Section 1: Program

My source code can be found in `source.zip` alongside this pdf.

Section 2: Teamwork

For this project, my team was able to effectively split up the work load to create an expansive and complete final product. Team member 2 created tests (example provided below) to ensure the implementation was correct, and team member 1 fixed anything that was incorrect based on those tests. Additionally, team member 1 spent the majority of their time implementing the program details based on the tests written by team member 2. Team member 2 also provided the Catscript documentation. The team was able to work effectively on the project and produced a robust final result.

```
public class Test extends CatscriptTestBase {
    @Test
    void listLiteralsAndFunctionReturnsAndForStatementsExecuteProperly() {
        assertEquals("foo, [1, 2, 3], bar, \n", executeProgram(""
                + "var foo : list = [\"foo\", [1, 2, 3], \"bar\"]\r\n"
                + "function foobar(foo2 : list<object>) : string {\r\n"
                + "    var bar = \"\"\r\n"
                + "    for(x in foo2) {\r\n"
                + "        bar = bar + x + \", \"\r\n"
                + "    }\r\n"
                + "    return bar\r\n"
                + "}\r\n"
                + "print(foobar(foo))"));
    }
}
```

Section 3: Design pattern

In this project, we used memoization. The following code can be found in: src/main/java/edu/montana/csci/csci468/parser/CatscriptType.java.

```
public class CatscriptType {
    static HashMap<CatscriptType, ListType> cache = new HashMap<>();
```

```
    public static CatscriptType getListType(CatscriptType type) {
        ListType listType = new ListType(type);
        if (cache.containsKey(type)) {
            return cache.get(type);
        } else {
            cache.put(type, listType);
        }
        return listType;
    }
}
```

This pattern was used to ensure that a new `ListType` will only be created if a `ListType` of the input type does not already exist. If there is a Catscript program that utilizes 5 different lists, all of type int, the same `ListType` would be used every time the getListType() method was called. This design pattern not only has beneficial applications, but also requires less code to be written so that the product can be deployed more efficiently without cutting corners.

This results in some useful benefits, including a faster runtime and less memory usage. Additionally, even though the number of potential inputs is finite and currently known, memoization allows for attentional types to easily be integrated into the program structure.

Section 4: Technical writing.

Catscript Guide

Introduction

Catscript is a simple scripting langauge. Here is an example:

```
var x = "foo"
print(x)
```

Features

Print statements

In Catscript, you can print output to the console using the `print` statement, as shown below:

```
print("Hello world!")
```

Output:

```
Hello World
```

Variables

To declare a variable in Catscript, use the `var` keyword:

```
var a = 1
```

Variables in Catscript are statically typed. Rather than requiring a type declaration for each variable, however, Catscript simply requires a variable to be initialized upon declaration. The type of the variable is inferred from the expression assigned to the variable:

```
var y = 9       // produces an int type
var z = "hello world"   // produces a string type
```

If you would like to declare the type of your variable, use a colon (`:`) after the variable name:

```
var b : bool = true
```

The data types offered by Catscript are: `int` , `string` , `bool` , `object` , and `list<>` .

List types can be entirely inferred, or they can have only their component type inferred, or they can have their entire type declared:

```
var list1 = [1, 2, 3]   // produces a list<int>
var list2 : list = ["one", true, 3]     // produces a list<object>
var list3 : list<string> = ["foo", "bar"]       // produces a list<string>
```

Assignemnt statements

In Catscript, you can reassign variables to new values using the variable name followed by an equals sign (=) and then the desired value:

```
var x = 5
x = 1   // The variable x is reassigned to the value 1
```

Note that the type of the variable must remain consistent in the reassignment, since Catscript is a statically typed language:

```
var foo = "bar"
foo = "hello world"     // This is fine
foo = 20        // Error: strings are not assignable from ints
```

If statements

In Catscript, if statements are used for conditional flow control. An if statement tests a boolean expression and executes the statements in the `if` block if the expression is true, and the `else` block otherwise. Intermediate boolean expressions can be tested using `else if` blocks.

To write an if statement in Catscript, use the `if` , `else if` , and `else` keywords. Every `if` and `else if` is immediately followed by their respective boolean expression, enclosed in parentheses. Statement blocks associated with each `if` , `else if` , and `else` are then included within curly braces. An example is shown below:

```
var a : int = 1
var y : int = 9
if(a == y) {
        print("a and y are equal")
} else if(a < y) {
        print("a is less than y")
} else {
        print("a is greater than y")
}
```

Output:

```
a is less than y
```

For loops

A for statement in Catscript iterates over the values in a given list. The statements within the `for` block execute for every value in the list. After each time these statements execute, the program returns to the top of the loop, checking whether there are more items in the list. If it finds another value, it will execute the `for` block again, continuing to loop in this way until the items in the list are exhausted.

You can use for loops to iterate over a list using the `for` and `in` keywords, specifying a variable name to represent the current value from the list being used in the loop, as well as the list to be iterated over. This is followed by the statements you wish to execute, enclosed in curly braces. An example is shown below:

```
var list1 : list = ["a", 2, false]
for(x in list) {
        print(i)
}
```

Output:

```
a
2
false
```

Functions

Functions in Catscript are used to define a block of statements that can be executed whenever that function is called. Functions are defined using the `function` keyword, followed by the name of the function, the parameter list in parentheses, then an optional return type, and finally the body of statements within curly braces.

Here is an example of a Catscript function without a return type:

```
function foo() {
        print("bar");
}
foo()
```

Output:

```
bar
```

To add a return type to your function, use a colon (`:`) after the parameter list:

```
function foo2() : string {
        return "bar"
}
var result = foo2()
print(result)
```

Output:

```
bar
```

When your function has arguments, you can simply list the parameter names, or you can declare the type of the parameter as well:

```
function foo3(num : int, value) {
        var step = "Step " + num + ": " + value
        return step
}
var result = foo3(2, "eat a cookie")
print(result)
```

Output:

```
Step 2: eat a cookie
```

Return statements

In Catscript, return statements are optional in functions with void return types but required in functions with non-void return types.

To return from a void function, simply use the `return` keyword:

```
function foo() {
        return
}
```

To return a value from a function, use the `return` keyword, followed by the value you wish to return:

```
function foo() : string {
        return "bar"
}
```

Equality expressions

In Catscript, equality expressions are used to determine whether or not the values of two expressions are equal. They resolve to a boolean (`true` or `false`) value.

To return true for equal values and false otherwise, use the `==` operator:

```
var a = 1
var b = 1
var c = 0

a == b  // evaluates to true
a == c  // evaluates to false
```

On the other hand, to return false for equal values and true otherwise, use the `!=` operator:

```
var a = 1
var b = 1
var c = 0

a != b  // evaluates to false
a != c  // evaluates to true
```

Comparison expressions

Comparison expressions in Catscript are similar to equality expression in that they compare two expressions and evaluate to a boolean value.

To determine if an expression is strictly less than or greater than another expression, use the `<` or `>` operator, respectively:

```
var x = 9
var y = 11

x < y   // evaluates to true
x > y   // evaluates to false
```

To determine if an expression is less than or equal to, or else greater than or equal to, another expression, use the `<=` or the `>=` operator, respectively:

```
var m = 7
var n = 7
var o = 8

m <= n  // evaluates to true
m <= o  // evaluates to true

n >= m  // evaluates to true
n >= o  // evaluates to false
```

Additive expressions

In Catscript, additive expressions can be used to add and subtract numbers or to concatentate strings.

When working with numbers, use the `+` operator to add and the `-` operator to subtract:

```
print(3 + 6)
print(7 - 3)
```

Output:

```
9
4
```

When working with strings, Catscript will automatically treat the additive expression as string concatenation if either the left- or right-hand side expression evaluates to a string:

```
var c = "cake"
print("the " + cake + " is a lie")
```

Output:

```
the cake is a lie
```

Factor expressions

Factor expressions in Catscript are used to multiply and divide numbers. Use the `*` operator to multiply and the `/` operator to divide:

```
print(6 / 3)
print(7 * 3)
```

Output:

```
2
21
```

Unary expressions

The only unary expressions in Catscript are negations. To negate a boolean expression, use the `not` operator, and to negate a number, use the `-` operator:

```
var x = 5
print(-x)
```

```
var y = true
print(not y)
```

Output:

```
-5
false
```

Primary expressions

In Catscript, primary expressions include identifiers, literals, function calls, and parenthesized expressions.

Identifiers are the names of variables and functions used in the code, such as in assignment statements and function calls.

Literals are hard-coded values in the code. These can be strings (enclosed in either single or double quotes), integers, boolean values (`true` or `false`), the `null` value, or a list literal (a list of comma-separated values within square brackets). For example:

Function calls consist of a function name followed by the list of arguments in parentheses. This is how you execute the code within a function definition. Here is an example:

```
function foo() {
        // function body
}
foo() // executes the code in the functio body
```

Parenthesized expressions are used to specify operator precedence:

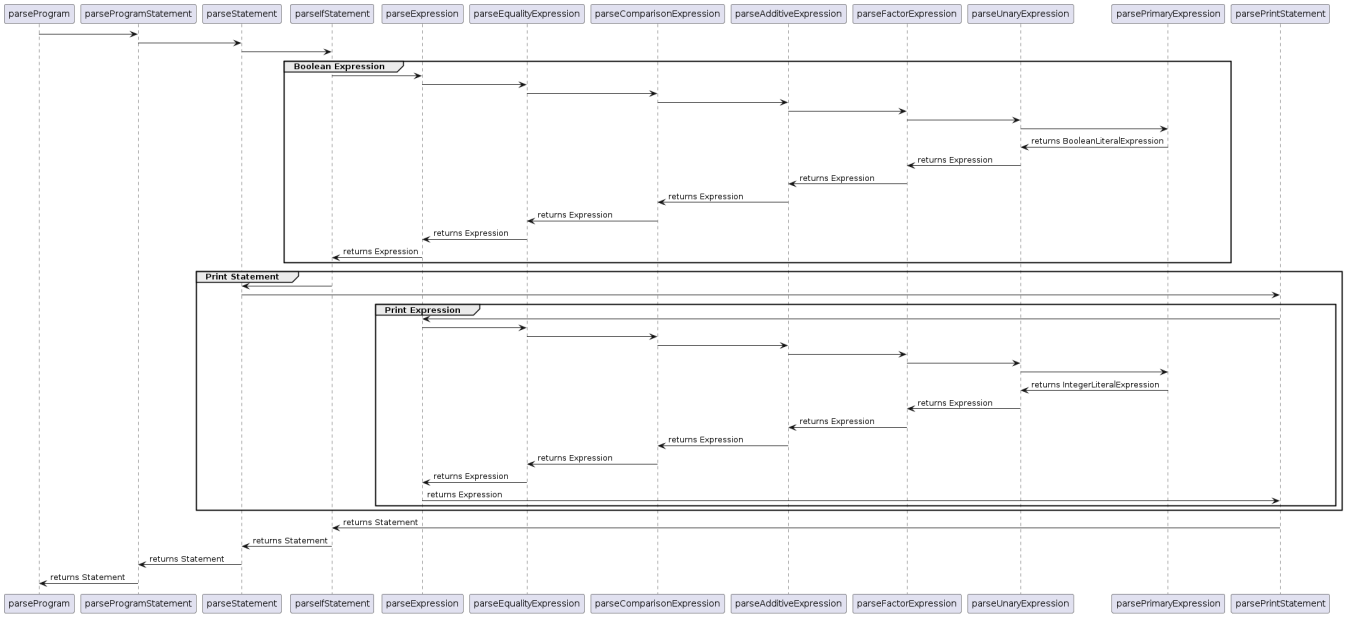
```
var x = 5 + 2 * 2
var y = (5 + 2) * 2

print(x)
print(y)
```

Output:

9
14

Section 5: UML.

This is a sequence diagram for parsing "if (true){print(1)}". It displays recursive decent in a very clear and visual manner. Each time a new statement or expression is created, the parser uses recursion to travel down the tree until it lands on the correct token type, and in true recursion form, returns the result all the way back up the tree. This is utilized throughout the parser for both statements and expressions to create a complete recursive descent parser.



Section 6: Design trade-offs

Two of the main methodologies that are used for creating a compiler are `recursive descent` and `code generation`. Despite these both being relatively popular, one clearly outshines the other. Recursive descent is the algorithm that was chosen for this particular project, over all the other potential methodologies.

Recursive descent was chosen for an array of reasons, among which are certain perks of the algorithm and the downfalls of other methods like code generation. Recursive descent parsing allows for the direct translation of grammar rules into code, resulting in code that closely mirrors the structure of the grammar. This direct correspondence enhances code readability and maintainability, as developers can easily trace the logic of the parser back to the grammar rules. Additionally, this makes it much easier for a user of the language to understand exactly what is going on under the hood of the program. With code generation, this task is either extremely difficult, or nearly impossible. Code created by a generator is very difficult to debug, which is a stark contrast to handwritten code, and that makes maintenance and creating new functionality an arduous task. Adding new features within the context of a recursive descent parser is relatively easy, and it will also sharpen the skills of the developer rather than sharpening their skills with low use case tools. That reality notwithstanding, the ability to interact with the codebase in a meaningful way is by itself almost enough reason to choose recursive descent when creating a compiler. That fact, though, is not the only thing that recursive descent has going for it. A handwritten recursive descent tokenizer will almost always be substantially shorter than a program created by a code generator. There is no need for programs to be as complicated as code generators make them, and the ability to read and understand code because it is handwritten is invaluable.

For our project in particular, implementing the compiler with this methodology was immensely helpful. It allowed us to conceptualize the project as a whole while implementing each individual piece of the puzzle without getting too lost in the details. The algorithm was specifically helpful while writing the parser. Being able to debug and visually see each element as it was added not only made the programming fun and fulfilling, but also allowed us to develop an intricate understanding of the project as we went. This is a beautifully simple but effective algorithm that can complete the task at hand in a good, clean manner.

Recursive descent was chosen for this project for many reasons, and the reality that it was implemented by hand means that we have been better equipped for the real world and solving the problems that we will be met with in a reasonable manner, with reasonable methodology.

Section 7: Software development life cycle model

Just as there are multiple different methods for designing a compiler, there are multiple different ways to go about implementing the actual code for that compiler. For this project my team used test driven development (TDD). This method can be implemented by writing test code, or code that you run against your codebase to see if it runs as expected. One subset of this is what is called test first development. The important distinction is that test first development is done by writing your tests before any production code is completed. In this situation, the development process is driven by the tests themselves, as opposed to unit tests, integration tests, or end to end tests. Test first development is what was used for this particular project, and although it is unlikely that I will utilize it very often in the future, it was effective in this particular application.

This method of development was immensely helpful for our team throughout the course of the project. First and foremost, it helped ensure that the code we wrote was high quality and would work reliably. Having our tests before the code was written allowed us to have a clear line of success that outlined how the system was expected to behave. It also helped with identifying any potential bugs early on in the life cycle of the code. This was especially important because each new layer of the compiler completely relied on the previous layer; the parser could not function without the tokenizer. Because of our tests, most of the bugs that revealed themselves later in the life cycle were not only easy to find with an effective debugger but were also easy to fix because the code was already fairly robust.

It also helped our team to work on smaller portions of code without getting lost in the large scope of the project as a whole. The tests broke the codebase into very manageable portions that were easy to complete correctly and integrate into the project. The code we wrote was cleaner because we were not overwhelmed with the whole system and was more maintainable because we could debug in chunks. We also realized that if we were producing a real product, this method would allow us to continuously integrate the features without being overly afraid of sizeable issues in the production system.

My conclusion about this method of programming is, clearly, very positive. Despite this, it is likely that I will utilize TDD methods like integration tests in the future. While in this specific situation the test first method was very helpful, it seems that grasping the entire scope of a project in a professional environment would be much more difficult. Without having a more concrete picture of implementation details, writing tests to effectively demonstrate the correctness of the code seems overly ambitious. That is not to say that test first development will never have a place in my personal professional career, nor am I claiming that it should not be used in other contexts. I simply am claiming that, according to my current understanding, having at least a rough baseline would be extremely helpful for creating effective tests.

Overall, TDD was an extremely effective model for this project. We were able to produce high quality code that was robust enough to facilitate bug fixes that arose from future code with ease. This model forced us to split up the code into ‘bite sized chunks’ so that we would not get lost in the large scale of the project. Using test first development, we found that meeting the requirements of the project was attainable, and could be achieved with clean, effective, and well-integrated code. Therefore, we were not forced to combine different portions of the project utilizing nonconventional methods and sloppy integration. TDD provided a structured and disciplined approach to this project and created a fitting end to my formal education in software development (at least for now).