

Section 1: Program

To see view the source code navigate to the `source.zip` file in the same folder as this document.

Section 2: Teamwork

The Catscript compiler, interpreter, and parser were written by myself building atop the codebase provided by Mr. Carson Gross. Team member 2 was tasked with writing 3 tests for my code to ensure it properly implemented Catscript as well as writing documentation for the language provided in this document. In terms of time spent, I probably did 95% of the total hours ours spent on the project while team member 2 with the smaller workload did the remaining 5%. In exchange, I did approximately 5% of their project by writing documentation and 3 tests for it.

Section 3: Design pattern

We opted to use the memoization design pattern to optimize the creation and storing of list component types. The list type in Catscript contains another type in it known as the component type (the type of the items in the list). The component type can even be another list type allowing for the creation of an unlimited number of types such as `list<string>`, `list<list<string>>`, and even `list<list<list<list<list<list<string>>>>>>`. Memoization is a pattern to cache identical data, in this case list types, so we don't need to create new instances all the time. This can be very helpful when instantiating a class is performance intensive but even in our case where this isn't the case it can save a little on memory by reusing the same object to represent the same type.

Provided below is the implementation of the Memoization pattern highlighted in yellow. We used a hash map as our cache to keep track of created list types and in the `getListType` method we try to first return an existing list type from the cache before creating a new one.

`src/main/java/edu/montana/csci/csci468/parser/CatscriptType.java:`

```
private static final HashMap<CatscriptType, CatscriptType> LIST_OF = new HashMap<>(); // the cache

// this method returns a list type that wraps the passed componentType
public static CatscriptType getListType(CatscriptType componentType) {

    // try to reuse an existing instance of the list type
    if (LIST_OF.containsKey(componentType)) {
        return LIST_OF.get(componentType);
    }

    // otherwise create a new instance of it and save it
    CatscriptType newListType = new ListType(componentType);
    LIST_OF.put(componentType, newListType);

    return newListType;
}

// This is needed for memoization with a hashmap so equality of different objects isn't
// based of their memory address but instead the type that they represent.
@Override
public int hashCode() { return Objects.hash(toString()); }
```

Debugging this code, we can see how the hash map fills up with component types as keys and the list type wrapping that component type as the values. By reusing the same list type object it is able to be more efficient.

```

∞ LIST_OF = {HashMap@1994} size = 4
> {CatscriptType$ListType@2000} "list<string>" -> {CatscriptType$ListType@2009} "list<list<string>>"
> {CatscriptType$ListType@2009} "list<list<string>>" -> {CatscriptType$ListType@2011} "list<list<list<string>>>"
> {CatscriptType@1993} "string" -> {CatscriptType$ListType@2000} "list<string>"
> {CatscriptType$ListType@2011} "list<list<list<string>>>" -> {CatscriptType$ListType@2015} "list<list<list<list<string>>>>"

```

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

The following documentation written by team member 2 gives a brief example heavy overview of how to write Catscript.

Catscript Documentation

Introduction

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

```

var x = "foo"
print(x)

```

This language has basic functionality and basic typing. It is a functional programming language, with the capability of adding new functions but no capability to create new objects. It employs for loops, if statements, variable statements, and variable reassignments. A print function is also included for ease of testing. It has six simple types: int, string, bool, list, null, and object.

Features

For Statements

The for statement is used to run the same code repeatedly. It takes a list and makes a new variable that iterates over the said list, updating to the next value in the list every time the for statement loops. The for statement stops once the list has been completely iterated over. Here is an example:

```

for (x in [0, 1, 2]) {
    print(x + 1)
};

```

In the example, a new variable x is created and has the first value of the list assigned to it. For each loop, x + 1 is printed and if there are additional values in the list x is assigned to the next value and another loop occurs

For loops cannot be broken out of early, except by using the return expression if the loop is in a function.

Here's the formatting for a Catscript for statement:

```

'for', '(', IDENTIFIER, 'in', expression ')', '{',
{statement}, '}'

```

If and Else Statements

The if statement is used to run code when a specific condition is met. It takes an expression that evaluates to a Boolean value, known as a bool type in Catscript and runs code if that Boolean value is true. If the value is false, an optional else statement can be added to run a different code. An else if statement can also be added that functions as an else statement but with another condition. Here is an example:

```

if (x > 1) {
    print(x)
}
else if (x == 1) {
    print(1)
}
else {
    print(0)
}

```

In the example, we give the if statement a variable called x (implied to be an integer) and check if it is bigger than 1. If it is, then we print it. If not, then we check if x equals 1. If it does, then we print 1. If not, then we move on to the final else which will print 0.

As shown below, if and else statements require brackets, making it impossible for an ambiguous else statement that other languages might have.

Here's the formatting for a Catscript if statement:

```

'if', '(', expression, ')', '{',
    { statement },
'}' [ 'else', ( if_statement | '{', { statement }, '}' ) ];

```

Print Statements

Print statements are simple. It just takes an expression and outputs it to the system output. The print statement is added as a syntactic element instead of a function call for the purpose of being able to use it before function calls are fully developed.

Here's an example:

```

var x = "hello world"
print(x)

```

This print statement will output the string "hello world", which was assigned to x.

Here's the formatting for a Catscript print statement:

```

'print', '(', expression, ')'

```

Variable Statements & Assignments

Variable statements are used to create a new variable under a specified name if that name is not already in use. The type can be explicitly stated or left out and assumed from the right side due to type inference. Here's an example:

```

var x : int = 15

```

In this example, if “: int” was not included, the new variable “x” would still be of type int, due to inferring the type from 15. Also, if “var” was not included, this statement would be assumed to be a variable reassignment, which would fail due to “: int” whether or not “x” was previously defined or not.

Variable assignments are virtually the same, except without explicit typing or the “var” word in front, since the variable is already initialized. These assignments reassign the value of a given variable to the new value stated. Here's an example using the variable from the previous example:

```

x = 20

```

A few notes concerning variable assignment: variables cannot be reassigned to a new typing, null can be assigned to a variable of any type, and any type can be assigned to a variable with the object type.

Here's the formatting for a Catscript variable statement:

```
'var', IDENTIFIER, [':', type, ] '=', expression;
```

And here's the formatting for a Catscript variable assignment:

```
IDENTIFIER, '=', expression;
```

Functions and Returns

Functions in Catscript are somewhat simpler than in other languages. A function declaration starts with the word 'function'. After, an identifier for the function's name is required. This is followed by a list of the parameters. Each parameter requires a name but does not explicitly need a type, otherwise defaulting to an object type. After the parameter list, a return type can be stated, otherwise, the return type is assumed to be void. Finally, we have the function body contained in two braces, which contain the code to be run by the function. Here's an example of what a function declaration looks like:

```
function addFive (num : int) : int {  
    return num + 5  
}
```

The return expression functions the same as a return from Java, being that a return statement on its own will end a function and a return function with a value returns that value.

Function calls work the same as Java. To call a function, you state the name of the function along with any required parameter in parentheses. Here's an example of a function call using the addFive function:

```
var x : int = addFive(5)
```

Here's the formatting for a function declaration:

```
'function', IDENTIFIER, '(', parameters, ')', [ ':' + type ], '{', {statement}, '}'
```

And here's the formatting for a function call:

```
IDENTIFIER, '(', arguments , ')'
```

Types

Object

A type that can be assigned a value of any type. Any of the following examples are examples of an object (except null which is a value).

Int

A simple 32-bit integer type supporting addition, subtraction, multiplication, and division. Below are some examples that evaluate to an Int:

- 32/2
- 50*5-50

String

A Java-style string that supports concatenation. Here are some examples that evaluate to a String:

- `"This is a string"`
- `"Hello " + "World"`
- `"Number " + 1`

Bool

A boolean value. To flip a Bool, Catscript uses the keyword "not". Here are some examples that evaluate to a Bool:

- `true`
- `not (5 < 6)`

List

A list of values with type "x". Lists are unique since when stating a list as a type, they take the format `list<x>`, where "x" is any type. Lists are covariant, meaning narrower component types can be cast to a wider type. For example, a list of objects can be assigned a list of strings. This does not cause issues because lists in Catscript are immutable. Here are a few examples that evaluate to a list:

- `[1, 2, 3]`
- `[[1,2],[2,3],[3,4]]`

Since lists are unique, here are some snippets of code that involve lists too:

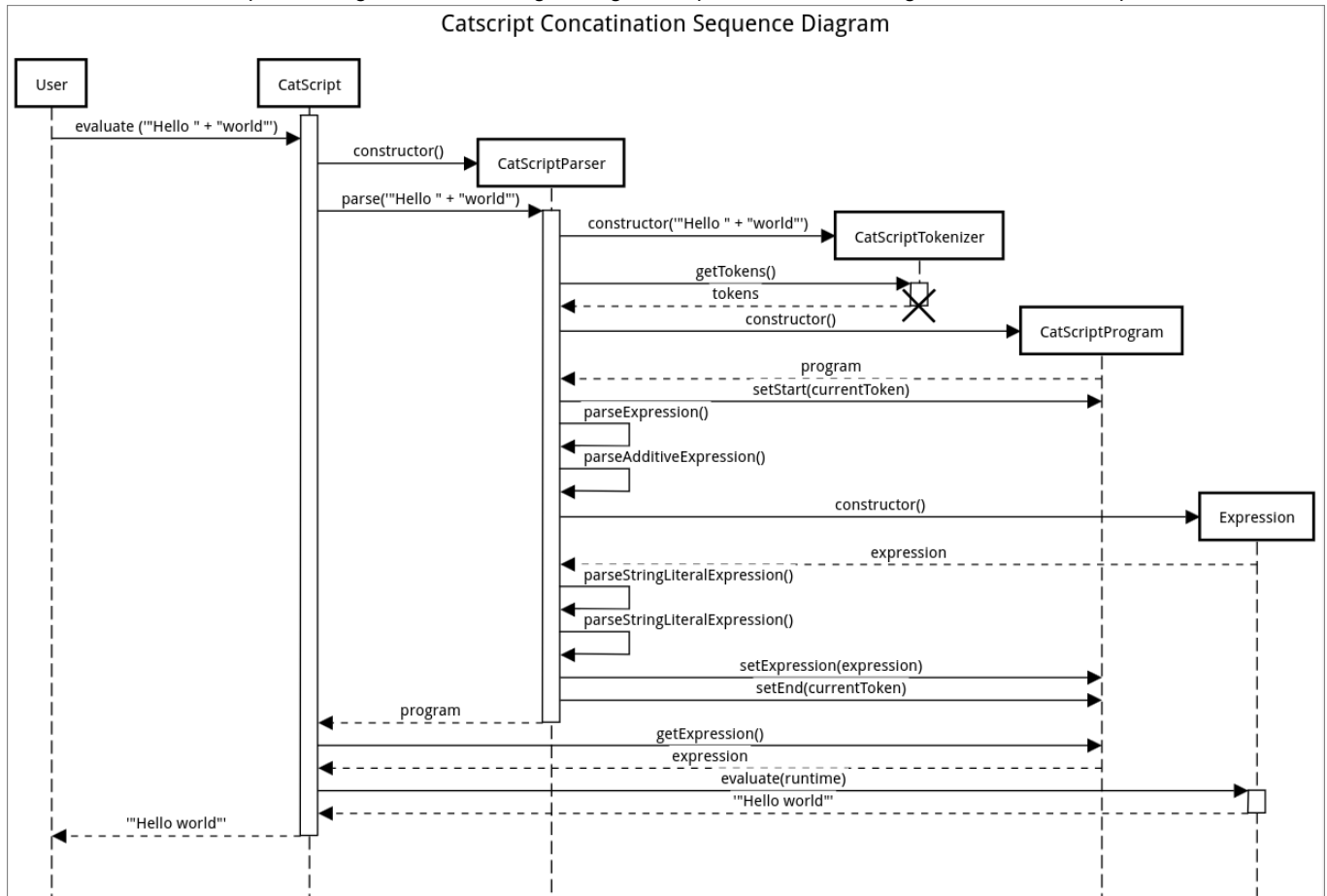
- `var i_list : list<int> = [1, 2, 3]`
- `var o_list : list<object> = i_list`
- `var o_list : list<object> = [0, "Hello", true]`
- `var i_list_list : list<list<int>> = [[1,2],[2,3],[3,4]]`

Null

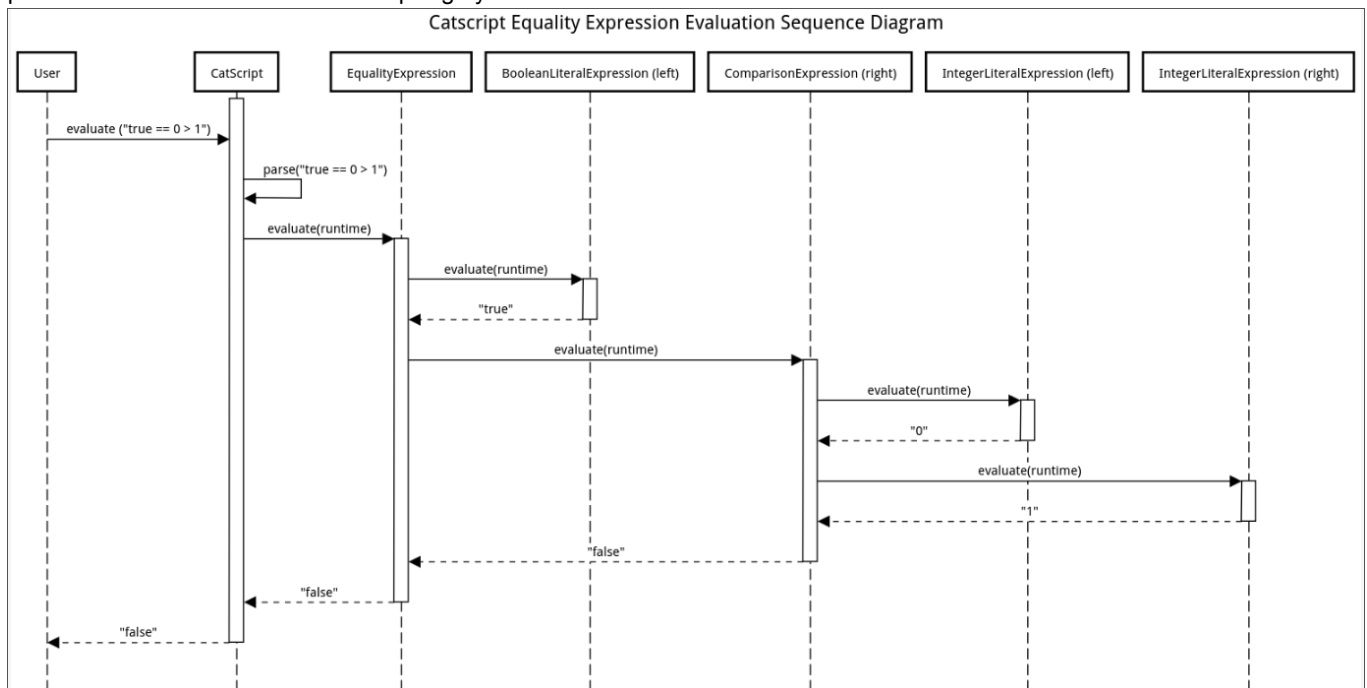
Although null is not a type in Catscript, it is relevant in that an object of any type can be assigned a value of null, representing an absence of a value.

Section 5: UML.

Provided below is a sequence diagram summarizing the high level process of evaluating a concatenation expression.



Provided below is a sequence diagram showing the nested execution of our evaluation method. This same nested calling pattern is even used even when compiling bytecode.



Section 6: Design trade-offs

The major design trade off in this project was whether or not to use a visitor pattern on the intermediate representation. For context, the intermediate representation in this project includes classes like `StringLiteralExpression` and `PrintStatement`.

After tokenizing the source code and parsing, instances of these classes are what hold all of the information that was just ingested. This data is then used to evaluate or compile the program to bytecode. However, down the line, one might want to also add a transpile operation to another programming language or another compile operation to native machine code. If all of these methods are added directly to the intermediate representation classes then it betrays the notion of separation of concerns. Whenever you add a new operation you would need to modify every single intermediate representation class. Ideally, then, the intermediate representation would be a more generic construct and these operations on these constructs would be stored elsewhere. This can be accomplished using a visitor pattern which lets you bundle all the methods for one operation like compiling to bytecode under one visitor class with `visit(intermediateRepresentationObject)` methods with method signatures corresponding to each intermediate representation class. The downside of this system is that it is more complex and if you create a new intermediate representations class you have to edit every single visitor operation class to have a visit method for it.

The visitor pattern is probably worth it for organization at scale if support for a high n number of operations on the intermediate representation classes is required. For our purposes however, our number of operations is only two (evaluation and compiling to bytecode). Meaning, the visitor pattern would have added a lot of extra complexity without providing any real benefit. Thus, we chose to have our operations directly as methods on each intermediate representation class.

Section 7: Software development life cycle model

We used Test Driven Development (TDD) for this project. Throughout the semester we would begin work on the next checkpoint we needed to finish. Each checkpoint would have associated tests to prove that we had implemented that aspect of Catscript adequately. Different tests would rely on completing previous tests so we would start with the simplest tests usually being expressions and then work our way up from there through statements and then functions. This way of starting with the simplest tests and then working our way up proved very effective because it allowed us to focus on creating one atomic feature at a time breaking up what would have otherwise been a daunting amount of work into doable parts. Additionally, the convenience of being able to run past tests proved useful for catching any regressions as we completed later tests.

One possible disadvantage of TDD is it can be challenging to have truly exhaustive tests for every edge case. Meaning, you may have passed all your tests but there may still be obvious bugs that show up when you try anything other than the tests. For example, I tried to write a small program in Catscript to try out the interpreter and in doing I used an assignment statement which as it turns out had been left out of the provided tests for the interpreter. Luckily, it was an easy fix but had the product been shipped to a customer in that state it would have been very embarrassing to miss something so commonly used.

As for collaboration, it worked very well with TDD for the following reasons. Firstly, we could write any missing tests we found in our own personal experience implementing the project such as my missing assignment statement interpreter test and exchange with our team member making both of our Catscript implementations better. Secondly, When collaborating by writing code together typically you need a source control system like Git and you have to deal with the process of merging code together as you both produce code that relies on each another. If instead your just collaborating with tests then the tests can typically be sent once by team member 2 and then you can work at your own pace to tweak the code until those tests pass. In other words, collaborative TDD is very easy to perform and provides a lot of benefit by helping catch edge cases.