Catscript Compiler: CSCI - 468 Capstone

Ivan Cline
Matthew Keck
Spring 2024

Program

I have included a link to the zip file containing the Catscript compiler source code here: source.zip, or within the ”source/src” directory.

Teamwork

For this capstone project, I mainly focused on the implementation of the Catscript Compiler. This included programming the tokenizer, parser, evaluator, and bytecode compilation steps in Java. I also ensured that the Catscript compiler passed the unit tests needed to have a functional compiler, these tests were provided to me in the Compilers class. My partner, Matthew Keck, provided me with three additional unit tests to ensure that my compiler was working correctly. The first unit test included checking for valid return coverage in function definitions, the second unit test ensured that equality expressions were correctly evaluated when the expressions included in the binary operation were function calls. Finally, the third unit test ensured that the type inference system for list component types inferred correctly. My partner also wrote all of the documentation for my Catscript compiler, describing and providing examples for the unique features and behaviors of the Catscript programming language. Overall, I value my partner’s contribution to this project.

I would say that we both put a large amount of our time and effort into ensuring quality in this project. The documentation can be found within this write up in the section labeled “Technical Document”. My partner’s unit tests can be found below, and can also be found in the ”source\test\java\edu\montana\csci\csci468\demo\PartnerTests.java” file.

```
/*
The functionDefinitionStatementHasReturnCoverage test verifies that the
function definition statement throws a parse error
(ParseErrorException). When there is not a return statement in every branch of
execution. The first statement
doesn't have a return in either branch of the if statement. The second and
third branches have a return in one branch and not the other.
*/
@Test
public void functionDefinitionStatementHasReturnCoverage() {
   assertThrows(ParseErrorException.class, () -> {
       FunctionDefinitionStatement expr1 = parseStatement("function x(a : int)
: int {if(a > 10){  } else {  }}");
   });
   assertThrows(ParseErrorException.class, () -> {
       FunctionDefinitionStatement expr2 = parseStatement("function x(a : int)
: int {if(a > 10){ return 1 } else { }}");
   });
   assertThrows(ParseErrorException.class, () -> {
       FunctionDefinitionStatement expr3 = parseStatement("function x(a : int)
: int {if(a > 10){ } else { return 2 }}");
   }); }
```

```
/*The equalityExpressionWorksWithFunctionCalls test verifies that the equality
expression can return the correct
value if the operators are function calls. The first program asserts that
truthy, a function that returns true, and
falsey, a function that returns false, are unequal. The second program does the
same, except it uses the
not equal to the operator (!=) instead of the equal operator (==).
*/
@Test
public void equalityExpressionWorksWithFunctionCalls(){
   assertEquals("false\n", executeProgram(
           "function truthy(){\n" +
                   "return true" +
                   "}\n" +
                   "function falsey() : bool {\n" +
                   "return false" +
                   "}\n" +
                   "print(truthy() == falsey())"
   ));
   assertEquals("true\n", executeProgram(

           "function truthy() : bool {\n" +
                   "return true" +
                   "}\n" +
                   "function falsey() : bool {\n" +
                   "return false" +
                   "}\n" +
                   "print(truthy() != falsey())"
   ));
}
/*
The listLiteralOfDifferentTypesIsAssignedObjectType test verifies that the
program assigns the type object to the list
when there are at least two types. The first expression has both integers and a
string in its list.
The second expression has both integers and a boolean in its list. The third
and final expression has both a null object and integers in its list.
*/

@Test
public void listLiteralOfDifferentTypesIsAssignedObjectType(){
   ListLiteralExpression expr1 = parseExpression("[1, \"2\", 3]");
   ListLiteralExpression expr2 = parseExpression("[1, 2, true]");
   ListLiteralExpression expr3 = parseExpression("[null, 2, 3]");

   assertEquals("list<object><object>", expr1.getType().toString());
   assertEquals("list<object><object>", expr2.getType().toString());
   assertEquals("list<object><object>", expr3.getType().toString());}
```

Design Pattern - Memoization

A valuable design pattern that I chose to implement for this project was the memoization design pattern. Memoization was utilized in the “CatscriptType” class. The memoization pattern was used to quickly return the ListType for a Catscript list by looking up the Catscript component type as a key within a hashmap named “cache”. If that key did not exist, I would add the type as a key in the hashmap and have it point directly to the correct ListType.

This allowed for a constant time assignment and “creation” of ListTypes if the component type had already been added as a key to the “cache” hashmap. Catscript allows recursive list types, for instance, a list containing a list of integers would be a ListType “list<list<int>>”. When we constantly have to infer the type of these lists and instantiate new ListType objects, it can get computationally taxing, this is a notable benefit of utilizing the memoization design pattern. See my implementation of the memoization pattern below.

```
static HashMap<CatscriptType,ListType> cache = new HashMap<>();
3 usages  Carson Gross *
public static CatscriptType getListType(CatscriptType type) {
    cache.computeIfAbsent(type, k-> new ListType(type));
    return cache.get(type);
      return new ListType(type);
}
```

Technical Document

Catscript Documentation

The Catscript scripting language supports two main syntax features, statements and expressions, to perform computations. The statements group includes if, for loop, print, variable, function definition, and assignment statements. The expressions group includes equality, comparison, additive, factor, and unary expressions. The documentation below describes, explains, and gives examples of the statements and expressions in Catscript.

Statements

For Loop Statement

Catscript has only one iteration tool that allows the user to iterate over all list elements. In the example below, x represents the reference to the current component of the list for that iteration. Variables declared inside the for loop are considered first before checking if there is a global variable of that name. In other words, for loops have a local scope.

```
for(x in [1, 2, 3]){
```

```
    print(x)
}
```

If Statement

Catscript supports only two kinds of control flow: if statements and else statements. There are no else if statements or switch statements in the Catscript language. The expression that controls the flow of logic in the if statement must be an expression that produces a true or false value. Therefore, the only expressions that can be used are equality, comparison, and boolean expressions.

```
if(x > 10){
    print(x)
}
```

```
if(x > 10){
    print(x)
} else {
    print( 10 )
}
```

Print Statement

The print statement allows users to output the contents of an expression to the terminal.

```
print(x)
```

Variable Statement

Variable statements allow the user to allocate memory for a specific type. Catscript is a strongly typed language; however, the variable statement supports type inference, meaning the user can declare the variable type or let the compiler determine the type. Catscript supports five types for assigning the variable statement: int, string, boolean, object, and list. Examples of each assignment can be seen below. Because Catscript is statically typed, the variable's name, once declared, is reserved and can't be used in the same scope for other variables in that scope.

```
var x = "foo"
```

```
var x : list<int> = [1, 2, 3]
```

```
var x : object = null
```

```
var x : string = "hello world"
```

```
var x : bool = true
```

Assignment Statement

The assignment statement in Catscript allows allocated memory to be assigned new values. Because Catscript is strongly typed, the assignment must be the same type as the declared variable; more information is available in the variable statement.

```
x = null
```

Function Definition Statement and Return Statement

The function definition statement allows the user to jump to chunks of code and return to where the function was called. Functions are also typed based on what the function returns. If no return is used, the function return type is considered void. A return must be declared on every branch of computation unless the function's return type is void. All variables declared in the function are locally scoped and will be referenced first before variables in the global scope. The return statement is used in function definitions by using the keyword (return) followed by nothing to return null or an expression. Return statements must return the same type as the function.

```
function x() {
   print(1)
}
```

```
function x(a, b, c) {
   print(a)
   print(b)
   print(c)
}
```

```
function x() : int {
   return 10
}
```

```
function x(a : object, b : int, c : bool) {
   print(a)
   print(b)
   print(c)
}
```

Expressions

Equality Expressions

The equality expression is evaluated as true or false depending on whether its two operands are equal or not equal. The equality expression has two operators, equals (==) and not equals (!=). The equality expression must have two operands. The types the operands are allowed to be an int, boolean, string, and null. If the two operands are integers or booleans, the expression checks whether the values of the integers are equal or not. If the type of the operands are not both integers or booleans, the equality expression tests whether or not they are the same object. The equality expression doesn't test if two strings are the same.

```
1 == 1
output: true
```

```
1 != 1
output: false
```

```
true == true
output: true
```

```
null == null
output: true
```

```
true != null
output: true
```

```
true != 1
output: true
```

Comparison Expressions

The comparison expression evaluates two integer operands as true or false depending on one of four operators used. The four available operators are less than (<), greater than (>), less than or equal to (<=), and greater than or equal to (>=).

```
1 >= 1
output: true
```

```
1 > 1
output: false
```

```
1 <= 1
output: true
```

```
1 < 1
output: false
```

Additive Expressions

An additive expression can be evaluated as one of two different values. Two integers produce the first value as the difference or addition of the two integer literals. The second value is made by two types that are not integers, and the value will be the concatenation of them to a string. Examples of this can be seen below. An additive expression is left associative; the last example shows this property. The operators available to the additive expression are the addition sign (+) and the subtraction sign (-). The plus (+) operator must be used to concatenate strings.

```
1 - 1
output: 0
```

```
1 + 1
output: 2
```

```
1 + "a"
output: "1a"
```

```
"a" + 1
output: "a1"
```

```
null + "a"
output: "nulla"
```

```
"a" + null
output: "anull"
```

```
1 - 2 - 1
output: -2
```

Factor Expressions

The factor expression evaluates two integers as the product or quotient of the two integer literals. The two operators are (*), which corresponds to multiplication, and (/), which corresponds to division. Factor expressions, like additive expressions, are left-associative.

```
3 * 4
output: 12
```

```
12 / 4
output: 3
```

```
1 / 2
output: 0
```

```
24 / 2 * 2
output: 24
```

Unary Expressions

Unary expression evaluates two different values, depending on the operand type. When the operand is an integer, the value produced is the reverse of the operand's sign. When the operand is a boolean, a true value is evaluated to be false, and a false value is evaluated to be true. The operator for integer types is the minus sign (-), and the keyword (not) is used for boolean types.

```
not true
output: false
```

```
-1
output: -1
```

Function Call Expressions

The function call expression allows the user to refer to the function definition statement by passing values for each of the definition's arguments and receiving a value that is the result of the computation performed in the function definition statement.

```
foo(1, 2, 3)
```

Parenthesized Expressions

The parenthesized expression allows a user to increase the precedence of other expressions. The example below shows that we add precedence to the additive expression, which usually has lower precedence than the factor expression. The operators for parenthesized expressions are opening and closing parentheses.

```
(5+10)*2
output: 30
```

Identifier Expression

The identifier expression allows the user to reference allocated memory with a variable name. The variable name must be a word that starts with a letter of the alphabet or an underscore followed by zero to many letters or numbers. The identifier expression is used for names in many other Catscript statements, such as function definition, function argument, function call, variable, for loop, and assignment. Wherever we need to reference, a name-value pair identifier expression is used. The code block below gives an example of the identifier expression being used in a print statement.

```
var x : list<int> = [1, 2, 3]
print(x)
output: [1, 2, 3]
```

Integer Literal Expression

Integer literal expressions are Catscripts representation of integers. The number characters one through nine and the minus sign are used to represent the class of both negative and positive integers. The example below shows a variable statement storing a positive integer and negative integer. Integer literal expressions can be the operands for many expressions such as equality, comparison, additive, factor, unary, and parenthesized. Integer literal expressions are often the values that are stored by a list literal expression, variable statements, and

assignment statements. Integer literals are given the int typing according to the Catscript type system.

```
var x : int = 12984
```

```
var x : int = -490460
```

String Literal Expression

String literal expressions are Catscripts representation of strings. String literal expressions can be declared using opening and closing quotations; the example below shows a string being stored with a variable statement. The characters allowed in the quotes are any character in the alphabet, any symbol, and any integer. String literal expressions are operands for the additive expression and the equality expression. They can also be stored in a list literal expression, variable statements, and assignment statements. The string literals are given the string type with the string keyword according to the Catscript type system.

```
var x : string = “hello world”
```

Null Literal Expression

Null Literal Expressions are Conscripts representation of a nothing value. Null literal expressions are operands for the additive expression and equality expression. They can also be stored in a list literal expression, variable statements, and assignment statements. The null literals are given the type null using the null keyword according to the Catscript type system

```
var x : object = null
```

List Literal Expressions

List literal expressions are one of two ways a user can allocate memory in Catscript, the other being variable statements. The list literal expression can take any type in Catscript. The list component type is converted to an object if there are different types in the list. This makes the type of the list literal expression covariant on its component types. The list's literal expression cannot be written to. Once the list is declared, it is effectively immutable. The examples below show ways of storing the list literal with the variable statement. The list literal type is declared with the list keyword.

```
var listOfList : list<list<int>> = [[1, 2],[1, 2],[1, 2, 3]]
```

```
var listOfString : list<string> = [“hello world”, “goodbye
```

```
world”]
```

```
var listOfObject : list<object> = [null, “hello”, 3, true,
[1, 2, 3]]
```

Boolean Literal Expressions

Boolean literal expressions consist of two keywords: true and false. As you would expect, the boolean literal expression evaluates the true keyword to a true value and the false keyword to a false value. The boolean literal expression only has operands, no operators. The type of the boolean literal is declared with the bool keyword.

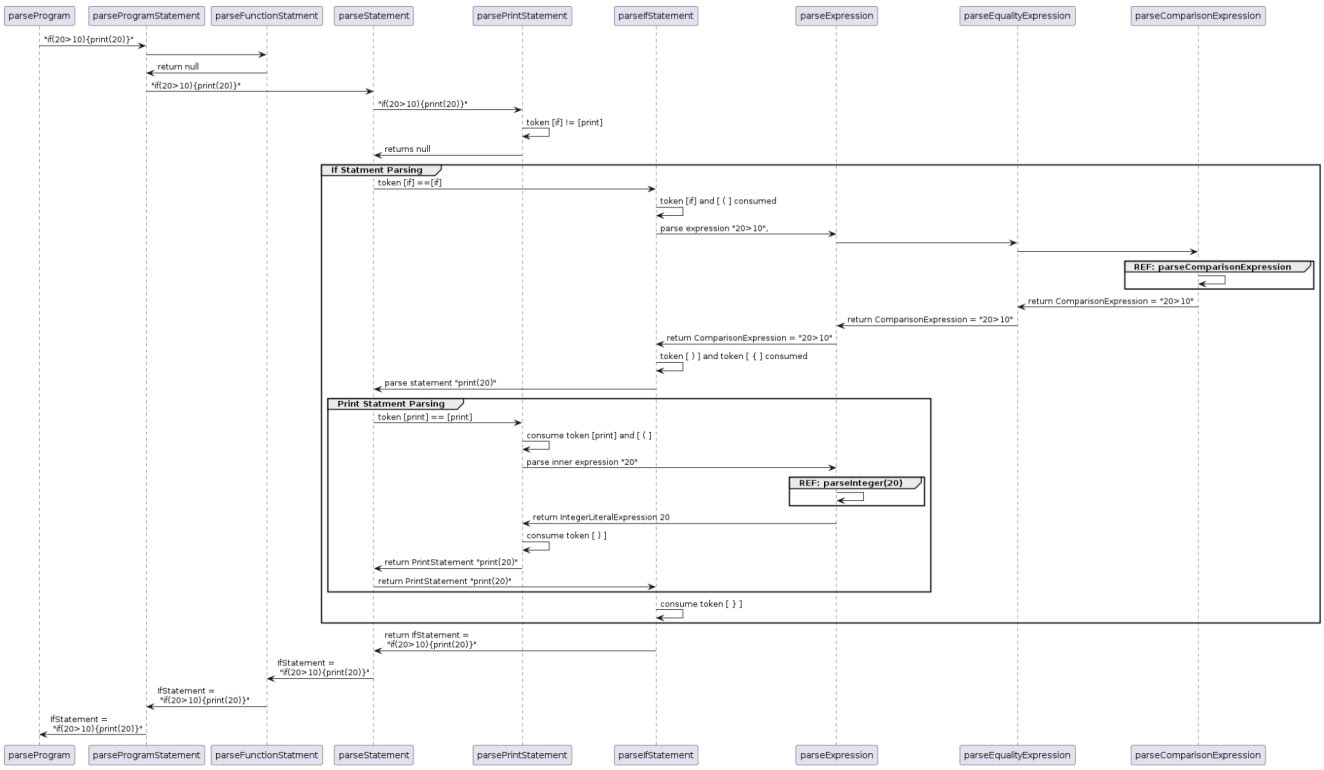
```
var trueBoolean : bool = true
```

```
var trueBoolean : bool = false
```

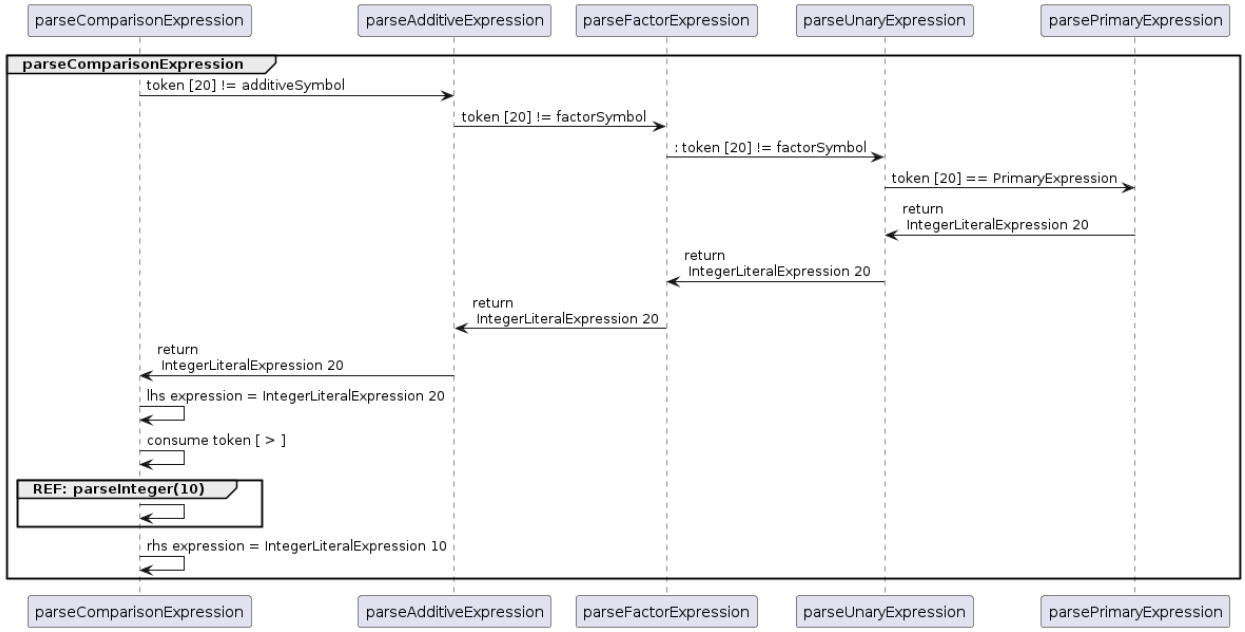
UML - Sequence Diagram

The sequence diagram that I included in this project will allow someone to visualize the recursive nature of a recursive descent parser. I showcased how the compiler would parse the already tokenized statement “if(20>10){print(x)}”, so that we may build a parse tree for evaluation. I have three diagrams below. Directly below is the Parse “if(20>10){print(20)}” sequence diagram, which shows the entire process of parsing the if statement. Within the Parse “if(20>10){print(20)}” sequence diagram, there are references to other diagrams denoted by the REF tag; these referenced sequence diagrams can be found within this section as well. I added references to other diagrams to make the Parse “if(20>10){print(20)}” sequence diagram more aesthetically pleasing. Think of these REF sequence diagrams as zoomed in portions of my sequence diagram that had to be moved elsewhere in this document.

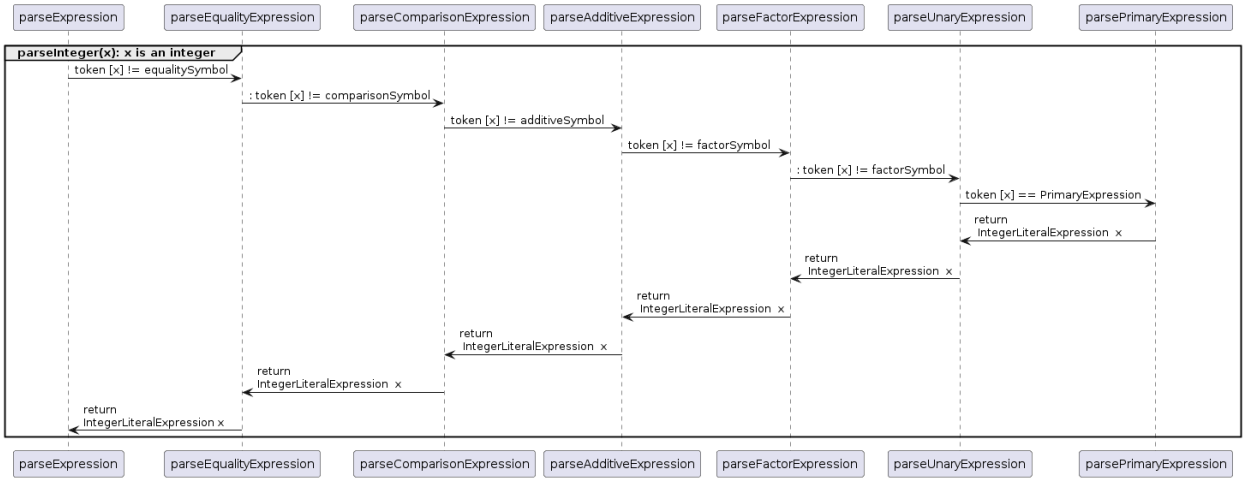
Parse “if(20>10){print(20)}”



REF: parseComparisonExpression



REF: parseInteger(x)



Design Trade Offs - Recursive Descent vs a Parser Generator

The main design trade-off that was considered during the creation of the Catscript compiler was whether I should use a recursive descent parser or a parser generator. Parser generators are essentially code generators. They are somewhat related to transpilation, as they typically take some lexical grammar in the form of a regular expression or grammar in the form of EBNF as an input, and output a generated tokenizer and parser for your described language. The parser generator that we talked about in the Compilers class was the ANTLR, or “Another Tool For Language Generation” parser generator.

To complete the first step of parsing, which would be lexing, this parser generator tool takes, as input, a .g file, which is where you would specify your language’s syntax. ANTLR would then provide you with a generated Java file named “org.example.TLexer.class” that would act as your compiler’s tokenizer or lexer. Creating the Catscript tokenizer without a parser generator was a lot more of an involved process, as it involved writing the entire tokenizer by hand in Java. This process took time, however, I believe that in contrast to using the ANTLR parser generator, I was given the freedom to tokenize Catscript however I would like with almost no restrictions. Also, the handwritten tokenizer was far easier to debug. A very real issue with the ANTLR parser generator, and parser generators in general, is that the code they generate is incredibly unreadable and almost impossible to debug. The tokenizer I wrote by hand may have lacked some level of optimization that the ANTLR parser generator possessed, but it was well worth it to understand and have control of every step in the Catscript tokenization process.

Creating a generated parser with a parser generator was very similar to creating a generated lexer. The finished generated parser would be the tool that generates and validates a parse tree for evaluation given an input of tokens from the tokenizer, or lexer. The parser generator tool would once again take, as input, a file with a list of EBNF rules to describe your language’s grammar. It would then output a Java file named “org.example.TParser.class“ that would be capable of lexing and parsing text files using your described grammar rules. Creating the Catscript parser without a parser generator involved implementing the recursive descent algorithm by hand.

The recursive descent algorithm for parsing is very intuitive and understandable while also being incredibly flexible and expressive. Handwriting the recursive descent parser in Java provided me with a rich understanding of the recursive nature of programming languages. It also provided me with the freedom to describe my grammar in whichever way I wanted and was still readable enough to debug and make further additions to. While using a parser generator is arguably a much quicker process, the intuitiveness and readability of my handwritten recursive descent parser likely saved me the time I took to write it through the ease of debugging alone. Using a parser generator also likely produces a more well-optimized parser, but those runtime gains do not tip the scales enough in favor of parser generators. Implementing the recursive descent algorithm by hand for Catscript has its many drawbacks as well, but when compared to a parser generator, utilizing recursive descent was undoubtedly the correct choice.

Software Development Lifecycle- Test-Driven Development

The Catscript compiler project's software development lifecycle was directly inspired by Test-Driven Development (TDD). Test-driven development involves fulfilling a large set of predetermined tests within a test suite as a way to track progress and add new features. If I were to continue this project, the addition of new features would also be driven by new tests, systematically adding more test cases to my test suite, while ensuring that the tests in the initial suite still work once the new feature has been fully implemented.

Using this software development lifecycle allowed me to verify that none of my changes or additions to the Catscript compiler were breaking other features I had coded before, and gave a satisfying measurement for tracking the production progress of my program. It also made debugging my code far easier, as I knew exactly what features in my code were failing, and could easily set breakpoints to hunt down the error in my code for that particular issue.

The tests in my test suite were provided to me in the Compilers class, and three of the tests in the test suite were provided to me by my partner. Writing tests that adequately cover all possible edge cases in my Compiler is likely not possible. However, this initial set of tests and the tests provided by my partner were adequate to create a fully functional Compiler that can handle typical use cases. This gives me a great product to start from, expand on, and share with the public if I so wish. Furthermore, utilizing the test-driven software development lifecycle inadvertently provided me with a huge suite of tests, and to some extent, documentation for others who would like to work on it including myself. Test-driven development worked very well for this particular project, and I am excited to use this framework in future projects going forward.