University of Leeds **School of Computing**

COMP2932, 2019-20

Compiler Design and Construction

Jack Compiler

By

Hollow Man

hollowman186@vip.qq.com

Hollow Man

Date: 04/05/2020

1. Introduction

I use Python 3 to realize this compiler. I believe I have meet and realized all the requirements on the project instruction. The reason why I choose Python 3 is because I like Python and Python has easy-to-implement data structures.

Gitlab location: https://gitlab.com/HollowMan6/jack-compiler-songlin-jiang

2. The Lexical Analyser

I use nested case statements (if… elif… since Python doesn't support case) to implement the lexical finite automata logic. The reason why I choose such kind of way instead of table driven is to avoid complexity and make the reader easily understand what my code is about.

I created a isolated python source file called "lexer.py" so that if you want to use the lexer, just import it as a library. As required, I divide the tokens into 10 types – Identifier, Integer, String, Boolean, Null, Symbol, Keyword, Operator, Method, EOF. And I created a class Token, it includes Method GetNextToken() and PeekNextToken() the source code is stored in the code parameter in the form of a list, and such list consists of strings, and a string in such list is the content of a line from source code file. For example, code[line] represents the line string in the source file, while code[line][pointer] represents the pointer character in a specified line number. I also have line parameter for storing the line number that has consumed, and the pointer parameter for storing the character number specified in that line that consumed.

When first entering the GetNextToken() Method, it will handle out whether the source file is empty or pointer or line parameter exceeds the actual size. Then it begins to consume beginning tab, whitespace and comments until the next token occurs. If we use /* without */, the lexer will stop with an error message emerged. Finally we begin to identify those tokens, emerge errors if a string ends without " in a line or there is a symbol that not allowed or a wrong identifier. If succeed, the function will firstly increase the pointer, and then return a tuple which contains the lexem and the type.

The PeekNextToken() Method just keep the line and pointer number and then execute GetNextToken(), finally, restore the line and pointer number.

During the first version's test, error occurred when it tried to identify the divide symbol. It came out to be that there's a logic problem in my comment identification process.

Then I use the following test set to test my programs – An empty file, a file with only empty lines, a file missing */ for comment, the test set given. My lexer turns out to work well on those test sets.

3. The Parser

Construct using recursive descent parsers taught in Lecture 8. Using the grammar based on the courseware:

Full Jack Grammar

classDeclar → **class identifier** { {*memberDeclar*} }
memberDeclar → *classVarDeclar* | *subroutineDeclar*
classVarDeclar → (**static** | **field**) *type* **identifier** {, **identifier**} ;
type → **int** | **char** | **boolean** | **identifier**
subroutineDeclar → (**constructor** | **function** | **method**) (*type* | **void**) **identifier** (*paramList*) *subroutineBody*
paramList → *type* **identifier** {, *type* **identifier**} | ε
subroutineBody → { {*statement*} }
statement → *varDeclarStatement* | *letStatemnt* | *ifStatement* | *whileStatement* | *doStatement* | *returnStatemnt*
varDeclarStatement → **var** *type* **identifier** { , **identifier** } ;
letStatemnt → **let identifier** [[*expression*]] = *expression* ;
ifStatement → **if** (*expression*) { {*statement*} } [***else*** { {*statement*} }]
whileStatement → **while** (*expression*) { {*statement*} }
doStatement → **do** *subroutineCall* ;
subroutineCall → **identifier** [. **identifier**] (*expressionList*)
expressionList → *expression* { , *expression* } | ε
returnStatemnt → **return** [*expression*] ;
expression → *relationalExpression* { (**&** | **|**) *relationalExpression* }
relationalExpression → *ArithmeticExpression* { (= | > | <) *ArithmeticExpression* }
ArithmeticExpression → *term* { (+ | -) *term* }
term → *factor* { (* | /) *factor* }
factor → (- | ~ | ε) *operand*
operand → **integerConstant** | **identifier** [.**identifier**] [[*expression*] | (*expressionList*)] | (*expression*) | **stringLiteral** | **true** | **false** | **null** | **this**

But I find some mistake on the graph above, which leads to the failure to parser some of the test files provided. So I make corrections as follows:

1. *operand* -> **integerConstant** | **identifier** [.**identifier**] [[*expression*] (*expressionList*) | (*expressionList*)] | (*expression*) | **stringLiteral** | **true** | **false** | **null** | **this**
2. *relationalExpression* -> *ArithmeticExpression* { (= | (>(=|ε)) | (<(=|ε))) *ArithmeticExpression* }

Then I test it using the test file provided, after debugging, all the function works well.

4. The Symbol Table

I use dictionary in Python to store the symbol table. Key of the main symbol table is a list called level, which represents the name of the sub table, and reflect its hierarchy. values of the main symbol table are the sub tables, which use dictionary to store the identifier. Key of the sub symbol table is the name of the identifier. values of the sub symbol table are lists include the type, kind and a Boolean record whether the identifier has assigned.

The symbol table use the key of the main table to find the exact sub table. Which works as follows. When enter a new slope, the compiler will push the name of the class/function/etc. into level and use it as the key to create new sub tables. If the slope ends, level will pop it's last element to enter the parent slope. I also set up the if and while number to help identify those different slopes and create new tables by push "if/while" + number into level and number add 1 to itself.

5. The Semantic Analyser

Check if it's the first time to declare a variable: If you pass False to the "new" parameter, the add method in the symbol table will check whether that exist and return a Boolean, thus realizing the check. The find method can search all the parent slope to identify whether the identifier exists by the "deep" parameter.

10th Semantic Analyser Check: check after executing class declaration parser to see whether there exists unreachable code outside the class block and check after executing return statement parser, and to judge whether it's a "}" end of block symbol or an "else" keyword so that it can help find unreachable code after return statement.

Identify a class that hasn't been compiled: I use a simple version of parser which is located in SymbolTable.py to first build the symbol table, then again use the full version of parser to thoroughly check the grammars.

6. Code Generation

I created 9 methods for writing generated code: writePush(), writePop(), writeArithmetic(), writeLabel(), writeGoto(), writeIf(), writeCall(), writeFunction(), writeReturn() in jcparser.py. They are placed in the due part of the parser to generate the code.

7. Compiler Usage Instructions

If you want to run with source code on a DEC10 machine, after open the folder, you can type "./myjc.py myprog" or "python3 myjc.py myprog"

The compile will compile every jack file (including files in the sub-folder). If problems occur during handling specific source file, the compiler will just give up compiling this file and continue compiling the others remained in the folder. When all the files are handled, The compiler will show Compilation Complete! on the console.

Due to the coronavirus situation, I can only test it on my Windows 10 Machine instead of a DEC 10 machine, but I think it should work properly since I didn't use some special libraries. The only problem may be the LF and CRLF, if that do happen, please use command "dos2unix" to convert all my source files.