CS323 Documentation

About 2 pages

1. Problem Statement

The task is to create a compiler-like program that handles symbol table management and generates assembly code for a simplified version of the Rat24F language. The simplified Rat24F language includes basic control flow, variable declarations, and arithmetic operations on integers. The assignment is divided into two parts:

1. **Symbol Table Handling**:
   - Each declared identifier must be added to a symbol table.
   - The symbol table should track each identifier's lexeme, memory location, and type (only integer is supported).
   - The program should ensure that an identifier is not declared more than once and that all variables are declared before use.
2. **Generating Assembly Code**:
   - The program must generate assembly-like instructions for the simplified Rat24F language.
   - The assembly code is based on a virtual machine with a stack and specific instructions such as **PUSHI, PUSHM, POP, STDOUT, and JUMP.**
   - The program should handle basic arithmetic operations (addition, subtraction, etc.) and control flow (loops and conditional statements).
   - The output should include both the generated assembly instructions and the symbol table.

2. How to use your program

To use the program, follow these steps:

1. **Prepare the Source Code**:
   - Save the code in a folder: **(CPSC-323-Assingment-3)**
2. **Run the Program**:
   - Ensure that Python and the required files (lexer.py, parser.py, test_paser.py) along with the needed test cases (test1.txt, test2.txt, test3.txt)are available.
   - Open a terminal or command prompt and navigate to the directory containing the program files.
   - Execute the program using: **(python test_parser.py)**

Output:

- The program will create an output files (e.g., output_test1.txt)

3. Design of your program

The program consists of two main modules: the **lexer** and the **parser**.

Lexer:

The lexer's role is to read the source code and generate tokens that represent the smallest units of the language. It uses regular expressions to identify these tokens and classifies them accordingly.

- **Data Structures**:
    - A **tokens list** that stores recognized tokens and their types.
    - **Current position index** to track where the lexer is in the source code.
- **Main Tasks**:
    - Tokenizing the source code into keywords (e.g., integer, if), identifiers, literals, operators, and separators.
- **Parser:**

The parser processes the tokens produced by the lexer according to the language's grammar rules. It checks for syntactic correctness and generates assembly instructions for each valid statement.

- **Main Tasks**:
    - **Symbol Table Management**: Tracks variable declarations and ensures that each identifier is declared before use. A global variable Memory_address is incremented for each new variable and used to assign unique memory locations.
    - **Assembly Code Generation**: The parser translates parsed statements into assembly-like instructions. For example:
        - Assignments are converted into PUSHI (for integer values) and POPM (for memory locations).
        - Control flow statements (e.g., while, if) are translated into JUMP and LABEL instructions.
        - Input/output statements are translated into STDIN and STDOUT.
- **Data Structures**:
    - **Symbol Table**: A dictionary that stores variable names, their types, and memory locations.
    - **Instruction Table**: A list that holds the generated assembly instructions. Each instruction is associated with an address.
    - **Jump Stack**: A stack to manage control flow during loops and conditionals.
- **Key Algorithms**:
    - **Recursive Descent Parsing**: Used for parsing expressions and statements based on the grammar.
    - **Token Processing**: Each token is processed sequentially, and the parser performs syntax checks and generates the corresponding assembly instructions.

4. **Any Limitation**
    - **Problem**: While the program supports while loops and if statements, its ability to handle complex control flow, such as nested loops or conditionals, could be limited. The way jump addresses are generated (via JUMPZ and JUMP instructions) may become more complex and prone to errors if the structure of control flow becomes more intricate.
    - **Impact**: For more complex programs with deep nesting or additional flow control structures (e.g., switch statements or for loops), the current design might not work well, or the generated assembly might be harder to manage.

5. **Any shortcomings**
   None