Guideline

Homework for Week 02

Please submit your solutions as a single PDF on Brightspace.

The Python code in the exercise is for your testing. Do not submit it.

Multiple submissions are allowed before the due time; the last submission will be graded.

Grading Criteria

- Grading will be *highly strict*, with minimal tolerance for mistakes or misconduct.
- Factual grading errors will be corrected, but partial grading decisions will not be negotiable.
- If the solution appears to be plagiarized or AI-generated, the issue will be reported to the instructor.

Part 1. Regular Expressions (points = 25)

Below are two abbreviations that are widely used in regular expressions:

- The expression `\d` is a shorthand for the regular expression [0-9].
- The quantifier `{n}` is used to match exactly n occurrences of the preceding character class. For example, the regular expression `\d{3}` matches exactly three consecutive digits in a string; the regular expression [a-z]{5} matches five consecutive lowercase letters.

Solve the following questions:

1. What does the regular expression `\d{3}-\d{2}-\d{4}` match?

   a) A phone number in the format (xxx) xxx-xxxx

   b) An email address

   c) A date in the format mm/dd/yyyy

   d) A Social Security number in the format xxx-xx-xxxx

2. What does the regular expression `[a-z]\d*` match?

   a) Any word containing only lowercase letters

   b) Any word containing only uppercase letters

   c) Any word containing only numbers

   d) Any word containing only a lowercase letter followed by optional digits

3. What does the regular expression `\d{3}-\d{3}-\d{4}|\d{10}` match?

   a) A Social Security number in the format xxx-xxx-xxxx or a 10-digit phone number

b) A date or a phone number

c) A phone number or an email address

d) A Social Security number or an email address

4. What does the regular expression `[a-z]+?@[a-zA-Z_]+?.[a-zA-Z]{2,3}` match? Choose the closest answer.

   a) A phone number

   b) An email address

   c) A URL

   d) A street address

5. Please conduct some research. Your task is to determine the Linux command that can recursively search for all markdown files (with the ".md" extension) in the current directory that contains a negative integers. Note: to match the minus symbol, you can use `\-` or `[-]`. The following should be recognized: -89, -1, -007. The following should not be recognized: 0, -x, 42. A good starting point for your investigation might be familiarizing yourself with the `grep` command. You can refer to the Wikipedia page on grep for an overview.

Part 2. Context-free Grammar (points = 25)

6. What is the language generated by the following grammar? `S -> aSb | ε`

   A. The set of all strings that with 'a' and end with `b`.

   B. The set of all strings that contain an equal number of 'a's and 'b's.

   C. The set of all strings that contain an even number of 'a's followed by an even number of 'b's.

   D. The set of all strings that contain n 'a's followed by m 'b's, where n = m >= 0

7. Create a grammar that generates all strings over {a, b} that start and end with the same symbol.

8. Given the grammar with the following productions:

   ```
   S -> aSbb | ε
   ```

   Determine the language generated by the grammar.

9. Given the following grammar

```
E -> E + E | E * E | id
```

Draw different parse trees for the string id + id * id to demonstrate ambiguity.

10. Given the following grammar

```
S -> aAb
A -> c | d
```

Can `acb`, `adb`, `adab`, `aab`, `ab` be parsed? Give an answer for each but you do not need to explain.

Column 1	Can be parsed (true/false)
acb	
adb	
adab	
aab	
ab	

Part 3. Application: Regular Expressions for Lexical Analysis (pionts = 25)

In this exercise, you will fill in regular expressions to perform lexical analysis. Your goal is to tokenize a given input program:

```
if (x == 10) {
    y = x + 5;
    // Increment y
    y = y + 1;
    return y;
} else {
    return -1;
}
```

Your objective is to turn this program into a sequence of tokens:

```
Token(KEYWORD, if)
Token(DELIMITER, ()
Token(IDENTIFIER, x)
Token(OPERATOR, ==)
Token(LITERAL, 10)
Token(DELIMITER, ))
Token(DELIMITER, {)
Token(IDENTIFIER, y)
Token(OPERATOR, =)
Token(IDENTIFIER, x)
Token(OPERATOR, +)
Token(LITERAL, 5)
Token(DELIMITER, ;)
Token(COMMENT, // Increment y)
Token(IDENTIFIER, y)
Token(OPERATOR, =)
Token(IDENTIFIER, y)
Token(OPERATOR, +)
Token(LITERAL, 1)
Token(DELIMITER, ;)
Token(KEYWORD, return)
Token(IDENTIFIER, y)
Token(DELIMITER, ;)
Token(DELIMITER, })
Token(KEYWORD, else)
Token(DELIMITER, {)
Token(KEYWORD, return)
Token(OPERATOR, -)
Token(LITERAL, 1)
Token(DELIMITER, ;)
```

```
Token(DELIMITER, })
Token(EOF, )
```

Task

You will achieve this by filling in the "tofill" parts in the code below:

```
token_specification = [
    ('COMMENT',    r'tofill'),  # Single-line comments
    ('KEYWORD',    r'tofill'),  # Keywords
    ('IDENTIFIER', r'tofill'),  # Identifiers
    ('LITERAL',    r'tofill'),  # Integer and string literals
    ('OPERATOR',   r'tofill'),  # Operators
    ('DELIMITER',  r'[()\{\};]'),  # Delimiters
    ('WHITESPACE', r'\s+'),  # Whitespace (ignored)
    ('MISMATCH',   r'.'),  # Any other character
]
```

The code is copied to `<course_repository>/homeworks/hw02_code/main.py`. You can test your regular expressions in the code, but your answer for this question should be the following table:

Token Type	Regular Expression
COMMENT	
KEYWORD	
IDENTIFIER	
LITERAL	
OPERATOR	

Additional Notes

To make the code work, you may need to do some research yourself. In particular:

- You can use `.` for **matching any single character** except newline. It is useful for catching characters that do not match any of the other specified patterns.
- You can use `\b` for **word boundary** to ensure that patterns match only when they appear as whole words, not as part of other words.

The code provided is for your convenience; you do not need to make it work perfectly, but if it does, you will have peace of mind knowing your regular expressions are correct. In this exercise, only the program above is the test program; you do not need to stress-test your code with other inputs.

Provided Code

```
import re
from enum import Enum, auto

# Define the token types
class TokenType(Enum):
    KEYWORD = auto()
    IDENTIFIER = auto()
    OPERATOR = auto()
    DELIMITER = auto()
    LITERAL = auto()
    COMMENT = auto()
    EOF = auto()  # End of file

# Define the Token class
class Token:
    def __init__(self, type, value):
        self.type = type
        self.value = value

    def __repr__(self):
        return f"Token({self.type.name}, {self.value})"

# Define regular expressions for each token type
token_specification = [
    ('COMMENT',    r'tofill'),  # Single-line comments
    ('KEYWORD',    r'tofill'),  # Keywords
    ('IDENTIFIER', r'tofill'),  # Identifiers
    ('LITERAL',    r'tofill'),  # Integer and string literals
    ('OPERATOR',   r'tofill'),  # Operators
    ('DELIMITER',  r'tofill'),  # Delimiters
    ('WHITESPACE', r'\s+'),  # Whitespace (ignored)
    ('MISMATCH',   r'.'),  # Any other character
]

# Compile the regular expressions into a single pattern
token_regex = re.compile('|'.join(f'(?P<{name}>{pattern})' for name,
pattern in token_specification))

# Implement the lexer function
def tokenize(code):
    tokens = []
    line_num = 1
    line_start = 0
    for mo in token_regex.finditer(code):
        kind = mo.lastgroup
        value = mo.group(kind)
        column = mo.start() - line_start
        if kind == 'WHITESPACE':
            pass  # Skip whitespace
        elif kind == 'MISMATCH':
            raise RuntimeError(f'Unexpected character {value!r} on line
{line_num}')
        else:
```

```
                token_type = TokenType[kind]
                tokens.append(Token(token_type, value))
            # Update line number and start position
            line_breaks = value.count('\n')
            if line_breaks:
                line_num += line_breaks
                line_start = mo.end()
    tokens.append(Token(TokenType.EOF, ''))
    return tokens

# Example usage
if __name__ == "__main__":
    code = '''
    if (x == 10) {
        y = x + 5;
        // Increment y
        y = y + 1;
        return y;
    } else {
        return -1;
    }
    '''
    tokens = tokenize(code)
    for token in tokens:
        print(token)
```

Part 4. Application: Context-Free Grammar Analysis (points = 25)

In this exercise, you will analyze a context-free grammar and produce a parse tree for a given input program.

Context-Free Grammar (CFG)

Consider the following simplified context-free grammar that defines a subset of a programming language:

```
1. Program   -> StatementList
2. StatementList -> Statement StatementList | ε
3. Statement -> 'if' '(' Expression ')' '{' StatementList '}' 'else' '{'
StatementList '}'
              | 'return' Expression ';'
              | Identifier '=' Expression ';'
4. Expression -> Term ExpressionTail | Term ComparisonOp Term
5. ExpressionTail -> '+' Term ExpressionTail | '-' Term ExpressionTail | ε
6. ComparisonOp -> '==' | '!=' | '<' | '>' | '<=' | '>='
7. Term       -> Identifier | Literal
8. Identifier -> [a-zA-Z_][a-zA-Z_0-9]*
9. Literal    -> '-'? [0-9]+
```

Input Program

Given the following input program:

```
if (x == 10) {
    y = x + 5;
    return y;
} else {
    return -1;
}
```

Tasks

1. **Derive the Parse Tree:**

   Your first task is to manually derive the parse tree for the input program using the given context-free grammar. A parse tree visually represents the syntactic structure of the input based on the rules of the grammar. To create the parse tree, you can draw it on paper and scan it, use a diagram tool, or represent it in a text-based format using indentation or parentheses.

2. **Identify the Productions:**

   List all the productions (rules) from the grammar that are used to generate the input program. Following the order of when they are applied. There could be more than one orders.