**CS323 Documentation**

About 2 pages

1. **Problem Statement**

The task at hand encompasses two primary objectives: symbol table handling and assembly code generation for a simplified version of Rat23F. The modified Rat23F excludes <Function Definitions> and prohibits the use of "real" types.

Symbol Table Handling: It involves organizing every declared identifier into a symbol table and developing procedures for access.

a) Each entry in the symbol table should contain the lexeme and a "memory address" where the identifier is stored. The process involves initializing a global integer variable ("Memory\_address") to 7000 and incrementing it by one for each new identifier added to the table.

b) Essential procedures entail checking for existing identifiers, inserting new ones into the table, and printing all identifiers present. Additionally, the parser must flag errors for undeclared or duplicate identifiers and ensure type compatibility for operations.

Assembly Code Generation: The parser needs modification to comply with the constraints of simplified Rat23F, incorporating code generation to produce assembly instructions. These instructions are to be stored in an array, capable of holding a minimum of 1000 instructions, indexed starting from 1. The resulting array should facilitate label-based jumping within the code. Furthermore, the compiler output should include a comprehensive list of all identifiers used.

This assignment mandates the creation of robust symbol table handling mechanisms, error detection for identifier usage, type matching validation, and the generation of assembly code in adherence to the specified constraints of the simplified Rat23F language.

1. **How to use your program**

Prepare a text file containing the source code written in the simplified Rat23F language. Ensure that the code adheres to the specified guidelines, including the absence of <Function Definitions>, adherence to boolean arithmetic limitations, and strict type matching for arithmetic operations.

Usage: You are able to run the exe file located inside of the zip file, which requires the txt file to be located in the same directory as the exe. The exe will prompt the user to submit the name of the text file, and after doing so, the txt file is parsed and the output is provided in the same directory as the after giving the executable the text file’s name, it will create a output.txt which will have the compiled code with the appropriate symbol table and assembly code.

1. **Design of your program**

The program's main components include symbol table handling and assembly code generation for the simplified Rat23F language. In the symbol table handling part, every declared identifier is managed through procedures that place entries in a symbol table. Each entry in the table stores the lexeme and a "memory address" for the identifier. Procedures check for existing identifiers, insert new ones, and print all entries. The parser ensures error messages for undeclared or redeclared identifiers and checks type matches. For the parser, it is modified to produce assembly code given simplified Rat23F, using new code to produce instructions stored in an array holding at least 1000 assembly instructions. The compiler also generates a listing of all identifiers used in the program, with array indices serving as a label to jump to.

It prompts the user for an input file name, allowing an optional output file name (defaulting to 'output.txt' if not provided). It reads the input file, tokenizes it using the Lexer class, parses the tokens for syntax analysis using an external parser function, generates a symbol table based on the syntax analysis using the SymbolTable class, and creates assembly code using the Assembly class.

After processing, it prints the assembly code to the console and writes both the symbol table and assembly code to an output file. Error handling is implemented using a try-except block to catch exceptions and display relevant error messages. The program's flow is structured within the main() function, initiating the entire process and handling user inputs and outputs accordingly. To improve, clearer comments explaining each step and enhanced error messages could make it more user-friendly. Additionally, optimizations in code structure and potential enhancements in error handling could be considered for better reliability.

Regarding the syntax\_analyzer.py file, it contains functionalities related to the syntax analysis phase. Here, functions or a class (like a parser) handle parsing the tokens generated by the lexer. This phase verifies whether the token arrangement follows the syntax rules defined by the programming language's grammar. It structures the tokens into a coherent format, such as a syntax tree or intermediate representation, ensuring adherence to the language's syntax rules.

Both files, lexer.py and syntax\_analyzer.py, are integral to converting source code into a machine-understandable format. The lexer dissects code into tokens, while the syntax analyzer validates the code's structure according to the language's grammar. These components provide organized data for subsequent stages, facilitating further analysis and eventual code generation or interpretation.

1. **Any Limitation**

None

1. **Any shortcomings**

One unique shortcoming we ran into was with ensuring our program ran the order of operations in the correct order. It does not do PEMDAS, but instead parses the string from left, not prioritizing the multiplication or division before the addition. Apart from that we did not have any other shortcomings.