CS323 Documentation

1. Problem Statement

- Assignment 3 involves two main tasks: symbol table handling and generating assembly code for a simplified version of Rat24S programming language. In the symbol table handling part, every identifier declared in the program must be stored in a symbol table along with its memory address, ensuring uniqueness and type matching. Procedures are needed to check for existing identifiers, insert new ones, and print out all identifiers in the table, while also handling errors for undeclared or duplicate identifiers. The second part requires modifying a parser to accommodate the simplified Rat24S syntax and generating assembly code instructions based on a virtual machine's stack operations. The assembly code should be stored in an array and include labels for jumps and references to identifiers. The goal is to produce a functioning compiler that converts Rat24S code into executable assembly instructions for the specified virtual machine.

2. How to use your program

- Unzip the file
- Open up assignment3.py (make sure to have necessary items installed (python3), etc.)
- Open up the console and input:
    - python3 assignment3.py [input_file] [output_file]
    - Where [input_file] is the name .rat of the file to be tested and [output_file] is the text file you want to export the result to.
    - Output results will also show in the console
    - For example:

```
python3 assignment3.py input1.txt output1.asm
```

3. Design

- Lexer and Tokenizer: A lexer is used to scan the input source code and tokenize it into meaningful units such as identifiers, operators, keywords, and literals.
- Parser with Recursive Descent Parsing (RDP): The parser is responsible for syntactic analysis using RDP. It consists of functions corresponding to each grammar rule, such as Assignment Statement (A), Expression (E), Expression Prime (E'), Term (T), Term Prime (T'), Factor (F), While Statement (W), and If Statement (I).

- Symbol Table Handling: A symbol table is implemented using a data structure such as a hash table or a tree. It stores information about identifiers such as their names, types, memory addresses, and scopes. Semantic actions within the parser update the symbol table as identifiers are encountered.
- Code Generation: The program generates assembly code instructions based on the semantic actions defined in the grammar rules. It uses an instruction table to keep track of generated instructions along with their addresses, operations, and operands.
- Stack Operations: The program utilizes stacks for handling control flow and expression evaluation. For example, a jump stack is used to handle jumps in while statements, and a temporary stack may be used for expression evaluation.
- Back Patching: Back patching is implemented to update jump addresses after code generation. It involves modifying previously generated instructions with correct jump addresses.
- Error Handling: The program includes mechanisms for error detection and reporting. Semantic actions and conditional statements within the parser check for syntactic and semantic errors such as undeclared identifiers, type mismatches, missing tokens, and incorrect syntax.
- Data Structures: The program utilizes data structures such as symbol tables, instruction tables, stacks (e.g., jump stack), and queues or lists for temporary storage during parsing and code generation.
- Algorithms: Algorithms for parsing (RDP), symbol table management (insertion, lookup, and scope handling), code generation based on semantic rules, stack operations (push, pop), back patching, and error handling algorithms are implemented.

4. Limitations

- None

5. Shortcomings

- None