb

VI. Reduce Sb to S using the production S → aSb: Sa

VII. Reduce Sa to S using the production S → aSb: S

VIII. Match ε on the right-hand side of the grammar, replace with S: S

b) Differentiate ambiguous and unambiguous grammar. When does a context-free grammar is ambiguous?

Criteria	Ambiguous Grammar	Unambiguous Grammar
Definition	A grammar that can generate a string with multiple parse trees	A grammar that can generate a string with only one unique parse tree
Number of parse trees	Multiple parse trees possible for at least one string	Only one parse tree possible for all strings
Consequences	Parsing may not be deterministic and can result in incorrect interpretations	Parsing is always deterministic and produces unique interpretations
Example	`S → SS	(S)

Context-free grammar is ambiguous if it can generate a string with multiple parse trees. In other words, there are multiple ways to derive the same string from the grammar.