323 Assignment 3 Documentation

PROBLEM STATEMENT:

The objective of this project is to implement and extend the Rat25F syntax analyzer into a compiler for the "Simplified Rat25F" version of the language. Now, the program has two new functionalities, which are Symbol Table Handling and Assembly Code Generation.

HOW TO USE THE PROGRAM:

- **Compilation:** Make sure the main, parser, lexer and input file are all in the same directory. Compile the code using:
  g++ -std=c++17 lexer.cpp parser.cpp main.cpp SymbolTable.cpp assembler.cpp -o rat25f

- **Execution:** Execute the program with:
  ./rat25f

- **Input:** The input files are three text files called “test_small.txt”, “test_medium.txt”, and “test_large.txt” which contain the source code, or you can create your own .txt file.

- **Output:** The program will generate outputs containing assembly code listing and a symbol table, and are named their respective source code’s name + “_assembly.txt”.

Example input:

```
#
integer i, max, sum; "declarations"
sum = 0;
i = 1;
get (max);
while (i < max) {
sum = sum + i;
i = i + 1;
}
put (sum + max);
#
```

DESIGN:

Components

- Lexer: Processes the input characters one by one.
- Tokens: A data structure to allow the parsing to operate properly.
- SymbolTable: Manages variables and adds a memory address to new identifiers.
- Assembler: Manages the generation and storage of assembly instructions.
- Parser: Now includes *SymbolTable* and *Assembler*.

Data Structures

- **std::map<std::string, Entry> table**: Hash map storing variable names as keys and Entry structures as values, enabling O(1) lookup for symbol information.
- **Entry structure**: Bundles together all information about a declared variable (name, type, memory address).
- **int next_available_address**: Counter tracking the next available memory location, starting at 10000 and incrementing with each new variable.
- **std::vector<Instruction> instructions**: Dynamic array storing all generated assembly instructions in sequence.
- **std::stack<int> jump_stack**: Stack structure for managing nested control flow (if/else, while loops) by tracking instruction addresses that need back-patching.
- **Instruction structure**: Represents a single assembly instruction with operation code, operand, and instruction address.

Key Functions:

SymbolTable Functions:

- **symbolPush(std::string var_name, std::string type)**: Adds a new variable to the symbol table with a unique memory address; returns false if the symbol already exists (duplicate declaration error).
- **getAddress(std::string& lexeme)**: Retrieves the memory address of a variable by its name; returns -1 if the variable is not found in the table.
- **printTable(std::string filename)**: Outputs the complete symbol table to a file in formatted columns showing identifier names, memory addresses, and types.

AssemblyCodeGenerator Functions:

- **gen_instr(std::string op, int operand)**: Generates a new assembly instruction with the given operation and operand, assigns it the next instruction address, and adds it to the instruction list.
- **get_address()**: Returns the address of the next instruction to be generated, used for jump targets and back-patching.

- **back_patch(int instr_addr)**: Updates a previously generated jump instruction at the specified address with the current instruction address as its target.
- **push_jump_stack(int addr)**: Saves an instruction address onto the jump stack for later back-patching (used when entering if statements or loops).
- **pop_jump_stack()**: Retrieves and removes the most recent instruction address from the jump stack, returning it for back-patching.
- **print_assembly(std::string filename)**: Outputs all generated assembly instructions to a file in sequential order with addresses and operands.

LIMITATIONS:

None

SHORTCOMINGS:

None