University of Leeds **School of Computing**

COMP2932, 2018-19

Compiler Design and Construction

Jack Compiler

By

Jonathan Alderson

201094818 sc17j3a

Date: 2nd May 2019

1. Introduction

I have managed to implement all stages of the compiler and all the semantic rules. My compiler can run all the test files in the VM Emulator successfully, and reports errors correctly. For some of the test files my compiler reports errors, this is because some of the programs break the semantic rules we were asked to adhere to, so this is to be expected. Because of this, I added a way to compile, and only show the errors as warnings. Even when my compiler gives errors, the code can still be ran, this is just because of the semantic rules we had to enforce.

My choice of programming language was Python, as it is my most comfortable language to develop in, has nice string slicing and is not verbose compared to a language like Java.

I managed to follow the plan and got all the milestones in on time.

I developed this compiler between my personal laptop and the DEC-10 machines. Even though it is python based, I still had a few issues to work out when switching from Windows to Linux, but the compiler has been completed on Linux, so runs perfectly on the DEC-10 machines.

2. The Lexical Analyser

I made my lexer as a class, it takes a filename as input and the optional argument of errors vs warnings. I added the option to only report errors as warnings, this is because they are some discrepancies between the test code and what our compiler should accept. In a few test files, they allow for uninitialized variables to be used in loops and allow returning a class reference from functions that are expected to return integers. Because of this, I added the option to show a warning, but not stop compilation.

The lexing is done one file at a time, the comments are removed first. This is done by checking for both // and /* */ style comments.

Next, all the new lines and spaces are removed.

Then the code is split up. This is done by having a large list of important tokens that we know about and splitting whenever you encounter one of these. There is special care taken around strings since they should not be altered, so no splitting is done inside the string literals.

When lexing, checks are done to work out what type everything is, first we check through a saved dictionary to see if each token is a keyword or known symbol, all others are either variables or variable names, so they can be integers, strings etc or just identifiers. I save all my tokens into an xml format for debugging. Some errors reported here are unexpected EOF and unrecognised characters that are not allowed in JACK.

I tested my lexer by looking at the xml output and seeing if it was what I expected. I worked through all the standard programs to see if they lexed successfully, as they were all syntactically correct, so they were good to test against.

3. The Parser

The parser is the most complicated file in the compiler. To create it, I looked very closely at the jack grammar slide and started outlining all the functions that would be needed. I initially tried a much more difficult approach, instead of having a function for each non-terminal i.e. a subroutine call. I tried implementing a dictionary which contained the production rules, and then having a general function, that could take any production rule, and if it knew the terminals and non-terminals it produced, parse it successfully. This was working quite well until I had to do debugging. The recursive depths got very large, and it was too complex for me to complete.

When I changed my approach to each non-terminal being in its own function it became simpler. Now, before calling a function, the terminal is peeked, if it meets the criteria for the next function it carries on to that one. Inside my parser there are lots of extra functions and data structures that help with semantic rules. It was very difficult to keep a file of this length tidy and understandable, so I put a lot of effort into making it laid out well. Error reporting is done inside each function, if the next token is not what is expected, then an error is reported based on what is missing. It was important to make these clear to the user, so the file and line number were important. Every error message is different, so it will tell the user what is missing, to be as helpful as possible.

I tested my parser by looking printing out whenever it went to a new function and comparing hand traces of where I would expect it to go. This proved very helpful, as I realised some of my expressions were being parsed wrong, when the recursion did not surface at the right time, causing things to be accepted correctly, but in the wrong order. I also ran lots of test files with known errors in to try and get my parser to find them. I found it best to narrow a problem down, if a whole file does not parse, make it as small as possible while still including the part that causes the bug, so you are looking at the least amount of information you need.

4. The Symbol Table

In my compiler. I have three different symbol tables.

The global class table stores information about each class, it stores all the fields and variables about the class.

The global method table stores, for each class, the functions inside the class, and the variables and arguments inside each function.

I also have my own class methods table. This stores, for each class, for each function, the name, return type and number of arguments.

When we parse a new class, we add it to the global class table. Since I am using a dictionary, I can check if the class already exists as I create it. I then add the field and variables from the class as I parse through the classVarDeclar function. In a similar manner, I add to the other tables as I parse the relevant information. For the class methods, every time I parse a new function, I note down the arguments and return types, and add it to the dictionary. Once a file has been parsed, it returns all these tables to the compiler, to make one large global table. So, for every class and every function in every class, we have all the information we need.

The symbol table is used in lots of semantic checks. Whenever we need to check what the type of something is, we refer to the symbol table. I implemented these checks with a function, that would find the type of what you wanted. If we had not parsed something yet, such as a variable from another class, since we had not parsed it yet, we would have to assume it was correct and carry out the checks at the end.

If I were redoing this project, I would like to make my use of symbol tables better, I think using dictionaries was good, but I should have used a sub dictionary inside each one, this may have made the indexing more understandable.

For example, when I add a new variable to the class table, I have gone through the classVarDeclar function, so I have found the varName, varType, varKind and varID. So, this line is understandable. It may not be apparent what True means, this is to show that the variable is initialised since it is a class field, but I could have done something to make this easier to read.

```
1.  self.classSymbolTable[self.currentClass].append([varName,varType,varKind,varID,True])
```

```
2.  for variable in self.classSymbolTable[self.currentClass]:
3.          if(variable[0] == nextToken[0]):
4.              info = [variable[2],variable[3]]
5.          if(variable[1] == nextToken[0]):
6.              newInfo = [variable[2],variable[3]]
```

Because I have a dictionary referring to a list, when I iterate through the variables in the symbol table, the indexing gets messy. It is not apparent what ‘variable[0]’ or ‘variable[1]’ means. If I had created my token and symbol table as a sub-dictionary, I could have done ‘variable[varName] = nextToken[value]’ and it would have been more understandable. Instead of using indexes everywhere.

5. The Semantic Analyser

I implemented my semantic analysis one at a time by following the resource on Minerva. This was quite hard, because the examples were not very extensive, and most was not real jack code, so running the provided examples would give errors anyway, since they were invalid jack files.

Implementing the semantic checks would require extra data structures and add more complexity to the parser. This became increasingly hard to deal with as the parser got bigger and bigger. I read through all the semantic checks first to plan out what data I would need to do them all and planned out the structures.

I have implemented all the semantic checks and am happy with how they perform. Some of the checks become difficult such as type checking,

```
1.  var boolean b;
2.  let b = (5 < 6);
```

Here we have an example of some jack code. We have a Boolean, and it is being assigned to two ints. We get an error because you cannot assign a Boolean to an int. This gets more complicated, because in some cases, you can set a Boolean to be an int, if the value is 0 for example. Since this is the same as 'False'. Lots of times in the test files, the source code assigns the value of an uninitialized integer to a Boolean. The logic follows that since the int has not been set yet, it is a 0. Since 0 is the same as 'False' it all works out. But our semantic rules would give two different errors, because it is uninitialized and of different types. The example above gets more confusing because the < operator returns a Boolean value, so the above example is fine. From discussing this problem with other students, I know some have checked to see if a '<' or '>' operator is in an expression and call it as returning a Boolean value. But then, we just arrive at a problem such as.

```
1.  var boolean b;
2.  let b = (5 < 6) + 7;
```

So now, their semantic rule will pass, since it has gone the crude route of checking for a '>' or '<' operator, when this will not work. I have decided just to allow the user to turn off errors so these files can be compiled. Since we have discrepancies in our semantic rules.

Creating the semantic checks for all code paths return a value was interesting to solve. I edited my If Statement parsing function to return 'True' if both the first If Condition returned a value and the Else condition returned a value, and 'False' otherwise. This made my checks recursive and clean. To see if all code paths returned a value, I only had to see if there was a return statement encountered when we were not inside a loop. Or, an If Statement returned 'True'. Since, if it did, both sections inside it returned 'True'. If they were nested, it would still be fine, if all the nested if statements also return 'True' then we are fine.

Testing was done by creating my own series of test files, specifically for the semantic rules, I think this would have been handy to have in the test files, since everyone could have their rules more rigorously checked, since all the test files should pass, it would have been nice to have some difficult ones, like heavily nested if statements to check all code paths return a value.

6. Code Generation

Code generation was very difficult. The code generation lectures did not cover all the things we needed. We were not given any compiled VM code to check, a compiler or a VM Emulator.

To start, I found every time I accepted a token and inserted the VM code. I have learned in tutorials how to handle the simple things, when parsing my functions, I would find the name, type, number of local variables etc. So, it was simple enough to add the VM code. For if/while statements I used counters to keep track of the start and end of the loops.

Below is the simple case for adding the integer constant. As stated before, using sub dictionaries would have got rid of some of the indexing to make the code more understandable.

```
1.  if(nextToken[1] == "integerConstant"):
2.      # code generation
3.      self.vmCode.append(["push","constant",str(nextToken[0])]
```

After I had finished all that I knew, I compared the output from small test programs to what I expected. I knew to get anything to run I would need to compile with all the jack-os standard libraries. We had not been provided with them, so I found them online. These files were difficult to compile as they did lots of unexpected things I had not accounted for that broke our semantic rules or broke my parser. Lots of time was spent fixing these issues. Once these compiled, lots of my code generation was still wrong. To fix this, I downloaded a real jack compiler, and would compare the outputs on small programs, and see what the differences were. I did most of my learning for the difficult cases from here. A very common thing in the compilation was encountering a token, but not adding it to the VM code as you parsed it. Rather, adding it after the end of the next token, e.g. when you encounter a '+' sign. We do not insert it there; we parse the following expression then add it. I often used a temp list to store the things we would add at the end. This fixed a lot of my parser's issues.

Once I thought I had finished my code generation, I would go through all the test files and compile them using the NandToTetris compiler. I then compiled all of them again using my own compiler. Then, using a diff checking tool, go through and find all the differences. By this point in my development, around 99% of my VM code would be correct. It was just a case of spotting the patterns where my code was wrong and going through and fixing them. These were all non-trivial edge cases that I was not aware of. Such as void functions having to do a 'push constant 0' at the end. For items like arrays, they were difficult to understand, so I would make very small test files to understand how they work and insert my own VM code until they generated the same code.

Near the end, I struggled to find why my VM code would be wrong. These were things such as functions vs methods, how with constructors you would have to push an extra items or variables sometimes have a higher index because you must account for the 'this'. Now my VM code is all correct, but lots of testing was required to get them to work.

I think it would be good if we had more lectures on code generation specifics, and if we were given the full jack-os with the filled in functions, a compiler to check against and if we had the same semantics as the test files. If I were to improve the module, I would make the files be consistent with our grammar and semantics, since now we get errors when things are correct, and it does not feel very unified.

7. Compiler Usage Instructions

Running

Navigate to the directory the compiler is saved in.

```
# how to run the compiler
python3 compiler.py directory

# example with directory name
python3 compiler.py Jack_Programs/Set\ 3/Pong/
```

When compiling a directory, make sure to include the final '/'.

My compiler requires the jack-os directory to compile, since its parsers these files for the symbol tables when anything else is compiled.

Settings

Inside the compiler.py file, at the top, there are a few Booleans that can be changed for what printout you want to receive and if you would like errors to stop compilation or just be warnings.

Saving

Once compilation is complete, the VM files will be saved in the same directory as the jack file. The jack-os files will not be saved since they are compiled every time. You can run the code in the VM Emulator by selecting the directory that was compiled. Since it will now include the VM files that have been created by my compiler.

If there are any further issues with compilation, please contact me at

sc17j3a@leeds.ac.uk

Thank you for marking,

Jonathan Alderson