Section 1 Program:

https://github.com/AJ-Zetzer/csci-468-spring2025-private/blob/main/capstone/portfolio/source.zip

Section 2 TeamWork:

Team-Members: Cody Hagar.

For the compilers project, Cody and I wrote each other's technical documentation and tested each other’s code by writing three test cases for our respective compilers. I spent roughly two hours on the documentation and test cases, and Cody did the same for my project. Luckily, the tests written for my compiler all passed, so no changes had to be made.

While writing my tests for Cody’s compiler, I discovered that the assignment statement was broken in both my project and Cody’s. Even though the tests he wrote for mine passed, the process of writing tests for someone else helped me uncover errors in my own code.

Writing each other’s documentation and creating test cases was a valuable way to identify gaps in our compilers that the original tests missed. The documentation also allowed us to contribute to each other’s projects without taking away from the educational value.

Section 3 Design Pattern “memoization”:

When creating the Catscript compiler, I used the memoization design pattern during type inference and semantic analysis of lists, where the compiler frequently needs to construct or compare list types. Memoization is a design pattern in which the output of a computationally expensive function or method is cached. This allows the result to be reused without redoing the computation, saving time by reducing redundancy and improving the compiler's efficiency. While saving a list type is not particularly computationally expensive, this design pattern is scalable and can be applied to other features of the compiler to improve execution time. A longer Catscript program could potentially become memory-intensive and reduce performance if types are redundantly constructed. Finally, memoization simplifies type equality checks, since the same instances can be reused from the cache.

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

Section 4 technical Writing:

Catscript Guide

Documentation for the Catscript Programming language.

Introduction

Catscript is a simple scripting language that is compiled into JVM bytecode. Here is an example:

```
var x = "hello, world!"
print(x)
```

Output:

```
hello, world!
```

Catscript is heavily inspired by Javascript and Java. Catscript's full grammar can be found here.

Types and Variables

In Catscript, you can define a variable with the `var` keyword. You can also optionally explicitly set the data type of the variable. If you don't specify the type of the variable, it is automatically inferred.

```
var x: int = 1
var y: string = "hello, world!"
var z = 5 //without specifying the type
```

After defining variables, you can assign them different values.

```
var x = 1
x = 2
print(x) //output: 2
```

Catscript is a statically typed programming language, meaning the types of variables are known at compile time. Here are the types supported by Catscript, with examples:

`int`: a 32-bit integer

```
var x: int = 10
```

`bool`: a boolean value, either `true` or `false`

```
var x: bool = false
```

`string`: a java-style string

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

`null`: the absence of a value

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


`list`: a list of values. The components in the list can be specified by using `<>` similar to how generics work in Java.

```
var x: list = ["a", "b", "c"]
var x: list<int> = [1, 2, 3]
var x: list<list<bool>> = [[true, false], [false, true]]
```

`object`: could be any type

```
var x: object = 1
var x: object = "hello, world!"
var x: object = [1, 2, 3]
```

Expressions

Expressions are elements of a Catscript program that, when evaluated, produce a value. Catscript supports a number of ways to represent expressions.

Equality Expressions

Catscript uses standard Java-like syntax for most of its expressions, including equality expressions. To check if two values are equal, you can use ==. You can use `!=` to check if two values are not equal.

```
1 == 1 //true
1 == 2 //false
1 != 2 //true
```

Comparison Expressions

Catscript also uses standard syntax for comparing expressions. There are four different ways to compare values:

```
1 > 0 //Greater than
1 >= 1 //Greater than or equal to
0 < 1 //Less than
1 <= 1 //Less than or equal to
```

Like equality expressions, comparison expressions return a boolean value.

Additive Expressions

You can perform all four basic mathematical operations in Catscript. In order to add or subtract two values, simply use `+` or `-` respectively.

```
1 + 2 //3
2 - 1 //1
```

Factor Expressions

You can multiply or divide two values using `*` or `/`, respectively.

```
2 * 2 //4
2 / 2 //1
```

Unary Expressions

There are two types of unary expressions in Catscript: negating boolean values (switching from `true` to `false` and vice versa) and negating integer values (essentially multiplying by `-1`). You can negate a boolean value by using the `not` keyword. You can negate an integer value by using the `-` symbol.

```
not true //false
not not true //true
-5 //-5
```

Features

For loops

For loops are the main tool used for iteration in Catscript. They consist of a few things:

- the `for` keyword
- an identifier to represent the element on the current iteration
- an expression to iterate over
- a body of statements to execute over each iteration

A for loop in Catscript looks like this:

```
for(item in ["x", "y", "z"]){
 print(item)
}
```

Output: `xyz`

You can also use variables as the expression to iterate over in a for loop.

```
list = [1, 2, 3]
for(item in list){
 print(item)
}
```

Output: `123`

If Statements

If statements are used to control the flow of execution in a Catscript program. If statements consist of a few things:

the `if` keyword
an expression
A body of statements. If the expression evaluates to `true`, then the body of the if statement will execute.

If statements in Catscript are very similar to if statements in Java. Here is an example:

```
if(x == 1){
 //do something
}
```

Catscript also supports `else` and `else if` statements.

```
if(x == 1){
 //do something
} else if (x == 2){
 //do something else
} else{
 //do another thing
}
```

Print Statements

Catscript provides a way for users to print data to stdout using the `print` statement. This is a straightforward statement. It takes in an expression as an argument and prints out its value.

```
print("hello, world!") //output: hello, world!
print(1 + 1) //output: 2
print("1 + 1")//output: 1 + 1
var x = 3
print(x) //output: 3
```

Function Definitions

Catscript provides a way to break programs into smaller chunks using functions. In order to use a function, you must first define the function. A function definition consists of the following items:

the `function` keyword
the name of the function
a list of parameters
the return type, if applicable
a body of statements

if applicable, a return statement at the end of the body that returns a value back to the caller (note that the value returned should match the return type of the function)

For example, a basic function called `myfunction` with a void return type would look like this:

```
function myfunction(){

}
```

Now let's say that we want `myfunction` to take a string and integer as parameters.

```
function myfunction(mystring, myinteger){
 print(mystring)
 print(myinteger)
}
```

We can optionally specify the type of each parameter.

```
function myfunction(mystring: string, myinteger: int){
 print(mystring)
 print(myinteger)
}
```

If we wanted to return an integer from our function, we can use the `return` keyword. We also have to set the return type of the function to return an integer.

```
function myfunction(mystring: string, myinteger: int): int{ //must
return an int  print(mystring)
```

```
 print(myinteger)
 return 5
}
```

It is important to know that the return type must be accurate for each possible branch of execution in a function. Simply put, the compiler checks to make sure that a function will **always** return what is specified by the return type. For example,

```
function myfunction(myinteger: int): int{ //must return an int
 if(myinteger > 0){
 return 1
 }
}
```


Is **NOT** a valid Catscript program, because the function will not return anything if `myinteger <= 0`. A fixed version of this program could look like this:

```
function myfunction(myinteger: int): int{ //must return an int
 if(myinteger > 0){
 return 1
 }else{
 return 2
 }
}
```

Function Calls

Just defining a function won't actually do anything. In order to execute a function, you need to call it. A function call consists of:

- the name of the function
- a list of arguments

For example, if we had a function that looked like this

```
function myfunction(myinteger: int): int{ //must return an int
 if(myinteger > 0){
 return 1
 }else{
 return 2
 }
}
```

We could call it like this

```
var x = myfunction(5)
print(x) //output: 1
```

It is important to note that the arguments you give to a function call must match the parameters in a function definition statement.

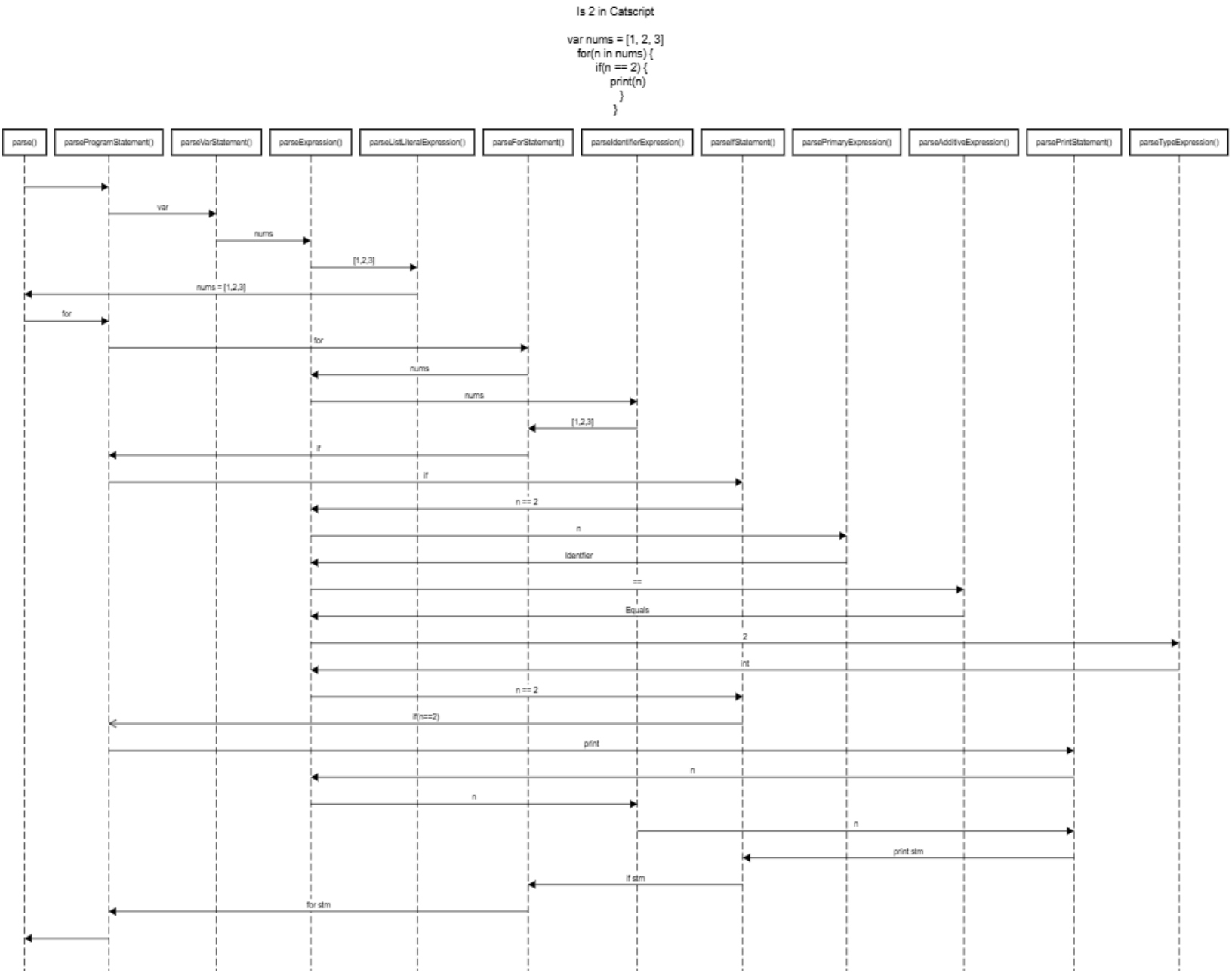
```
var x = myfunction(5, 6) //NOT valid, too many arguments
```

Comments

To add a comment in Catscript, you can use the `//` symbol, followed by your comment.


```
//this is a comment!
//use these to document your Catscript program.
```

Section 5 UML:



Section 6 Design TradeOffs:

The CatScript compiler uses recursive descent parsing rather than relying on an automated parser generator. This choice directly impacts how the language is implemented and understood.

One major advantage of recursive descent parsing is its simplicity. The parser closely mirrors the grammar rules, making it easy to follow and understand. Each type of statement and expression is handled in its own function, forming a hierarchical structure that reflects the syntax tree. This approach makes the parsing logic intuitive.

However, this clarity comes at the cost of repetition. Since each grammar rule is handled manually, similar logic is often repeated in multiple places, which can lead to more redundant code compared to a generated parser.

On the positive side, recursive descent provides more control over parsing behavior. Custom error handling and specific edge-case logic can be implemented directly where they are needed. This control makes incremental development easier, as new language features can be added by simply updating or adding specific functions without requiring changes to the rest of the parser.

That said, this approach is more labor intensive than using a parser generator. Automated tools can handle grammar changes more efficiently and often include built-in error recovery mechanisms, which must be implemented manually in a recursive descent parser.

Despite the added effort, recursive descent was the better choice for CatScript. It aligns with the project's goals of extensibility and educational value. Its structure also supports a test-driven development approach effectively, allowing for continuous growth and refinement of the language.

Section 7 Software Development LifeCycle:

For the development of CatScript, a Test-Driven Development (TDD) approach was used. In TDD, the tests are written first, and then the functionality is implemented. This ensures that the functionality aligns with the original scope of the project by clearly defining the expectations before implementation.

Using TDD helped shape the design and direction of CatScript's development by first defining the expected outcomes. Each parse rule, statement, and expression was built around these tests, providing direction during development.

The feedback loop of implementing a feature and then immediately testing it feels efficient and satisfying. Keeping track of progress becomes easy as the tests can be tackled one at a time. This turns a task that may seem daunting at first into something manageable by breaking it down into many small components.

Like the recursive descent parser, TDD also promotes incremental development and makes adding new features to the language easy. Features can be implemented one at a time in a manageable and compartmentalized way, keeping the project organized as it expands.

Overall, TDD aligned well with the goals of CatScript. It helped maintain code quality and ensured that development remained on track through the different stages of the project.