Section 1: Program

There is a zip file with the source code for this project titled source.zip.

Section 2: Teamwork

My partner provided me with three unit-tests that tested areas of the program that the provided testing of the program did not cover. These unit tests ensured that the CatScript parser works correctly when parsing multiple expressions and expressions that are combined.

My partner also wrote the technical documentation for the CatScript programming language, located in section 4 of this document. This documentation allows developer that would be looking at the language for the first time to have a technical document to guide them through the features and syntax of the language. Technical documentation is an extremely important part of any programming language and my partner did a great job implementing this for my project.

For my portion of the project I also gave my partner unit tests and documentation. My partner and I both wrote the entirety of our compiler project ourselves with the only shared code being the baseline that was given for the projects and the shared unit tests. For a time estimation, I probably spent around 90% of my time on the complier, and 10% of my time on the documentation and unit tests. I estimate this was similar for my partner.

Section 3: Design pattern

Memoization is a design pattern used to cache expensive function calculations into a HashMap so that the value can be retrieved without having to perform the calculations every time we need the value provided from the function call. Caching allows us to store and access the values that we calculate within the function and use them again. This is especially useful for functions that are often running computations on the same input. This significantly increases computation times of functions of these functions.

In the CatScript compiler, this is implemented by taking a CatScript type in as a parameter in the function that returns us a List Type. The CatScript type is stored as the key and the List Type is stored as the value in the map. The function checks to see if the cache contains the CatScript type key and returns the already calculated List Type if it does. If the cache does not contain this key, it calculates the List Type, then puts the CatScript type and List Type into the cache as a key value pair, then returns the newly calculated List Type. This speeds up the process of converting CatScript Types into List Types.

Here is the implementation of Memoization in CatScript below:

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

Section 4: Technical writing. Include the technical document that accompanied your capstone project.

Catscript Guide

This document serves as an informative guide to the catscript language. It satisfies section 4 of the capstone document

Introduction

Catscript is a simple scripting langauge that features both expression and statements. Below you will find a guide on how to use every feature in the language.

Expressions

Equality Expression

Equality expressions allow for equality between a left and right hand side to be checked. The right and let hand side must evaluate to either a string, integer, or boolean value. Additionally, the bang equal, represented with a != symbol will check if things are not equal.

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

The first expression will evaulate to false while the second will evaluate to true.

Comparison Expression

Comparison expressions allow for a comparison between integers to evaluate to either a true or false value. Four comparison are used in the language, the less than, less than or equal, greater, and greater than or equal.

```
1 > 2
1 >= 2
1 < 2
1 <= 2
1 <= 1
```

The above code will evaluate to false, false, true, true, true. It is important to note that only integers are supported and any other type will throw an error.

Additive Expression

The additive expression supports both the plus (+) and minus (-) symbol for integers. However, additive expression can be used for any other type + a string which will evaluate to a string representation. Using a minus symbol for anything other than two integers will throw an error.

```
1 + 1
1 - 1
1 + a
1 + null
```

The evaluation of the above code will be 2, 0, 1a, 1null

Factor Expression

The factor expression is used to either mulitply (*) or divide (/) two integers and can only be used on integer types without errors. It is important to note that a variable can be used to multiple or divide even if it was not declared as an int as long as it is in reality an integer because catscript will infer the type.

```
1 * 3
8 / 4
```

The code above will evaluate to 3, 2.

Unary Expression

The unary expresion is used to negate an integer or a boolean. To negate boolean values we can use the (not) keyword. To negate inegers we use (-) sybmol

```
false = not true
-1
```

The first evaluates to true and the second represents the negative integer negative 1.

List Literal Expression

The list expression alwasy begins with a opening bracket and ends with a closing bracket. Lists can either have primitive types as values or they can be of type object which will allow for the mixing of types.

```
[1,2,3]
[[1],1,2]
["hi","there"]
```

All the provided options are examples of valid lists.

Function Call Expression

The function call expression always begins with a valid identifier and is then followed by opening and closed parenthesis. Within these parentheses is an argument list that consists of valid expressions.

```
foo()
foo(1,"2")
foo(1 + 2)
foo([1,2,3])
```

All the above options are valid function call expressions

Statements

For loops

The for statement must begin with the word for, then an opening parenthesis followed by an identifier, the word in, and then an expression. Finally, the opening and closed brace are then required.

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

The above code demonstrates looping through a list expression and printing the values 1, 2, 3.

If statement

The if statement requires the keyword if, then opening and closing parenthesis that contain an expression that can be evaluated to either true or false. In catscript, after the if and the expression no matter if there is a body or not the opening and closing brace are required. Additionally, an else may by added that is either followed by more if statements or a list of statements.

```
if(true){ print(1) }
if(true){ print(1) } else { print(2) }
```

Above are examples of the if statement with an else and without.

Print Statement

A print statement is identified by the keyword print, and is followed by opening and closing parenthesis with an expression inside. The expression is enforced so printing nothing is not allowed.

```
print(1 * 5)
print("hello world")
```

The above code will work for any type of expression in catscript as long as it can be evaluated without error.

Variable Statement

A variable statement is identified by the var keyword. It is then optionally followed by a : and a type declaration. Next an equal sign is used to set the variable to an expression.

```
var x = "hello"
var y : int = 15
```

The above code shows how to declare a variable in catscript with or without a type.

Assignment Statement

The assignment statement is used to reassign a catscript variable that is within the current scope and shares the name of the identifier the programmer is trying to reassign. If a variable is declared as an object type anything else can be assigned to it. Additionally, nothing can be assigned to a null value.

```
var x = "world"
x = "hello"
```

The above code shows you how to assign a previously declared catscript variable.

Function Declaration

A function declaration is signified by the function keyword. It is then followed by an identifier, then parenthesis. Within the parenthesis there is a list of parameters. These parameters are a series of identifiers that are seperated by commas. Types can be declared or not declared

```
function foo(x) { print(x) }
function foo(x : int, y : string){print(x) print(y)}
```

The above code shows two valid was to declare functions in catscript. one with types and one without types for the paremeters

Return Statement

A return statement is signified in catscript by the return keyword and is then followed by empty space or an expression

```
function foo(x) : int { return x + 1 }
function foo(x : int, y : string){print(x) return}
```

The above code shows the returning of an expression and the returning of nothing.

Types

int

A 32 bit integer

string

This is a value that is enclosed in the " " symbols.

bool

A true or false value

list

This is a list of value that all are assignable to the type x

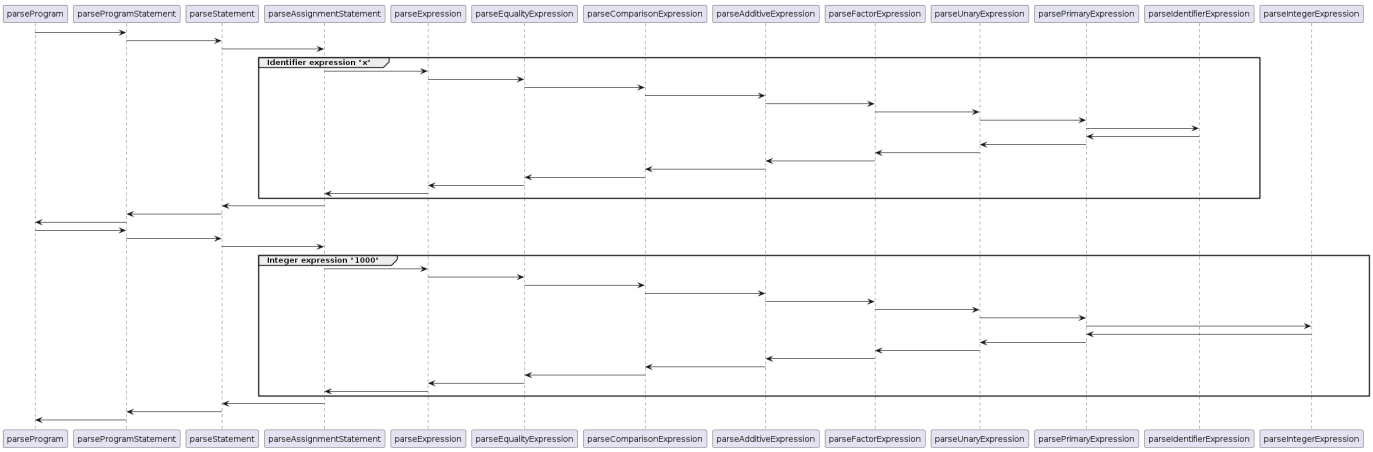
null

The null or nothing type

object

this van be any type of value

Section 5: UML.



This UML Diagram describes the parsing logic for parsing an Assignment Statement in CatScript. The diagram shows how a recursive descent parser begins at the top level of ParseProgram, and makes it's way all that way down to both the ParseIdentifierExpression and ParseIntegerExpression levels of the parser, all while calling each function recursively and making it's way back up to the top level of the parser.

Section 6: Design trade-offs

For this compiler we decided to use a Recursive Descent Parser for our design. Recursive descent is a “top-down” approach to implementing a parser and has a number of benefits when compared to other parsing methods. Most production parsers are recursive descent parsers because they offer so much more benefit over other parsing methods. Here I will talk about the pros and cons of using recursive descent as our design choice.

Pros:

- Recursive descent parsers are extremely easy to read when compared to other parsing methods such as Parser Generators.
- Recursive descent parsers are also much simpler than other parsing languages. Meaning that the parsing logic often closely resembles the actual grammar of the language it is parsing. This makes the parser extremely simple and easy to follow.
- Recursive descent also makes error reporting easy by generating easy to read errors and more detailed error messages.

Cons:

- Parsers like Parser Generators require less written code than recursive descent with is written mostly “by hand”.
- Recursive descent parsers require a lot of infrastructure to implement and maintain. This could be a negative trade off for developers when comparing with other parsing methods.

Section 7: Software development life cycle model

We used Test Driven Development (TDD) as the model to develop our capstone project. TDD is a development approach where the tests for the project are written before the implementation of the working source code. TDD was extremely beneficial in this project because it allowed us to see what the expected outcome of the portions of the compiler we were working on was and give an end goal for the implementation of each feature. TDD also allowed us to become very comfortable working with the debugger and stepping through the project one line at a time, allowing us to get a great understanding of the codebase and implementation of each aspect of the project.

I really enjoyed TDD as the model for this project, and it will be extremely useful in future work. Often in industry, TDD is an ideal but unrealistic model of development unless a developer finds themselves building something from scratch. However, the implementation of TDD in this project will certainly still have large benefits with writing, fixing, improving, and understanding unit tests in other code bases. Unit tests and testing coverage is a crucial part of commercial software, and the experience with testing in this project will prove extremely beneficial.