Compiler Design Project

Semester: FALL 2024

Project Title: Mini Compiler for Scripting Language

Project Description: In this project, you will develop a mini compiler that performs lexical analysis, syntax analysis, and basic semantic analysis for Scripting Language.

Project Phases:

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis

Notes:

• You should identify token classes in a way that will assist with the following phase (parser).

• You can use any programming language you prefer for building the scanner.

• The project deadline will 25/11.

• Each team should have 3 to 4 members.

• A discussion will be with all group members; all members should participate in implementation.

• Implement core functionalities without using built-in libraries for lexical or syntax analysis; build your own methods wherever possible.

• **Very important: any plagiarism detected will lead to losing the project marks**

Scripting Language Specifications:

- Case Sensitivity: Language is case-insensitive.
- Reserved Keywords: All keywords are reserved (shown in uppercase for emphasis).
- Function Calls without Parameters: Parentheses are optional in a CALL statement if no parameters are passed.
- Operators:
    - Arithmetic Operators: Supports +, -, *, and /.
    - Relational Operators: Includes =, >, <, and != (for "not equal to").
    - o Logical Operators: Supports AND, OR, and NOT (written as `and`, `or`, `not` respectively).
- Comments: Enclosed in braces {}.

The Scripting Language has valid types of statements:

- Variable Declaration and Assignment
    - LET Identifier = Expression
- Conditional Statements
    - IF Condition THEN Statement(s) ELSE Statement(s) ENDIF
- Looping Structures
    - WHILE Loop: WHILE Condition DO Statement(s) ENDWHILE
    - FOR Loop: FOR Identifier = Start TO End STEP Increment DO Statement(s) ENDFOR
    - DO-WHILE Loop: DO Statement(s) WHILE Condition
    - Range-based FOR Loop: FOR Identifier IN Range(Start, End, Step) DO Statement(s) ENDFOR
        - Allows iteration over a range of values, especially useful with step values
    - Repeat-Until Loop: REPEAT Statement(s) UNTIL Condition
        - Similar to DO-WHILE, but ends when the condition becomes true.

- Function Definitions and Calls
  - FUNC FunctionName(ParamList) BEGIN Statement(s) RETURN Expression END
  - Function Call: Identifier(ArgList)
- List Operations
  - LET Identifier = [Element1, Element2, ...]
  - Access List Element: Identifier[Index]
- Compound Assignment Operators
  - Identifier += Expression
  - Identifier -= Expression
  - Identifier *= Expression
  - Identifier /= Expression
- Increment and Decrement Statements
  - Identifier++ or Identifier--

Notes when implementing:

Keywords vs. Identifiers:

- Ensure that your lexical analyzer distinguishes between keywords and identifiers, even if they have similar names. For example, an identifier ANDvalue should not be confused with the AND logical operator.
- **Tip**: Implement a clear priority in the tokenizer to recognize keywords before identifying general identifiers.

Comment Handling:

- Implement robust comment handling, where the lexical analyzer skips over everything between { and } until it encounters the closing brace.

- Include error handling for unclosed comments to inform users of syntax errors.

Operator Ambiguities:

- Ensure operators like = and != are consistently recognized, as = could also be a part of the LET assignment syntax.
- Define precedence rules to avoid token conflicts and ensure each character sequence maps to a unique token.

Error Handling and Feedback:

- Develop error-handling routines for token recognition errors, such as invalid characters or incomplete tokens (e.g., an unmatched quote in a string literal).
- **Descriptive Error Messages**: Provide detailed feedback, including the line number and position of the error, to help students debug their code efficiently.

1. Lexical Analysis:

- Implement a lexical analyzer (scanner) that reads the source code character by character.
- Define a set of regular expressions to represent different token types (e.g., keywords, identifiers, literals, operators).
- Develop a token recognition mechanism that maps input patterns to token types.
- Build a symbol table to store information about identifiers encountered during tokenization.

Important Note:

Please note that the provided example is just an example, meaning that your program should be able to tokenize any code written in scripting language not just the provided input of the example.

Input (Source Code)

Code:

```
LET a = 5
LET b = 10
IF a < b
THEN
    LET c = a + b
    LET d = c * 2
ELSE
    LET e = a - b
ENDIF
CALL myFunction(a, b)
```

Output (Tokens and Symbol Table)

Tokens:

Token: let, Lexeme: LET

Token: identifier, Lexeme: a

Token: equal, Lexeme: =

Token: number, Lexeme: 5

Token: let, Lexeme: LET

Token: identifier, Lexeme: b

Token: equal, Lexeme: =

Token: number, Lexeme: 10

Token: if, Lexeme: IF

Token: identifier, Lexeme: a

Token: operator, Lexeme: <

Token: identifier, Lexeme: b

Token: then, Lexeme: THEN

Token: let, Lexeme: LET

Token: identifier, Lexeme: c

Token: equal, Lexeme: =

Token: identifier, Lexeme: a

Token: operator, Lexeme: +

Token: identifier, Lexeme: b

Token: let, Lexeme: LET

Token: identifier, Lexeme: d

Token: equal, Lexeme: =

Token: identifier, Lexeme: c

Token: operator, Lexeme: *

Token: number, Lexeme: 2

Token: else, Lexeme: ELSE

Token: let, Lexeme: LET

Token: identifier, Lexeme: e

Token: equal, Lexeme: =

Token: identifier, Lexeme: a

Token: operator, Lexeme: -

Token: identifier, Lexeme: b

Token: endif, Lexeme: ENDIF

Token: call, Lexeme: CALL

Token: identifier, Lexeme: myFunction

Token: left_paren, Lexeme: (

Token: identifier, Lexeme: a

Token: comma, Lexeme: ,

Token: identifier, Lexeme: b

Token: right_paren, Lexeme:)

Symbol Table:

Name: a, Type: integer

Name: b, Type: integer

Name: c, Type: integer

Name: d, Type: integer

Name: e, Type: integer

Name: myFunction, Type: function (with parameters: a, b)