Program

The source code for this project can be found in the source.zip file alongside this PDF.

Teamwork

With a project as vast as building a Catscript compiler, it was evident that collaboration was imperative to succeed. It is, for this reason, that the development of catscript was split between two teammates, henceforth referred to as 'team member one' and 'team member two'. Each team member determined tasks that could be completed independently of one another to allow for synchronous progress to be made over the course of the semester.

Team member one was responsible for the implementation and programming of the Catscript compiler. They account for around 65% of the total time spent on this project. The code was started by building a tokenizer. This would first remove all the whitespace in the user's given code. The tokenizer would then scan each token and assign an identity for it. For example the token "{" would be identified as an "LEFT_BRACE" in the token list. Next, team member one built a parser to parse the tokens into the grammar of CatScript. This is done using the token list from the tokenizer. Statements and expressions were parsed using separate classes according to the grammar of catscript. After that, they worked on getting the execution logic completed for all statements and expressions. This logic is what allows the program to execute. Lastly, they worked on getting the bytecode to function properly. This would lay the fundamental foundation for the code to execute on hardware. Team member one relied heavily on team member two to verify their solutions performed the tasks correctly.

Team member two was responsible for the documentation and tests of the Catscript compiler. They account for around 35% of the total time spent on this project. Due to the test-driven development cycle that was utilized, the priority was to build tests for team member one to utilize. While this started as simple tests to see if the tokenizer worked properly, they quickly got to work producing more complex tests. Here are three examples of their tests and how they contributed to the success of this project:

```
// Designed to test whether an integer is assignable to an object in a function.
@Test
public void functionDefinitionStatementWithIntToObjectVarAssignment() {
    FunctionDefinitionStatement expr = parseStatement("function intToObj(x : int)
{ var y : object = x  return(y)} print(intToObj(5))");
    assertNotNull(expr); // Tests that function is properly parsed and assigned to
the "expr" variable
    assertEquals("intToObj", expr.getName());  // Tests that the function name is
identified as "intToObj"
    assertEquals(1, expr.getParameterCount()); // Tests that there is only one
parameter identified for the function
    assertEquals("x", expr.getParameterName(0)); // Tests that the parameter's
name is "x"
    assertEquals(CatscriptType.INT, expr.getParameterType(0)); // Tests that the
parameter type is INT
    assertEquals(2, expr.getBody().size()); // Tests that there are two statements
in the function body
    assertTrue(expr.getBody().get(0) instanceof VariableStatement); // Tests that
the first statement in function body is a Variable Statement
    assertTrue(expr.getBody().get(1) instanceof ReturnStatement); // Tests that
the second statement in function body is a Return Statement
```

```
}
```

This test case checks for if a function definition statement can convert an integer into an object and return it. To check this is the case, team member two first writes some code in catscript and passes it to the parseStatement function. After it is parsed, it is checked that it exists and that the correct name has been assigned to the function. The parameter is then checked for if there is only one parameter, it has the right name, and is of the type integer. Next, the body of the function is verified that there are only two elements contained in it. Lastly, the two body elements are checked to verify they are a variable statement and a return statement. Team member one utilized this test to verify a wide range of features worked properly. For example, the success of the function declaration, return statement, and print statement are all attributed to this test.

```
// Designed to test negative multiplication as well as division with a function
call.
@Test
public void parseFactorFunctionCallExpressionWithNegativeMultiplication() {
    FactorExpression expr = parseExpression("-1 * (2 / foo(3))", false);
    assertNotNull(expr); // Tests that function is properly parsed and assigned to
the "expr" variable
    assertTrue(expr.getLeftHandSide() instanceof UnaryExpression); // Tests that
first part of expression is a Unary Expression
    assertTrue(((UnaryExpression) expr.getLeftHandSide()).isMinus()); // Tests
that the Unary Expression is recognized as a negative number
    assertTrue(expr.getRightHandSide() instanceof ParenthesizedExpression); //
Tests that the second part of expression is a Parenthesized Expression
    assertTrue(((ParenthesizedExpression) expr.getRightHandSide()).getExpression()
instanceof FactorExpression); // Tests that there is a Factor
    // Expression inside the parenthesis
}
```

This test checks that a function call can be made in a nested negative factor statement. To verify this worked properly team member two designed these checks. First, they wrote some catscript code and passed that to the statement parser. Next, they verified that it was successfully parsed. They then checked if the left side of the expression is a unary expression and is negative. Lastly, they checked that the right side of the expression is a parenthesized expression and contains a factor expression. Team member one utilized this test to build the factor expression, unary expression, and parenthesized expression. They were able to successfully verify their solutions thanks to this test.

```
// Designed to test a function definition with a list of integers as a parameter.
@Test
public void functionDefinitionStatementWithParamAndBody() {
    FunctionDefinitionStatement expr = parseStatement("function x(a : object, b :
list<int>, c : bool) { print(b) }");
    assertNotNull(expr); // Tests that function is properly parsed and assigned to
the "expr" variable
    assertEquals("x", expr.getName()); // Tests that the function name is
```

```
identified as "x"
    assertEquals(3, expr.getParameterCount()); // Tests that the function has 3
parameters
    assertEquals("a", expr.getParameterName(0)); // Tests that the first
parameter's name is "a"
    assertEquals("b", expr.getParameterName(1)); // Tests that the second
parameter's name is "b"
    assertEquals("c", expr.getParameterName(2)); // Tests that the third
parameter's name is "c"
    assertEquals(CatscriptType.OBJECT, expr.getParameterType(0)); // Tests that
first parameter is of type Object
    assertEquals(CatscriptType.getListType(CatscriptType.INT),
expr.getParameterType(1)); // Tests that second parameter's type is list of
integers
    assertEquals(CatscriptType.BOOLEAN, expr.getParameterType(2)); // Tests that
third parameter is of type Boolean
    assertEquals(1, expr.getBody().size()); // Tests that there is one statement
in the function body
    assertTrue(expr.getBody().get(0) instanceof PrintStatement); // Tests that the
one statement in the function body is a Print Statement
}
```

In this last example, team member two wanted to check the function definition worked. To do this they used almost every parameter they could in a function. First, they passed this catscript function to the statement parser and verified that it was returned successfully. Next, they checked that the expression was called “x”. Now they could check if the parameters existed and were the right names. Next, they verified that each element was the right type of variable. Finally, they checked if the function would print the value of b. This test was vital in helping team member one flush out each of the catscript types and the function declaration statement.

In total both team members contributed significant time over the course of the semester to see the project was completed on time. In total the team has spent over 100 hours on the Catscript compiler project. They met regularly over the course of this project to ensure both members were making good progress and sought each other's counsel on roadblocks. Without this teamwork, this project would likely have not succeeded.

Design pattern

The design pattern for this project was memoization. Simply put, memoization is when a function remembers output from previous runs. This is done to reduce runtime and prevent a program from repeating processesit has already accomplished. For example, a system that looks up details about restaurants could keep a record of popular places to eat in a lookup table. This way it can quickly access that information ratherthan query a database and wait for a response. A simple place memoization was used in the catscript compiler is in the getListType function of the CatscriptType class. This function is used a lot to find thetype of an attribute in our receive descent algorithm. It should therefore be memorized to prevent any unnecessary compile time. Here is the code for this particular case:

```
private static Map<CatscriptType, ListType> cache = new HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
    ListType listType = cache.get(type);
    if (listType != null){
        return listType;
    }
    else {
        ListType newListType = new ListType(type);
        return newListType;
    }
}
```

Here is an effective use of memoization. Without the effective use of memoization, every time we got a list type we created a new ListType(). This could cause the code to be bogged down with creating and storinginformation that we may have already created. Memoization provides an eloquent and simple solution to this problem. First, make a hashmap of all the types already created. When the function is then called it canthen check this map for the type queried. If there is no match, then we can create a list type and add it to the map. This prevents duplicate types from taking up more space and time. Otherwise, the hashmap canprovide the type that is then given to the user. By caching this information in a map, an expensive process has been simplified and shortened to a much better state.

CatScript Guide

This document details the CatScript programming language and its many features.

Introduction

CatScript is a simple scripting language with many different features and functionalities. Here is an example:

```
var a = "Hello!"
print(a)
```

The above example uses a variable statement and a print statement to print out the word "Hello!". There are many other types of statements and expressions that can be used in CatScript.

CatScript is a statically typed language, with a small type system. Below are the types available in the CatScript programming language:

- string - a java-style string, enclosed in quotation marks
- int - a 32 bit integer
- bool - a boolean value (true or false)
- list< x > - a list of value with the type 'x', ex. list< int > is a list of integers
- null - the null type
- object - any type of value

Features

For loops

For loops can be used to iterate over an iterable data structure, such as a list. A for loop statement starts with the "for" keyword, followed by a parenthetical statement. Inside the parentheses, there is an identifier followed by the iterable data structure, either as a variable representing the data structure or the data structure itself. A body of at least one statement follows the closing parentheses, with the body enclosed in curly brackets. The statement(s) could be used to print variables, perform mathematical computations, call a function, and more.

The code below shows an example of a for loop with the CatScript formatting. The identifier in this example is x, with the value of x changing after each iteration. The data structure in this example is the list [1, 2, 3]. The value of x would first be 1, then 2 on the next iteration, and then finally 3 as the loop iterates over the provided list. The statement of the body of the for loop in this example is a print statement, which prints the value of x. In a nutshell, this for loop iterates over a list and prints the values, which means this loop would print the values 1, 2, and 3 separately.

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

If Statement

If statements can be used to test conditions and take separate branches depending on if those conditions are met or not. To test multiple conditions, you can use an if statement followed by an else-if statement. That way, if the first statement's conditions are not met, then you can test another set of conditions. If the conditions of an if statement are met, then the code inside the statement executes. Otherwise, it moves on, either to a following else-if statement, a following else statement, or simply move onto the next line of code outside of these statements. An if statement is started with the "if" keyword, followed by a condition in parentheses that will be tested. Then, there is at least one statement in the body of the if statement, which is enclosed by curly brackets. An if statement can be followed by one or more else-if statements, which follow the same formatting except with "else if" as the identifying keywords, but an else-if statement is not required. Similarly, an else statement can follow an if statement (or an else-if statement, if one has been included). An else statement is identified by the "else" keyword followed by the body of statements enclosed in curly brackets. There are no conditions being tested in an else statement.

The following example details an if statement followed by an else-if statement, which is then followed by an else statement. The if statement tests whether the value of the variable "x" is less than 5. If it is, then the program would print the value of x minus 2. If the value of x is greater than 5 and does not meet those conditions, the program moves onto the else-if statement. This statement checks if the value of x is less than 10. If it is, the program would print the value of x plus 4. If the value of x is greater than 10 and does not meet those conditions, the program moves onto the else statement. Here, there are no conditions to check. The program would simply print the value of x multiplied by 2.

```
if (x < 5) {
    print(x-2)
} else if (x < 10) {
    print(x+4)
} else {
    print(x*2)
}
```

Print Statement

A print statement prints the value that is enclosed in the statement. This kind of statement starts with the "print" keyword, followed by parentheses. Inside the parentheses can be any value that one wishes to print. A single value can be provided, as well as a variable representing a value. Mathematical computations can also be executed inside the parentheses, which would lead to the statement printing the result of the computation. A function can also be called inside the parentheses, which would print whatever value is returned from the function. To summarize, a print statement can be used to print a value, and there are many ways that that value can be represented.

The following example shows a print statement with a mathematical computation inside the parentheses. This statement would add together the values 6 and 4 and print the result. Therefore, this print statement would print the value 10.

```
print(6 + 4)
```

Variable Statement

A variable statement is used to define a variable to be used later in the program. This statement can be identified by the "var" keyword, followed by the variable name. Optionally, a colon and variable type can follow the variable name, but it is not necessary. An equal sign would be next, which is then followed by the value that is being assigned to the newly declared variable. This value can also be in the form of an expression, such as performing a mathematical computation in which the result would be assigned to the variable.

The following example details the declaration of the variable "a". The "var" keyword starts the statement, followed by the variable name, which in this example is "a". This example uses the optional type declaration as well. A colon follows the variable name "a", which is then followed by "int", signifying that the variable is an integer. After the equal sign is the value 5, so the variable's value is assigned to be 5. To summarize, the following example creates an integer variable "a" with a value of 5.

```
var a : int = 5
```

Function Call Statement

A function call statement calls an existing function. The format of this statement varies depending on the function that it is calling, but the statement starts with the name of the function being called followed by parameters separated by commas in parentheses. If the function does not require any parameters, then the pair of parentheses would be empty. Similar to the print statement, the parameters can be the values themselves, variables representing the values, results of another function call, and more. As previously stated, there are many ways that a value can be represented, so there are many ways that a value can be passed to a function via a function call statement.

The following example is a function call statement calling the function "addTwoNumbers". This function would have to have already been declared to successfully call the function (see Function Declaration Statement below). The number and type of the values passed into this statement must match the number and type of the parameters outlined in the function declaration. If a function is declared to accept only an integer and a string, the function call would need to pass in only an integer and a string separated by commas. The order of the parameters in the parentheses of the function call statement must match the function declaration as well. In this example, we are calling the function "addTwoNumbers" and passing the values 1 and 2 respectively. This statement would call the function, which would then execute with the parameters provided by this statement.

```
addTwoNumbers(1, 2)
```

Assignment Statement

An assignment statement is similar to a variable statement, as it is assigning a value to a variable. However, an assignment statement is used to assign a value to an identifier that has already been declared, such as by using a variable statement. An assignment statement starts with an identifier, such as a variable name. An equal sign follows, which is then followed by the value that is being assigned to the variable that is pinpointed by the identifier being provided in the statement.

In the following example, the string "Hello World" is being assigned to the variable "a" (which has already been declared). If the variable had a type identified when it was declared, the new value being assigned must match that type as well. Once the new value is assigned, the old value is essentially overwritten as it has been replaced with the new value.

```
a = "Hello World"
```

Function Declaration Statement

A function declaration statement is a statement that declares a function to be used later in the program. After declaration, a function can be called using a function call statement (see Function Call Statement above). A function declaration statement starts with the keyword "function" followed by an identifier, which is the name of the function. A pair of parentheses follows the identifier. If the function requires no parameters, the parentheses can be empty. Otherwise, a comma-separated list of parameters will be inside the parenthesis. Each parameter is represented by an identifier, which is basically a name assigned to the parameter to be used inside the function. Each parameter can also have a type attached to it by following the identifier of the parameter with a colon and the type associated with the parameter, but it is not required. A colon and a type can follow after the parentheses containing the parameters to specify a return type for the function, but this is again not required. The function body follows next, which is at least one statement enclosed in curly brackets. The statement(s) could be used to print variables, perform mathematical computations, call a function, and more. If a return type was specified for the function, then there will need to be a return statement that returns a value that matches that return type.

The following example declares a function with the name "functionName". This function requires two parameters, which is an integer and a string. The integer is referred to by the identifier "a", and the string is referred to by the identifier "b". This function has a return type of "int", which means it returns an integer. The function has two statements, a print statement and a return statement. Using the identifiers for the variables, the print statement would print the string value that is passed into the function and the return statement would return the integer value that is passed into the function. For example, if this function was called with the integer 5 and the string "hello", the string "hello" would be printed and the value 5 would be returned.

```
function functionName (a : int, b : string) : int {
    print(b)
    return a
}
```

Return Statement

A return statement can be used to return an expression, such as returning a value from a function. The return statement starts with the keyword "return" followed by an expression. There are many different types of expressions (see below), so there are many different uses for a return statement. This statement can return variables of any type. A mathematical computation can be performed in this statement, and the result of that computation would be returned. Comparisons can also be computed, with the statement returning true or false depending on the result. To summarize, a return statement returns an expression, which can come in many different forms.

The following example shows a return statement with a mathematical computation. This statement would take the values of variables a and b, add the values together, and return the result.

```
return a + b
```

The following example is a more simple example, as it only returns the boolean value "true".

```
return true
```

Equality Expression

An equality expression is used to compare two objects to see whether they are equal or not equal to each other. The expression is formatted by starting with one object (which can be an integer, a variable representing a value, a boolean, etc.) followed by either two equal signs to test equality or an exclamation mark and an equal sign to test inequality, which is then followed by the second object. These can be used in many places throughout a program, but they are commonly used in conjunction with if statements to provide a condition to test as it returns a true or false value.

The following equality expression is testing whether the two values are equal since it is using the two equal signs. In this case, it is testing whether the integers 10 and 7 are equal, and since they are not, this would be false.

```
10 == 7
```

This next example is testing inequality (note the exclamation mark and equal sign) between two objects. In this case, the objects are variables "a" and "b", so the values of those variables are being compared. If those two values are not equal to each other, this would be true. Otherwise, this would be false.

```
a != b
```

Comparison Expression

A comparison expression takes two objects, usually integers or variables representing integers, and compares them by testing if one is less than (<), less than or equal to (<=), greater than (>), or greater than or equal to

(>=). A comparison expression is similar to an equality expression in the way that there are many places that a comparison expression can be used throughout a program, but they are commonly used in conjunction with if statements to provide a condition to test since they also return a true or false value.

The following example is comparing the integer values 5 and 8 and is testing whether the integer 5 is less than the value 8. Since it is, this would return true.

```
5 < 8
```

This next example is comparing two integer values of 4 and testing whether 4 is less than or equal to 4. Since they are equal, this would be true.

```
4 <= 4
```

This next example compares the integer value 2 with the value of the variable "a". This one is testing whether the value 2 is greater than the value of the variable "a". The result of this would depend on the value of "a", but if the value 2 is greater than the value of "a", then this would return true. Otherwise, it would be false.

```
2 > a
```

This final example is comparing the integer value 9 with the value of the variable "b". This one is testing whether the value of the variable "b" is greater than or equal to the value 9. The result of this one once again depends on what the value of "b" is, but if the value of "b" is greater than or equal to the value 9, then this would be true. Otherwise, it would be false.

```
b >= 9
```

Additive Expression

An additive expression is used to add two objects together or subtract one object from the other, such as adding two integers, concatenating two strings together, or subtracting the value of a variable from an integer. The format of this expression is a plus sign between the two objects to add or concatenate them together, or a minus sign between the two objects to subtract one from the other. These expressions are commonly used in mathematical computations.

The following example is concatenating two strings together, "string1" and "string2". The result of this expression would be one string with the two given strings concatenated together. So, the result would be "string1string2".

```
"string1" + "string2"
```

This next example is subtracting one integer from another. In this case, the value 2 is being subtracted from the value 5, so the result would be 3.

```
5 - 2
```

Factor Expression

A factor expression is used to multiply together or divide two objects, such as integers or the values of variables. The format for multiplication is a star (*) symbol between the two objects, while the format for division is a slash (/) symbol between the two objects. These expressions are commonly used in mathematical computations.

The following example is multiplying two integer together. In this case, the value 5 and the value 4 are being multiplied together, which would result in the value 20.

```
5 * 4
```

This next example is taking an integer and dividing it by the value of a variable. The result of this would of course depend on the value of "a", but for example, if the value of "a" was 2, then the expression would be 8 divided by 2, which is 4.

```
8 / a
```

Unary Expression

A unary expression is very simple as it contains only one operand and a unary operator, such as a negative number or the operator "not" to flip a boolean value. A unary expression can come in many different forms.

The following unary expression is an example of a negative number, in this case a negative 1. The unary operator is the minus sign to turn the operand (the value 1) negative. The value 1 could also be replaced by a variable to turn the value of the variable negative.

```
-1
```

This next example takes the boolean value "true" and essentially flips it so that the value is actually "false". In this case, the unary operator is the term "not" and the operand is the boolean value "false".

```
not true
```

Primary Expression

There are many types of expressions that fall under the umbrella of a Primary Expression. Below you can find descriptions and examples of the various types.

Integer Literal Expression

An integer literal is an integer whose value is directly represented and fixed. The following example of an integer literal expression shows the value 10, which is an integer literal. A variable with an integer value of 10 is not an integer literal, but the integer 10 itself is an integer literal.

```
10
```

String Literal Expression

A string literal is a fixed direct representation of a string. Again, a variable with a value of a string is not a string literal. Only a string itself is a string literal. A string literal is surrounded in quotation marks, as shown in the string literal expression example below.

```
"Hello"
```

Boolean Literal Expression

A boolean literal is one of two keywords, either "true" or "false". Similar to the integer literal and string literal expressions, a boolean literal is a direct representation of a fixed boolean value. The following example shows a boolean literal expression.

```
true
```

List Literal Expression

A list literal is the direct representation of a fixed list of expressions, which can be singular objects such as integers or more complex. The list itself is a set of square brackets with the list objects inside, separated by commas. Below is an example of a list literal expression.

```
[1, 2, 3]
```

Null Literal Expression

A null literal is the fixed and direct representation of the value "null". Again, a variable that is set to null is not the same as a null literal. Below is a null literal expression.

```

```

```
null
```

Parenthesized Expression

A parenthesized expression is simply an expression inside a pair of parentheses. This allows for parenthesized expressions to have higher precedence and be executed first. Below is an example of a parenthesized expression, with an additive expression inside the parentheses.

```
(1 + 2)
```

Identifier Expression

An identifier expression is simply an identifier keyword, mostly used to represent a variable. Below is an example of an identifier expression where the identifier is "x".

```
x
```

Sequence Diagram

Figure 1: The sequence diagram for the Catscript recursive descent parser parsing the code "1 != 1 + - 1"

This project utilized the recursive descent parser to great effect. Above is the sequence diagram of the Catscript parser parsing the code: “1 != 1 + - 1”. This will serve as a template for an exploration of how this parser algorithm works. To start, the tokenizer converts the code into its fundamental tokens. This is used to make a token list that is then passed to the parser. These steps are not shown in the diagram but are important to understanding how the Catscript parser works. Once the parser receives the token list, it begins to descend through the grammar and attempt to match the tokens to the rules encountered. At first, the parser will descend to the primary expression. This is done because expressions cannot start with ‘!=’ or ‘+’. In this case, it will identify the integer ‘1’ and consume its token. It will then return this value by walking up the tree. Each time the integer is returned, the function checks if the next token matches its rule. For example, the parseUnaryExpression function will check if the next token is a ‘!’ or ‘-’. Because ‘!=’ is not either of these, the expression is simply handed up the chain. Once the expression has reached the parseEqualityExpression function the token will match the rule. The function will consume the token and recursively call itself. This process will start over again without the ‘1’ and ‘!=’ tokens. The parser will once again find an integer ‘1’, consume it, and return the expression. This time the parseAdditiveExpression function will match with the ‘+’ token and trigger another decent. The parseUnaryExpression will match with the ‘-’ token and recursively call itself. This is done to support operations like “!!true” or not not true. Once all of these have been consumed, the parser will descend and find the last ‘1’ integer token. This whole expression is then walked back up to the parseExpression function and returned.

The Catscript parser has a lot of steps to parse even the most simple expressions. However, when debugging recursive descent proved invaluable to diagnosing problems. This is because recursion naturally works well with grammar and is difficult to break. Language has always had recursive elements to it. Statements like “Great-great-grandfather” and “re-re-write” are perfectly normal sentences. Yet, they demonstrate how natural recursion is used in everyday speech. This naturally has been extended to programming grammar. As stated before, ‘not not true’ is something that can be used in any programming language. While it can be parsed in a variety of ways, recursion follows a natural flow of logic. Taking each element and using it to call upon itself can make the flow of events more human and linear. The human factor is attributed to the simplicity of debugging possible. When an issue arose, just following the logic of the grammar often pointed to the issue. Additionally, recursive descent makes the compiler naturally robust to perform these repetitive operations. No matter how many times a ‘not’ is added, the recursive function will guarantee the expression is parsed properly. It will also naturally look for missing attributes. An expression cannot start with a factor or additive expression by itself. By descending to the bottom at the start, the Catscript parser will naturally throw an error if something unexpected appears in the token list.

Design trade-offs

The recursive descent parser was chosen because it is widely used in industry, is a simple way to follow the natural flow of grammar, and is easy to follow for debugging. However, many other compilers utilize the parser generator. A comparison of these two methods is warranted to understand why catscript was designed to utilize recursive descent. Here we will utilize the ANTLR parser generator to demonstrate how a parser generator works.

Rather than recursively going through a given statement, a parser generator is a separate program that takes a language specification and generates a parser for that specification. This is typically done with a '.g' or grammar file. This file starts with the keyword 'parser grammar' and then the name of the grammar. Next, there are options the user passes to the generator. This starts with defining the language the user is programming in. The next option is to tell the generator to produce an abstract syntax tree or parser tree. Then the user must define a superclass to allow ANTLR to implement helper methods. This is where one of the first problems with code generation is run into. When the Java compiler takes the grammar file in, it has to parse and convert it into Java code. This requires the use of hooks into the target language in the grammar file. In this case, we utilize a superclass as it is something often utilized in Java.

After all the options are set, the rules of grammar can be provided to the grammar file. This is done much simpler than recursive descent. Here, a grammar name is defined as well as what it will look like. In the case of Catscript, it was determined that the grammar could be easily added as it was very well defined. However, it quickly became apparent that this simplicity of implementation would lead to hardships later. An important part of this project was the use of test-driven development. To utilize this, regular debugging is required to see why tests are not passing. The parser generator could generate code that is far more complex than a simple recursive descent and utilize operations that the team was not familiar with. Additionally, it is far easier to navigate a parser if there is a better understanding of how it works. It is for these reasons, that it was determined that building the parser from scratch would lead to advantages later on in the project.

The use of the generated parser would not have been too dissimilar to recursive descent. Both tokenize the code source and pass it to their respective parser. They then both initialize their starting function. This is where the changes diverge, however. The recursive descent starts walking down the syntax tree to find the grammar the tokens match. The generated parser can quickly make an abstract syntax tree. This is done quicker than recursively stepping through a grammar. However, this generated tree is something that cannot be easily modified. Generated code is hard to modify and is likely to be regenerated over time. This would have made the project more difficult as time went on and tests became more complex.

It should be apparent that the advantages of utilizing recursive descent to parse statements and follow grammar. While it is far easier to build a parser generator, the debugging process would have been much more time-consuming. This is due to the team having to navigate an unfamiliar codebase and regularly look up operations the generator used. Additionally, the loss of control over what the abstract syntax tree looked like after being parsed would have made implementing more complex features troublesome.

Software development life cycle model

Test-driven development was essential to providing direction for development and ensuring the compiler operates correctly. Test-driven development is defined as an iterative approach to developing code where a programmer is focused on passing feature tests. This differs from traditional development which focuses on producing a product or program.

As mentioned before, tests were provided by team member two. These ranged from simple token tests to far more complex function assignments and calls. These tests were broken up into four categories; tokenizer, parser, eval, and bytecode. The tokenizer tests all focused on the tokenizing aspect of the compiler. This included a wide range of tokens and white space tests. These were done by passing a string containing tokens to a test that would then run the string through the tokenizer and return a list of tokens. If these tokens did not match the expected input, then the test failed. The parser tests tested the recursive descent parser in Catscript. The symbols, types, statements, and expressions were all tested. The iterative approach was still used to create simple tests and build up to more complex tests. From duplicate names to function declaration expressions were all diligently tested here. The eval category was made to check the more complex execution of code. Here entire functions and statements were built and utilized to verify the functionality of Catscript at the execution stage. Lastly, the bytecode category was for testing whether the transition from Catscript to machine-readable code occurred properly. At this stage, the compiler was strong enough to be giving actual readable values when tests were executed. As the compiler became more powerful and complex, the tests improved as well. It is for this reason regular checks were performed on the categories when the code they tested was modified. By the end of the project, the tests were able to pass full Catscript programs to the compiler and get a result like running code on any other compiler.

There are a lot of factors that contributed to the want to utilize test-driven development. Most important are the human factors. It can be frustrating to not see progress when working on a large project like this. Even if progress is being made, validation can keep morale high and give regular encouragement to make progress. Having tests slowly start passing over time is just the right motivation required to succeed in this project. These short obtainable goals broke apart an insurmountable project into a list of manageable tests. Another important effect of test-driven development was maintaining accountability. As mentioned before, there was a large range of tests produced by team member two. This way, the solutions that were provided could be verified to operate how they are intended to be implemented. A good example is recursive functions. The first solution presented for functions operated perfectly fine, but was unable to handle recursive calls. This would not have been noticed if there was not a test pushing the boundaries of the Catscript programming language. Lastly, test-driven development allowed for better debugging. The recursive parser would have been hard to implement if it was not for a range of tests that diagnosed every aspect of it. The ability to step through the execution of code allowed team members to slowly work around complex logic. Additional tests could also be developed to check what parts of the test code were breaking the compiler or causing issues. This allowed team member one to dive deeper into any problems that arose.

Without the use of test-driven development, it is likely this project would have faced serious delays if it was completed at all. The testing provided a framework in which the project could be developed, managed, and diagnosed. It also provided small, obtainable, and verifiable goals for the team to work towards. These factors were monumental in assisting in the production of Catscript. It will likely be utilized to great effect in future projects as well.