CPSC 323 Documentation

1. Problem Statement

The project aimed to develop a software tool that could generate intermediate assembly code from an input code. The tool used a SLR parser to parse the input code, which used a parsing table to shift or reduce states into or out of a stack. The parser was able to determine uninitialized variables and variable redefinition errors by using a symbol table. The output assembly code was generated during the SLR parse by applying semantic actions to the non-terminal symbols in the input code. These semantic actions generated intermediate code fragments that were combined to produce the final output assembly code. Overall, the project was successful in generating intermediate assembly code that was a faithful translation of the input code into the target assembly language.

2. How to Use the Program

Dependencies and Environment

- OS: Mac OS || Linux
- Compiler: LLVM Clang

Installing

- Open your terminal and execute the following command:
    - git clone https://github.com/guchiyams/compiler-construction.git
- Wait for the cloning process to finish. Once the process is complete, you should see a new directory with the name of the repository in your chosen directory.
- You can now navigate to the cloned repository and start working with the code.

How to run the program

- Open the command line of your OS
- Change into downloaded repository directory
- Execute the folliwng command:
    - ./ compile <input_file>
- **NOTE**: the <input_file> MUST be one of the following:
    - input_01
    - input_02
    - input_03
- NOTE: the output of the program will be printed into the corresponding input file suffix number output file inside the output/ directory (e.g. input_01 -> 01_output.txt)

- NOTE: for debugging, a parse log is generated in the corresponding input file suffix number output file inside the parse_log/ directory (e.g. input_01 -> 01_output.txt)

3. Design of your program

The goal of the project was to develop a software tool that could generate intermediate assembly code from an input code.

The SLR parser works by using a parsing table to shift or reduce states into or out of a stack. The parsing table is constructed by analyzing the grammar of the input language. The parser reads the input code from left to right and pushes tokens onto a stack. When it encounters a reduce action in the parsing table, it pops tokens off the stack and replaces them with a non-terminal symbol. This process continues until the input code is fully parsed and the resulting parse tree is constructed.

The SLR parser implemented in the project was able to determine uninitialized variables and variable redefinition errors by using a symbol table. The symbol table kept track of all the variables that had been initialized and their respective values. When the parser encountered a variable in the input code, it checked the symbol table to see if the variable had already been initialized. If the variable had not been initialized, the parser reported an error. If the variable had already been initialized and the input code attempted to redefine it, the parser reported a redefinition error.

During the SLR parse, the output assembly code was generated by applying semantic actions to the non-terminal symbols that were generated during the reduction steps. These semantic actions were defined in the grammar of the input language and were executed by the parser as it built the parse tree.

The semantic actions typically generated code that corresponded to the operation being performed by the input language. For example, if the input code included an arithmetic expression, the semantic action associated with the reduction step for that expression would generate assembly code that implemented the corresponding arithmetic operation. The generated code was then added to the output assembly code.

As the parse tree was being constructed, the parser maintained a stack of intermediate code fragments that were generated by the semantic actions. When a reduction step was executed, the parser popped the corresponding intermediate code fragments from the stack, combined them according to the semantic action, and then pushed the resulting code fragment back onto the stack. This process continued until the parse tree was fully constructed and the final intermediate code fragment was at the top of the stack.

Once the parse was complete, the output assembly code was extracted from the top of the stack and written to a file. This code represented a faithful translation of the input code into the target assembly language.

Overall, the project was successful in generating intermediate assembly code from an input code. The use of a SLR parser and a symbol table made it possible to identify and report errors related to uninitialized variables and variable redefinition. This tool could be useful for programmers looking to convert high-level code to assembly code.

4. Any Limitation

None

5. Any shortcomings

Not able to do type checking when attempting arithmatics.