Sean Kemp 1604413 26/11/21

Compiler Report

1 Scanner

The first action the scanner will need to do is to read in the source code from a file, so I pulled in the FileReader, IFileReader and Position files from the labs; these allow the scanner to move through the file character by character while keeping a current position which is useful for error reporting. I also pulled the ErrorReporter file from the labs so the compiler can display helpful errors when compiling to help in debugging, and the Debugger file which allows the writing of debugging logs when the DEBUG variable is set to true. I then pulled in the main class file, Compiler, from the labs and commented out the unnecessary code at this time: the parser, identifier, type checker, code generator, target code writer and the output file arguments. The Compiler class is what performs the compilation process and writes a message reporting on the success of the compilation.

With these files added I started on the tokenization of the source code, which is where the lab language and the assessment language starts to differ. I first pulled in all the Tokenization files from the lab into the solution, Token, TokenType and Tokenizer. The Token class holds all of the information that each token needs, token type, spelling and position. The TokenType class holds the types of tokens that are in the source language including non-terminals, terminal reserved words, terminal punctuation and special tokens, and it has a mapping of keywords in the language to the token type Enums. The Tokenizer class is where the scanning and tokenizing of the source code happens to turn the code into tokens. Both the TokenType and Tokenizer files needed to be altered to fit the new grammar in the assessment language.

The first step I took was to update the TokenType file as there were different and new token types. I first removed the unneeded types, Const, Semicolon and Is, then I moved Becomes to the reserved words as this was the new assign token. After that, I added the new types Nothing, Forever, For, Comma and Tilde. Nothing is used for blank commands; Forever for use in the while command; For for the new for command; Comma for replacing Semicolon in sequential commands or declarations, and for separating the grammar in the for parameter; and finally, Tilde for replacing the Is token in constant declaration. Lastly, I updated the Keywords dictionary to include the new token types and remove the old ones.

The next step was to update the Tokenizer file. The first difference in the grammar for the scanner was that the identifier must have a second letter or digit, instead of just needing one letter in the labs; for this, I updated the ScanToken method to include a check for the second character in the identifier section to make sure it was either a letter or digit. Next, I changed the Semicolon check to a Comma check, then removed the assign (=) check in the colon check section. The next big difference was in the character literal grammar as the apostrophe in the labs was changed to a quotation mark for denoting a character, and the allowed graphic in the quotations was limited to either a letter or digit, operator, or a single space; this was all taken care of in the character literal section of the ScanToken method. Lastly the comment denoter was changed from an exclamation mark to a dollar sign; this was done in the SkipSeperators method.

Error reporting was done for tokens that appeared as error token types and in more detail for identifier and character literal tokenizing.

2 Parser

I first decided to pull only the Parser file into the solution from the labs which allowed me to first update the differences in the parsing grammar code before adding complexity with the tree nodes, though I will explain both at once below. The Parser checks that the syntax of the program is correct and creates a tree showing the structure of the program. I decided to work and order the parsing methods in the order that the grammar appeared in the language specification. To create the tree structure, nodes were used for representing an abstract syntax tree; these were all pulled from the labs. The TreePrinter file was also added to allow for the node tree to be printed once the parsing was complete. To allow the parsing to run the IRuntimeEntity and TriangleAbstractMachine classes were needed, which at this point were empty apart from three declarations in the TriangleAbstractMachine.

Working through the methods of the Parser I first updated the ParseCommand method so that it uses Comma instead of Semicolon for sequential commands. Next, I changed the ParseSkipCommand method to be called ParseBlankCommand and updated it to accept a Nothing token before returning. I then updated the ParseWhileCommand so that it would include the new Forever grammar by adding a check for if the current token was forever then it would just accept Forever and Do then parse a single command, and then renamed the method to ParseWhileOrForeverCommand. Since the while forever command did not need an expression, I had to create a new node called WhileForeverCommandNode. To allow for the new node to be printed I also had to update the TreePrinter file ToString method, adding in a new case for the new node. To parse the new for command I created a new method called ParseForCommand which handles the accepting of tokens and parsing for the commands and expression according to the grammar. This also meant that I had to create a new node called ForCommandNode to hold the commands and expression, and like above created a new case in the TreePrinter ToString method for the new node to allow for printing. Once the new/changed methods were created/updated I then went back to the ParseSingleCommand method and updated the switch statement to add a case for Nothing and For, updated the While case to the new name of the parse while command method and changed the default case to report an error and return an error node.

The next section I had to adjust was the declarations. I updated the ParseDeclaration method to use Comma instead of Semicolon for sequential declarations then had to update both Const and Var declarations due to there no longer being a need for the Const and Var terminals in the assessment language grammar. I got rid of the ParseConstDeclaration and ParseVarDeclaration methods to avoid duplication of the position and identifier code and created and returned the Const and Var declaration nodes in the ParseSingleDeclaration method’s switch case. I also changed the switch cases and the token accepting inside them to Tilde and Colon.

The last change that needed to be done was in the ParsePrimaryExpression method to allow for a call expression which is new in the assessment grammar. To do this I had to add a check for a left bracket into the switch case for Identifier token type; I then accepted the brackets, parsed the parameter, and created and returned a new node. I created a new node for this expression called CallExpressionNode and updated the TreePrinter ToString method accordingly. I then uncommented the parsing code in the Compiler file to allow the compiler to parse and create a tree.

3 Semantic Analyser

To start off I first pulled in the four semantic analyser files from the labs, SymbolTable, StandardEnvironment, DeclarationIdentifier and TypeChecker. The SymbolTable is used in the identification stage for storing and retrieving identifiers that are known about to allow for quick look up and avoid repeating work; it also allows for different scopes through pushing and popping dictionary scopes from a stack. The StandardEnvironment holds all the global identifiers that need to be added to the symbol table at the global level so that they will always be in scope. The DeclarationIdentifier is the identification part of the compiler where it matches identifiers to their declarations; it discovers if there are any problems with out-of-scope variables or missing declarations. The TypeChecker carries out type checking for nodes in the compiler to make sure that the types of items such as commands and operands are appropriate for the context they appear in. The TriangleAbstractMachine file was also updated to include everything from the lab, except for the CodeBase address, to allow the semantic analyser to work. To fit the new grammar, it was the DeclarationIdentifier and TypeChecker files that needed to be changed.

The only new change that needed to be added to the DeclarationIdentifier was the identification on the new nodes which I had added in the parsing stage, WhileForeverCommandNode, ForCommandNode and the CallExpressionNode. The method added for the while forever command was called PerformIdentificationOnWhileForeverCommand, which performs identification on the WhileForeverCommandNode command child. The next method was the PerformIdentificationOnForCommand for the identification of a for command node which performs identification on the commands and expression child nodes. The last method added was the PerformIdentificationOnCallExpression method for identifying a call expression node; this performs the identification on the call expression's identifier and parameter child nodes.

Similar to the changes above in the identification stage the only new change in the TypeChecker file was the addition of three new methods, PerformTypeCheckingOnWhileForeverCommand, PerformTypeCheckingOnForCommand and PerformTypeCheckingOnCallExpression. The type checking for the while forever command and for command was fairly simple, just needing to call PerformTypeChecking on the command child of the while forever command, and the expression and command children of the for command; though there was also a need to make sure that the type of the expression child in the for command was boolean. The call expression type checking was more complex as once I had done the type checking on the children nodes, I had to check that the identifier was a function and then check the arguments of the function to make sure that it had the correct number and type, making sure to also set a type for the call expression as well using the function's return type. I lastly updated the Compiler file to allow the program to perform identification and type checking on the tree that was created in the parsing stage.

4 Code Generator

To start off I added the Code Generation files, Address, Instruction, IRuntimeEntity, RuntimeKnownConstant, RuntimeUnknownConstant, RuntimeVariable, ScopeSizeRecorder, TargetCode and CodeGenerator, replacing the empty IRuntimeEntity with the lab one; then I added the TargetCodeWriter file from the lab. Now that the Address file was added, I uncommented the CodeBase address in the TriangleAbstractMachine file. For the changed/new grammar the only file that needed to be changed was the CodeGenerator file.

Similarly with the previous semantic analysis files, I only had to add three new methods to the CodeGenerator file, GenerateCodeForWhileForeverCommand, GenerateCodeForForCommand and GenerateCodeForCallExpression.

The GenerateCodeForWhileForeverCommand method is used for generating code for the new while forever command; to code this I saved the address of the code generation for the command child and then added a jump instruction after the command was handled so that it would do the code generation again.

The GenerateCodeForForCommand method is used for generating code for the new for command; in the coding of it I first generate code for the first command child, then save the address of the next address, which is the code generation for the expression child, so it can be jumped back to when it needs to re-evaluate the expression. After the expression, an address is saved for the next action, the jump-if instruction, which jumps if the expression returns a false value; this is so that the jump address can be patched into the instruction later. Code generation is then done for the do command and second parameter command, with the jump instruction to the previous expression address being after it; lastly, the patch instruction is done for the jump-if instruction above.

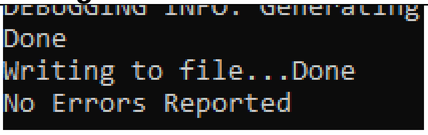
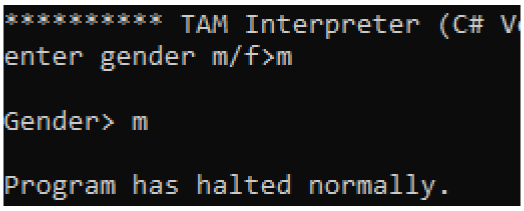
The GenerateCodeForCallExpression method is used for the code generation of the new call expression grammar and all it does is generate code for the parameter child and then the identifier child.

To finish off I updated the compiler file for the code generation and added the 2 extra parameters back into the file so that it is back to being similar to the lab compiler file. The parameters are for the output files of the compiler, one which outputs human-readable assembly code, and the other is what the Triangle Abstract Machine uses to run the compiled source code. To allow this to work I also updated the application arguments for the vs project, adding two extras, out.tam and out.txt.

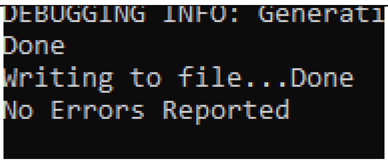
5 Testing

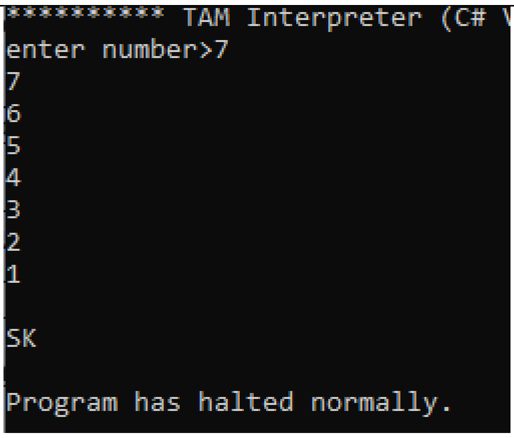
In my testing, I wanted to make sure that all commands, declarations, parameters, expressions, and literals were tested and compiled successfully. To do this I split my testing into three smaller programs that tested multiple aspects at once. I show what aspects were tested in each test program below. The expected outcome of the tests is that they will all compile successfully and work in the Triangle Abstract Machine.

The first program I created was test_program1 which focused on character declaration and assigning, while also testing other aspects listed below:

Aspects tested	Example of test	Test Passed	Screenshots of compile and finished Triangle Abstract Machine run
Let command	let … in …	Yes	 ```DEBUGGING INFO: Generating Done Writing to file...Done No Errors Reported```  ```********** TAM Interpreter (C# V enter gender m/f>m Gender> m Program has halted normally.```
Var declaration (character)	gender: Char	Yes	
Begin command	begin … end	Yes	
Sequential command	put("e"), put("n")	Yes	
Call command with value parameter (expression)	put("t")	Yes	
Character literal (making sure to test single space and operator in character)	"r" or " " or ">"	Yes	
Call command with var parameter	get(var gender)	Yes	
Call command with blank parameter	puteol()	Yes	
Call command with identifier expression parameter	put(gender)	Yes	

For test_program2 I used the example_chars file from the labs as a base to test commands using int literals, while also testing the new Call expression.

Aspects tested	Example of test	Test Passed	Screenshots of compile and finished Triangle Abstract Machine run
Let command	let … in …	Yes	 ```DEBUGGING INFO: Generati Done Writing to file...Done No Errors Reported```
Sequential declaration	MAX ~ 15, n1: Integer	Yes	
Const declaration with int literal	MAX ~ 15	Yes	
Var declaration (int)	n1: Integer	Yes	

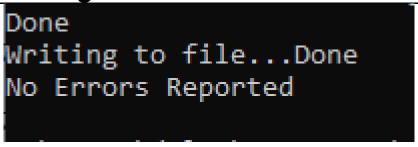
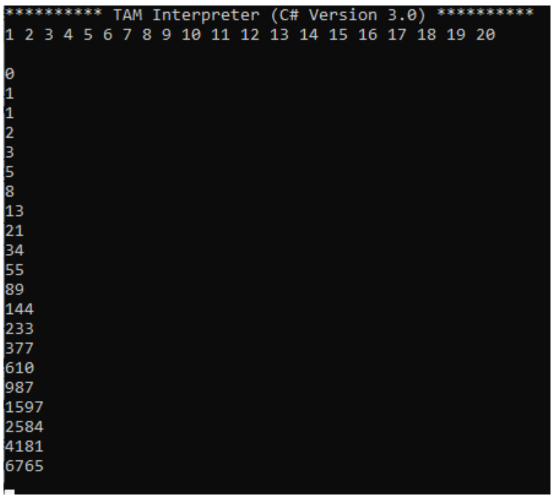
Begin command	begin … end	Yes	
Sequential command	put("e"), put("n")	Yes	
Call command with value parameter (character expression)	put("t")	Yes	
Call command with var parameter (identifier)	getint(var n1)	Yes	
If command with binary expression, using an operator	if n1 > 0 then … else …	Yes	
While command with binary expression	while n1 > 0 do …	Yes	
Assign command	n1 becomes n1-1	Yes	
Nothing command	nothing	Yes	
Call expression	chr(83)	Yes	

```
*********** TAM Interpreter (C# V
enter number>7
7
6
5
4
3
2
1

SK

Program has halted normally.
```

For test_program3 I wanted to test the remaining commands and expressions, while also testing a nested let command so scope changes can be tested, allowing the option to test that declaring a variable of the same name in a different scope is possible.

Aspects tested	**Example of test**	**Test Passed**	**Screenshots of compile and finished Triangle Abstract Machine run**
Let command	let … in …	Yes	 
Var declaration (int)	n1: Integer	Yes	
Begin command	begin … end	Yes	
For command	for (n1 becomes 1, n1 < 21, n1 becomes n1+1) do …	Yes	
Call command with Id expression parameter	putint(n1)	Yes	
Sequential command	putint(n1), put(" ")	Yes	
Nested Let command, declaring a variable with the same name as in the scope above	let n1: Integer, n2: Integer, sum: Integer in …	Yes	

```
Done
Writing to file...Done
No Errors Reported
```

```
********** TAM Interpreter (C# Version 3.0) **********
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
```

Assign command	n1 becomes 0	Yes	
While Forever command	while forever do …	Yes	
If command with unary expression and bracket expression (binary expression in brackets)	if \(n1 > 10000) then … else nothing	Yes	

The end result was that all tests were completed successfully with no errors that occurred. Test programs are included in the code zip file.