North South University

HW1

Submitted by:

Arka Karmoker

ID: 2112343642

Course: CSE425

Section: 1

Submitted to:

Dr. Md Shahriar Karim [MSK1]

Department of Electrical and Computer Engineering (ECE)

North South University

Date of submission: 22/03/2025

Homeworks 1

Problem 1: What do you understand by imperative programming programming languages? How is it related to Von Neumann architecture? Draw a schematic of von Neumann architecture and explain.

~~Solution~~:

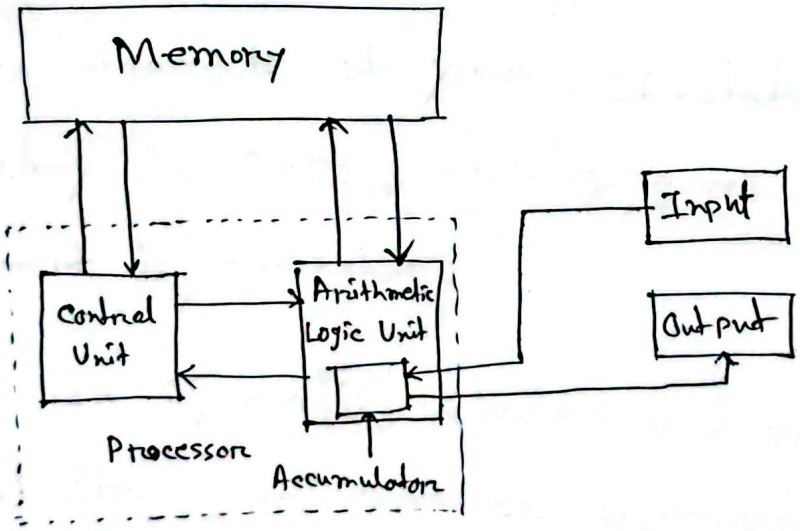
Answer: Imperative programming is a paradigm ~~that~~ where programs are written as a sequence of statements or commands that change a program's state to achieve a desired outcome. It focuses on how to solve a problem by explicitly telling the computer what steps to take, such as assigning values to variables or using loops and conditionals.

This relates to the von Neumann architecture because this architecture is designed to execute sequential instructions stored in memory, which aligns with the step-by-step nature of imperative programming.

The architecture consists of:

- CPU (Central Processing Unit): Executes instructions (arithmetic, logic, control).
- Memory: Stores both data and instructions.
- Input/Output: Interfaces with the outside world.
- Bus: Connects CPU, memory, and I/O for data transfer.

schematic:



The Von Neumann architecture ~~sh~~ schematic illustrates a computer design with a central CPU, containing a Control Unit and ALU, connected via a ~~sigle~~ ~~b~~ single bus to a unified Memory unit storing

both data and instructions. Input and output units link to the bus, enabling data entry and result display. ~~Alto~~ Arrows along the bus depict the bidirectional flow of instructions and data between components, highlighting the sequential processing nature of this foundational model.

Problem 2: For C-programming, discuss the following with an example of yours: If-statement is not mandatory in C-programming, given that while control statement is available.

Answer: In C, an `if` statement isn't mandatory because control flow can often be managed with other constructs like `while`. For example, an `if` condition can sometimes be rewritten as a `while` loop that executes once (or not at all).

Example:

```
#include <stdio.h>
int main() {
    int x = 5;
    // Using if
    if (x>0) {
        printf("Positive\n");
    }

    // Equivalent using while
    while (x>0) {
        printf("Positive\n");
        break; // Ensures it runs only once
    }
    return 0;
}
```

Here, the "while" loop with a "break" copy the "if" statement's single execution, showing that 'if' isn't strictly necessary when "while" is available.

Problem 3: Discuss the following criteria of efficient language design criteria with example of each. select a programming language of your preference and evaluate it according to these criteria.

- Generality
- Uniformity
- Extensibility
- Restrictability

Answer:

I'll evaluate these criteria using Python as the example language.

1. Generality: A language should support a wide range of applications. Python's generality is evident in its use for web development, data science, and scripting. For example, Library like NumPy extend its mathematical ~~capabilit~~ capabilities.

2. Externality: The language should integrate well with external systems. Python excels here with tools like `os` and `sys` modules for interacting with the operating system.

3. Uniformity : Syntax and semantics should be consistent. Python's readable, indentation-based syntax (e.g., if x>0:) is uniform, though exceptions like lambda functions can feel less consistent.

4. Restrictability: The language should allow limiting features for safety or simplicity. Python supports this weakly. For example, we can't easily restrict dynamic typing, which may lead to runtime errors.

Python scores high on generality and extensibility, moderately on uniformity, and lower on restrictability due to its flexibility.

Problem 4: Discuss (in brief) the types of errors that generally occur in a program. Provide an example pseudocode for each type of error.

Answer:

1. Syntax Error: Incorrect syntax that prevents compilation.

- Pseudocode: `if x>5 print x` (missing colon or parentheses).

2. semantic Error: code runs but produces incorrect results.

- Pseudocode: `x=5 ; y=x/0` (division by zero).

3. Logic Error: Flawed reasoning in the algorithm.

- Pseudocode: `sum=0 ; for i=1 to 3 : sum=i`

(overwrites instead of adding, result is 3 instead of 6).

~~Problem~~

Problem 5 : Write short notes on the followings

(a) Why are compilers separated into front-end and back-end?

(b) Von Neumann bottleneck

(c) What roles do symbol table have in compilers?

(d) Portability of programming languages.

Answers:

(a) Compilers :

Front-end vs Back-end : The front-end parses source code (syntax, semantics) and generates an intermediate representation, while the back-end optimizes it and produces machine code. Seperation allows portability across platforms.

(b) Von Neumann Bottleneck : The shared bus between CPU and memory limits speed, as data and instructions compete for access, slowing execution.

(c) Symbol Table in Compilers: A data structure tracking identifiers (variables, functions) and their attributes (type, scope), aiding in semantic analysis and code generation.

(d) Portability: A language's ability to run on different platforms. High-level languages like Java use virtual machines (JVM) for portability, unlike low-level Assembly.

Problem 6: Scoping strategies and C program

Answer: I would prefer lexical (static) scoping for clarity - variables are resolved based on their location in the source code, not runtime stack.

C program Analysis :

```
#include <stdio.h>
int i=6;
int main (int argc, char *argv[]){
    printf("%d\n", i); //6 (global i)
    int i=7, j=8; // New local scope shadows global i
    printf("%d\n", i, j); //7 (local i, j ignored in single %d)
   int i=5; // Newer local i shadows previous i
   printf("%d\n", i, j); //5 (newest i, j ignored)
  printf("%d\n", i); //5 (newest i)
  print f("%d\n", i); //5 (newest i)
  return 0;
}
```

Output: `6, 5, 5, 5, 5`

C uses block-level scoping; each `int i` declaration shadows the previous one within its block.

Problem 7: Programming Languages .....

Answer:

1. Short-code: Early high-level language (1949).

- Advantage: Simplified programming over machine code.
- Limitation: Manually translated, slow.

2. Assembly: Mnemonics for machine code.

- Advantage: Direct hardware control, fast.
- Limitation: Non-portable, complex.

3. Fortan: First high-level language (1957).

- Advantage: Scientific computation ease.
- Limitation: Limited control structures initially.

4. ALGOL 58-60: Structured Programming Pioneer.

- Advantage: Influenced modern syntax (blocks, recursion).
- Limitation: No standard I/O, faded use.

5. C/C++ : Low-level control with high-level features.

- Advantage : Performance, system programming (1970s).
- Limitation : Manual memory management risks.

6. Java : Platform-independent (1995).

- Advantage : "Write once, run anywhere" via JVM.
- Limitation : slower than native code.

7. Python : Readable, versatile (1991).

- Advantage : Rapid development, beginner-friendly.
- Limitation : slow execution for computation.

8. LISP / Scheme : Functional programming (1958).

- Advantage : symbolic processing, AI focus.
- Limitation : steep learning curve, niche use.

Homework 2

Q1:

1. Lexemes: A ~~lexemed~~ lexeme is a sequence of characters in the source code that forms a basic unit recognized by the lexical analyzer. It's the raw input before categorization.

Example: In the code `int x=5;`, the lexemes are `int`, `x`, `=`, `5`, and `;`.

2. Token: A token is a categorized unit of lexemes, typically assigned a type by the lexical analyzer (e.g., identifier, keyword, operator).

Example: For the lexeme `int`, the token might be <keyword, int>. For x, it might be <identifier, x>.

3. Reserved words: These are predefined words in a programming language that have special meanings and cannot be used as identifiers.

Example: In C, `if`, `while`, and `return` are reserved words

4. Metalanguage: A language used to describe another language, often used to define syntax rules (eg., BNF or EBNF).

Example: BNF (Backus-Naur Form) is a metalanguage used to define the syntax of programming languages like <stmt> ::= if <expr> then <stmt>.

5. Context-Free Grammer (CFG): A formal grammer where the left-hand side of each production rule is a single non-terminal, and the right-hand side can be any combination of terminals and non-terminals, independent of context.

Example: <S> → a <S> b | ε defines a CFG for strings like ab, aabb.

6. Derivation: The process of generating a string in a language by repeatedly applying production rules of a grammar, starting from the start symbol.

Example: For $\langle S \rangle \rightarrow a\langle S \rangle b \mid \varepsilon$, the derivation of aab is: $\langle S \rangle \Rightarrow a\langle S \rangle b \Rightarrow aa\langle S \rangle bb \Rightarrow aabb$.

7. Production Rule: A rule in a grammar that specifies how a non-terminal can be replaced with a string of terminals and/or non-terminals.

Example: $\langle A \rangle \rightarrow b \mid ab$ is a production rule where $\langle A \rangle$ can be replaced by: b or ab.

Q2:

Advantages of EBNF over BNF:

1. Compactness: EBNF uses repetition {}, optional [], and choice () notations, reducing the number of rules.

2. Readability: EBNF is more intuitive and easier to understand due to its concise syntax.

3. Expressiveness: EBNF can directly express repetition and optional elements without recursion.

Example BNF Grammar:

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<number> ::= <digit> | <digit> <number>

This defines a number as one or more digits.

Transformed to EBNF:

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<number> ::= <digit> {<digit>}

- {<digit>} means zero or more repetitions of <digit> after the first <digit>, making it more concise than the recursive BNF form.

Q3:

Given Grammer:

<S> → <A> a <B> b

<A> → <A> b | b

<B> → b

String 1: babb

<S> ⇒ <A> a <B> b (applying <S> rule)

⇒ b a <B> b (applying <A> → b)

⇒ b a b b (<B> → b)

Derivable: Yes

String 2: bbbab

$\langle S \rangle \Rightarrow \langle A \rangle a \langle B \rangle b$

$\Rightarrow \langle A \rangle ba \langle B \rangle b \quad (\langle A \rangle \rightarrow \langle A \rangle b)$

$\Rightarrow bba \langle B \rangle b \quad [\langle A \rangle \rightarrow b]$

$\Rightarrow bbabb \quad [\langle B \rangle \rightarrow b]$

So, it is not derivable.

String 3: bbaaaabc

$\langle S \rangle \Rightarrow \langle A \rangle a \langle B \rangle b$

$\Rightarrow \langle A \rangle ba \langle B \rangle b \quad [\langle A \rangle \rightarrow \langle A \rangle b]$

$\Rightarrow bba \langle B \rangle b \quad [\langle A \rangle \rightarrow b]$

$\Rightarrow bbabb \quad [\langle B \rangle \rightarrow b]$

So, it is not derivable.

String 4: aaaaaa

$\langle S \rangle \Rightarrow \langle A \rangle a \langle B \rangle b$

$\Rightarrow ba \langle B \rangle b \quad [\langle A \rangle \rightarrow b]$

$\Rightarrow babb$

So, it is not derivable.

Q4:

Grammer:

$\langle S \rangle \rightarrow a \langle B \rangle b$

$\langle B \rangle \rightarrow a \langle B \rangle b \mid b$

This ensures at least one [a] (from [<S>]) and two [b] (one from <S>, one from <B>), with <B> adding matching [a] and [b] pairs.

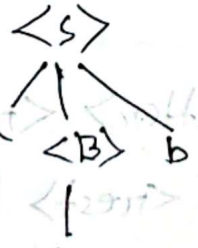
Derivation for abb:

$\langle S \rangle \Rightarrow a \langle B \rangle b$

$\Rightarrow abb \quad [\langle B \rangle \rightarrow b]$

Matches (n=1): 1 a, 2 b

Parse Tree for abb:



Derivation for aabbb :

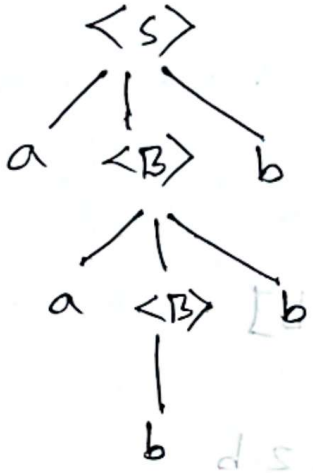
$\langle S \rangle \Rightarrow a\langle B \rangle b$

$\Rightarrow aa\langle B \rangle bb \quad [\langle B \rangle \rightarrow a\langle B \rangle b]$

$\Rightarrow aabbb \quad [\langle B \rangle \rightarrow b]$

Matches ($n=2$) : 2a, 3b

Parse Tree for aabbb :



Q5:

Grammar:

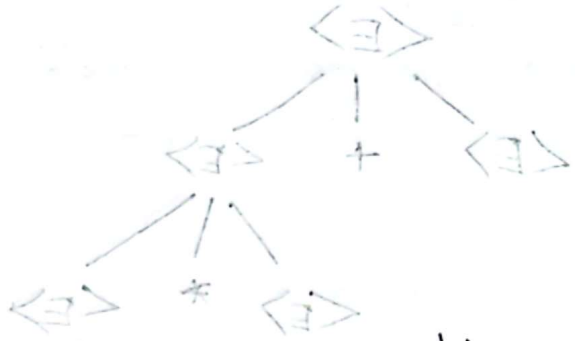
<identifier> → <letter> <rest>

<rest> → <letter> <rest> | <digit> <rest> | ε

<letter> → a | b | c | ---- | z | A | B | C | ---- | Z

<digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<identifier> starts with a <letter>, followed by zero or more <letter> or <digit> symbols via <rest>.



Q6:

Ambiguity: A grammar is ambiguous if a string in its language can be derived in more than one way, producing distinct parse trees.

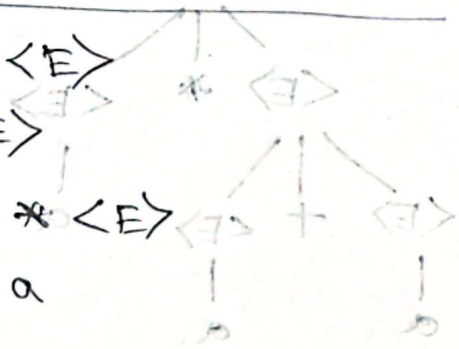
Example grammar:

<E> → <E> + <E> | <E> * <E> | a

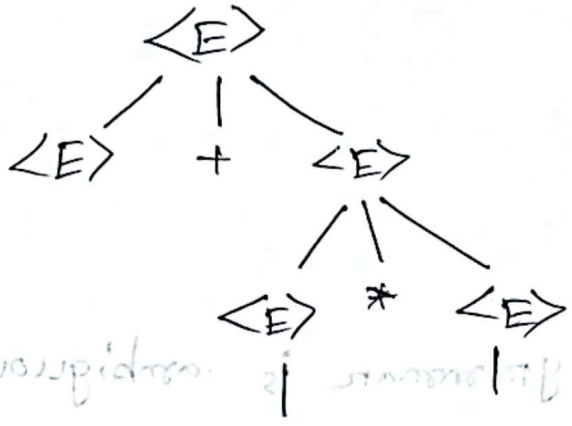
String: a + a * a

Derivation 1 (Left Associative):

<E> => <E> + <E>
=> a + <E>
=> a + <E> * <E>
=> a + a * a



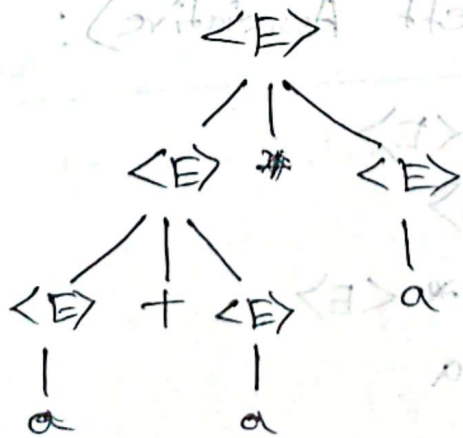
Parse Tree:



Derivation 2 (Right Associative):

$$\langle E \rangle \Rightarrow \langle E \rangle * \langle E \rangle$$
$$\Rightarrow \langle E \rangle + \langle E \rangle * \langle E \rangle$$
$$\Rightarrow a + a * a$$

Parse Tree:



Two distinct trees show the grammar is ambiguous. Typically, * has higher precedence than +, but this grammar doesn't enforce that.

Q7:

Selected statements: if, while, for from C.

1. if statement:

<if_stmt> → if (<expr>) <stmt> | if (<expr>) <stmt> else <stmt>

<expr> → <identifier> | <number> | <expr> <op> <expr>

<stmt> → <identifier> = <expr> ; | <if_stmt> | { <stmt_list> }

<stmt_list> → <stmt> | <stmt> <stmt_list>

Covers basic if and if-else.

2. while statement:

<while_stmt> → while (<expr>) <stmt>

Loops while <expr> is true.

3. for statement:

<for_stmt> → for (<init> ; <expr> ; <update>)
<stmt>

<init> → <identifier> = <expr>

<update> → <identifier> = <expr>

Includes initialization, condition, and update.

These grammars define the syntax of control structures in a simplified form.