Compilers Portfolio

CSCI 468

Mike Kadoshnikov, Jacob Tanner

Section 1: Program

Please include a zip file of the final repository in this directory. Zip up the source files and place it in the GitHub as source.zip.

The program uses Java to iterate through the recursive descent parser and uses the JVM Bytecode in order to complete compilation actions.

Section 2: Teamwork

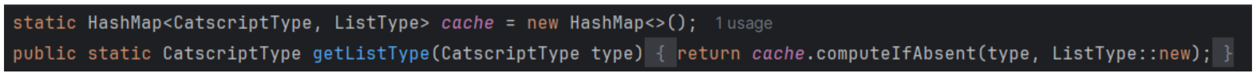
Team Member 1 (90%):

This team member primarily worked on the compiler and got the tests passing. They worked through evaluate, compile, statement, and expression parsing. The parts of the compiler that were developed were Additive Expression, Boolean Literal Expression, Comparison Expression, Equality Expression, Factor Expression, Function Call Expression, Identifier Expression, Integer Literal Expression, List Literal Expression, Null Literal Expression, Parenthesized Expression, String Literal Expression, and Unary Expression. The statements that were developed were Assignment Statement, For Statement, Function Call Statement, Function Definition Statement, If Statement, Print Statement, Return Statement, and Variable Statement. These were developed as independent classes, but the majority of the work was done in the CatScript Parser, where the statement and expression parsing were implemented.

Team Member 2 (10%):

This team member generated the technical writing document for the portfolio as well as the 3 tests to run on the compiler. The technical writing document consisted of a guide for programming the CatScript programming language. This guide is designed to help users to start programming in the language. It should consist of examples and the format of the expressions and statements. The three tests were done to verify that the compiler is implemented and works, but also to gain experience with writing tests for Test Driven Development.

Section 3: Design Pattern



```java
static HashMap<CatscriptType, ListType> cache = new HashMap<>();  1 usage
public static CatscriptType getListType(CatscriptType type) { return cache.computeIfAbsent(type, ListType::new); }
```

We have a function in the CatScriptType called getListType, this function takes advantage of the memoization design pattern. Which is used to speed up programs by storing the results of expensive functions and returning the cached result when the same inputs occur again. Rather than initializing a new list type, every time we get a list type we cache the result and return the list type from a map so we only have to initialize a List Type once and return the same results. The reason we use this pattern is due to the possibility of same type arrays being initialized, and rather than doing duplicate initialization we reduce the time complexity by grabbing the cached version. Although not noticeable in small use cases, when large projects are made the time difference should be noticeable.

Section 4: Technical Writing Document

This document is Catscript.md, the document should be a guide for the CatScript language. The contents of the document are provided below with the tests:

```
// test 1
function sumList(lst: list<int>): int {
   var sum: int = 0
   for (val in lst) {
      sum = sum + val
   }
```

```
    return sum
}

// test 2
print(sumList([1, 2, 3, 4]))

function contains(lst: list, x: object) {
    for (val in lst) {
        if (val == x) {
            return true
        }
    }
    return false
}

// test 3
print(contains([1, 3, 6], 2))

function recursion(num: int): int {
    if (num == 0) {
        return 0
    } else {
        return 1 + recursion(num-1)
    }
}

print(recursion(10))
```

CatScript Guide

This is an introductory guide to CatScript.

It includes information on all the expressions and statements that are included in the scripting language.

Introduction

CatScript is a simple scripting language. Here is an example:

```
var x = "foo"

print(x)

// this is a comment in CatScript
```

Expressions

Additive expression

The additive expression allows for addition or subtraction of integer literals or integer variables.

It evaluates to an integer value.

```
1 + 3 // evaluates to 4

5 - 1 // evaluates to 4
```

String concatenation can be also done with an additive expression.

```
"Cat" + "Script" // evaluates to "CatScript"
```

Factor expression

The factor expression allows for multiplication or subtraction of integer literals or integer variables.

It evaluates to an integer value.

```
4 * 4 // evaluates to 16

6 / 3 // evaluates to 2
```

Equality expression

Equality expressions allow for comparison of all CatScript data types.

It evaluates to a boolean value.

```
"Hello" == "Hello" // evaluates to true

1 == 3 // evaluates to false

"Hello" != "Bye" // evaluates to true
```

Comparison expression

Comparison expressions allow for greater than, less than, greater than or equal to, and less than or equal to comparisons of integer literals or integer variables.

It evaluates to a boolean value.

```
10 > 5  // evaluates to true

9 >= 10 // evaluates to false

4 <= 4 // evaluates to true

4 < 9 // evaluates to true
```

~~~
```
~~~

Unary expression

Unary expressions allow you to invert the value of a boolean with the keyword ```not``` or make the value of an integer negative with ```-```.

```
not true // evaluates to false

-54
```

Primary expression

Primary expressions include literal values for strings, integers, booleans, null values, and lists.

They also include variables, functions calls, and parenthesized expressions.

```
x

"Hello world!"

134

true

false

null

[1, 3, 5, 6]

foo()

(5 + 7)
```

Type expression

Type expressions allow you to specify the type of a variable, a function parameter, and the return type of a function.

It is not necessary to specify the datatype for variables or function parameters in CatScript.

List types can have a specified type or remain general lists.

```
int

string

bool

object

list // list type is not required but can be defined as such list<int>, list<string>, list<bool>, list<object>
```

List literal expression

You specify a list using a list literal expression.

Lists in CatScript are immutable, which means they cannot be modified.

```
[1, 3, 5, 6]
```

Statements

Print statements

Print statements allow for output of integers, booleans, strings, lists, and objects.

Print statements in CatScript automatically add a newline character ```\n```.

```
print("Hello World!")

print(5)
```

Variable statement

Variables can be initialized using the ```var``` keyword.

A value must be initialized with the new variable, but you may initialize with a null value.

You can also specify a variable type, but it is not necessary.

```
var x = "Hello"

var y: int = 4

var z: int = null
```

Assignment statement

Variable values can be changed by specifying a variable name and value int the format ```name = value```.

You cannot change a variable to an incompatible type after it has been given a value, even if an explicit type is not specified.

```
var x = "Hello"

x = "I have changed"

x = 5 // this will throw an incompatible type error
```

For loops

For loops allow you to iterate through a list of items.

You must specify a variable name, then the ```in``` keyword, and a list to iterate through.

In a for loop, the variable represents each element within the list as the loop progresses through it.

```
var lst = [1, 2, 3]

for (i in lst) {
```

```
    print(i)
}
```

Output:

```
1
2
3
```

If statement

If statements allow you to evaluate a conditional or boolean expression, running the code only if the condition evaluates to true.

Else statements allow you to run code if the condition evaluates to false.

There are no else if statements, but nested if statements are allowed in CatScript.

```
if (<condition>) {
    print("true")
} else {
    print("false")
}
```

```
if (<condition 1>) {
    print("condition 1 passed")
    if (<condition 2>) {
        print("condition 2 passed")
    } else {
```

~~~
        print("no conditions passed")
    }
}
```
~~~

Here is a real world example:

```
var x: int = 5
if (x >= 3) {
    print("x is greater than or equal to 3")
} else {
    print("x is less than 3")
}
```

Output:

```
x is greater than or equal to 3
```

Function definition statement

Function definition statements define a function and the parameters it requires.

You do not need to specify the type of the parameters, but you must specify a return type, or the function will be void.

Functions in CatScript can be recursive.

```
function hello() {
    print("Hello")
}
```

```
function fun(x, y: string, z: string): int {
    print(x)
    print(y + z)
    return 9
}
```

Here is an example of a recursive function in CatScript:

```
function recursive(curr: int, stop: int) {
    if (not (curr >= stop)) {
        print(curr)
        recursive(curr+1, stop)
    }
}

recursive(0, 3)
```

Output:

```
0
1
2
```

Return statement

Return statements allow you to return a value from a function.

You must ensure that the value you are returning is of the function's return type.

```
function add(x: int, y: int): int {

   return x + y

}
```

```
function stringConcat(x: string, y: string): int {

   return x + y // this will throw an incompatible type error

}
```

Function call statement

Function call statements call a function and pass parameters to it.

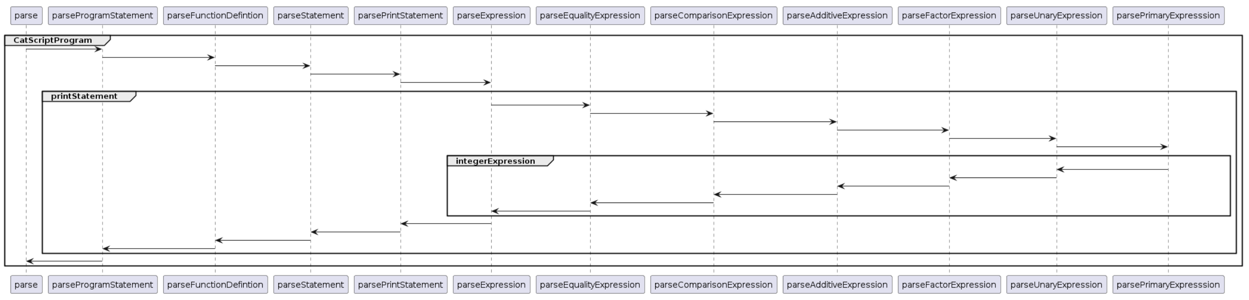
When calling a function, it's important to provide parameters that match both the expected data type and the required number of parameters specified by the function definition statement.

```
hello()


add(1, 5)
```

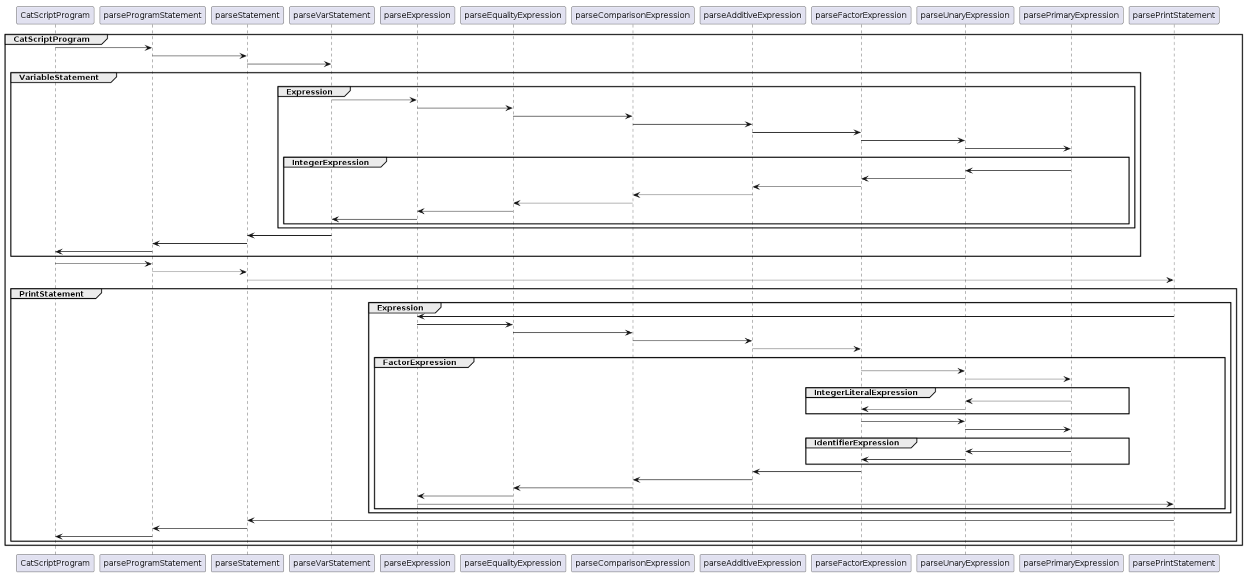
Section 5: UML

This sequence diagram shows the recursive descent parsing of a print statement. Specifically the parsing of “print(10)”.



For this simple statement we can see within the CatScript program we have a printStatement that should handle the parsing of the elements contained within, and since this simple example only includes the integer 10, it will parse it as an IntegerExpression, then return it to the print statement to be added as the internals.

To further illustrate the recursive nature I will provide a diagram for the parsing of: “var input = 5\n print(5 * input)”



This example is more complicated as we initialize a variable, then print that variable after it is multiplied, so the variable statement propagates down until we need to parse the expression, which continues down until it is caught as an IntegerExpression then returned up to the variable statement. Then the program continues to the PrintStatement which contains the multiplication, so the statement continues until the parseExpression, which then propagates to the FactorExpression where the multiplication is caught. It then parses the expression contained in the FactorExpression, which contains an IntegerLiteralExpression for the value of 5, then it catches the IdentifierExpression for the variable we had previously set. Then it returns these values to the FactorExpression where it is added as attributes then the FactorExpression is returned as the expression in the PrintStatement. Finally, the PrintStatement is added to the overall CatScriptProgram.

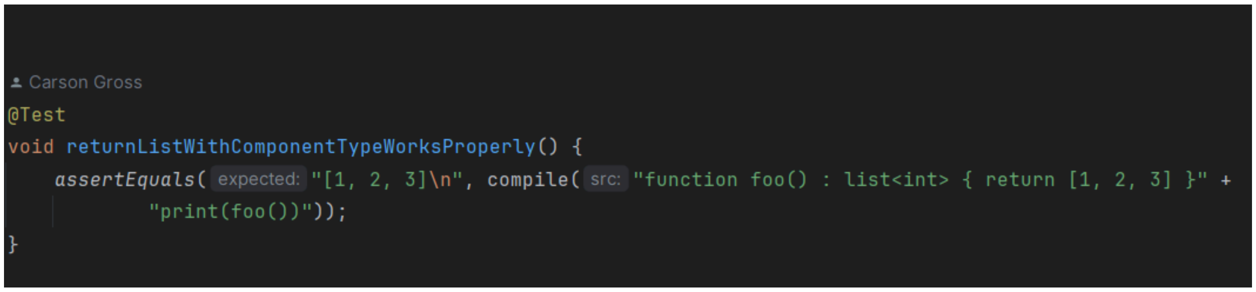
Section 6: Design Trade-offs

The design trade-off for this project was discussing a recursive descent parser and a parser generator. The recursive descent parser is a better project to implement as it reflects how the industry makes parsers. Given that recursive descent parsers give students a better feel about the recursive nature of grammars by having them work directly with the parser, they also have to implement the EBNF nature directly with code. The recursive descent parser also allows for students to get hands on with object bytecode and to see how the stack is managed. Although parser generators do have some benefits to consider; they are less code than doing it by hand, and there is less infrastructure to get right so there is less to worry about. Although they do have some upsides, generally the idea of a recursive descent parser seems to be better educationally and it tends to be better for debugging as well, since you can step through each function and determine what is happening, but the parser generator debugging makes less sense due to obscure methodologies for developing the language. Even though the recursive descent parser is more work, and does require more from the developer to implement, that same developer will have a better sense of how the language works having made the language directly rather than generating it with a parser generator.

Section 7: Software development life cycle model

For this project we used Test Driven Development (TDD). Test Driven Development sounds exactly like what it is, we are given tests, and we need to develop the CatScript Compiler so that we pass the tests provided. I do believe that in this instance test driven development would be the optimal choice, as it asserts that the output of the input must be specific, so it would be harder to mess up a fundamental aspect of the parser. Although test driven development would be good in this instance, to do proper test-driven development so that the tests are complete and independent is a difficult challenge.

Throughout the development of this project, we experienced some tests that didn’t satisfy completeness, leading to the potential of some issues from the parser or evaluator in the compilation part of the project. One example that stands out is in the parser test, we are given a test that tests whether a list can be specified as an ‘list<int>’ list, but not as a non-descript list as such ‘list’. If someone were to miss this in the parser then in the compilation phase, they would end up with a test failure. Although I did not fall for this trap, I did notice the tests didn’t satisfy completeness. For example, I will provide a screenshot of a test from the Catscript Compiler:



```
Carson Gross
@Test
void returnListWithComponentTypeWorksProperly() {
    assertEquals( expected: "[1, 2, 3]\n", compile( src: "function foo() : list<int> { return [1, 2, 3] }" +
            "print(foo())"));
}
```

Now it isn’t to say that the tests were terrible and weren’t helpful. I believe that the way the compiler was structured for us to develop was great and reflected a different way of developing a project. I would prefer Test Driven Development over other software engineering practices; it seems easier to work with than Continuous Integration and other developmental life cycles.