Montana State University

Catscript Capstone Portfolio Report

Carson Diehl

Wyatt Vopel

Compilers - CSCI 468

Spring 2024

Professor - Carson Gross

Section 1: Program

The Framework of the Catscript Programming language was provided by Carson Gross. It was completely written in Java, and was edited by me in the Eclipse IDE.

Section 2: Teamwork

The Majority of this project was completed independently with little work being shared. Each team member created a unique version of the catscript compiler as well as supplied the other member with UML and Testing.

Team Member 1

As stated above, Team Member 1 made the code for the tokenization, parsing, and evaluation portions of the Catscript Compiler ensuring the proper function of the program. This is where a majority of the work time lies. Each checkpoint took somewhere between 5-14 hours to complete. They also supplied Team Member 2 with the proper testing to ensure proper production in their compiler.

Team Member 2

Team Member 2 did all of the same things as Team Member 1 and more. Not only did they complete the tokenization, parsing, and eval but also completed a significant portion of the bytecode potion as well. They also gave Team Member 1 several test cases to make sure their compiler works as assumed as well as the documentation for the technical writing and capstone documentation.

Time Breakdown

Team Member 1: 52 hours 91%

Team Member 2: 5 hours 9%

Section 3: Design pattern

Memoization

Memoization is a technique used to speed up computational tasks by storing the results of expensive function calls and returning the cached result when the same inputs occur again. This technique is particularly useful in dynamic programming, recursive algorithms, and functional programming.

The memoization design pattern involves creating a cache to store the results of function calls. When a function is called with certain inputs, the result is computed and stored in the cache. Subsequent calls with the same inputs retrieve the result from the cache rather than recomputing it, thus improving performance.

Catscript Memoization

Memoization in the Catscript Compiler occurs in the CatscriptType.java class. The directory is “src/main/java/edu/montana/csci/csci468/parser" starting at line 37.

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

By caching the results of previous calls, the getListType method avoids recomputing list types for input types that have been seen before, thereby improving the overall efficiency of the program.

Section 4: Technical writing.

Catscript Introduction

Catscript offers a range of functionalities. It includes features like type inference and list literals for any Catscript type. Unlike languages such as Java or C, Catscript allows the use of the "var" keyword for declaring variables without specifying their types. It supports various expressions, including String, 32-bit integer, object, null, and boolean. It can also do operations such as addition, subtraction, multiplication, and division, along with logical operators like 'not'. It also has the functionality of concatenation using the “+” operator. However, it does not support the modulus operator.

Catscript is statically typed, meaning that once a variable's type is declared, it remains consistent throughout the program's execution. Lexical scoping is utilized, which means that variables defined within a certain scope, such as a for loop, will no longer exist once that scope ends unless defined as a field.

The language encompasses conditional statements like 'if', iterative statements like 'for' loops, and supports recursion. Functions can be defined and invoked, with functions lacking an explicitly defined return type defaulting to void.

Catscript's range of features provides ample opportunity for exploration, which will be further illustrated in subsequent sections with accompanying sample code to enhance understanding.

Catscript provides a versatile feature set, including type inference and flexible variable declarations. Despite its simplicity, it offers robust functionality such as various expressions and operations, though it lacks certain operators like the modulus operator. Its emphasis on static typing and lexical scoping ensures code consistency and organization, while support for conditional statements, iterative loops, recursion, and function definition enhances its versatility for programming tasks. Overall, Catscript strikes a balance between simplicity and functionality, making it suitable for various programming needs.

Syntax error handling:

Catscript manages errors by employing line numbers and offsets to provide users with information regarding the location and nature of the error. For instance, an error might occur when attempting to assign a string value to an object variable, in a format resembling the following:

```
Parse Errors Occurred:

Line x: booleanVariable = "'string'"
                        ^

Error: Incompatible types
```

Commenting:

Similar to many contemporary programming languages, Catscript enables users to comment out code. This functionality mirrors that of languages like C and Java, where a double slash denotes a comment that is ignored by the compiler. Here's an example of commenting in Catscript:

```
// this is ignored
```

Type References:

Catscript employs static typing, which implies that once a variable is declared with a type, it cannot be assigned any other type. The data types supported in Catscript include integers, strings, booleans, objects, and null values. These types are declared using keywords such as int, string, bool, object, and null. Notably, types like chars, floats, bytes, or any other unlisted types are not supported in the Catscript language. Functions can return a single type, defaulting to void if not specified, while expressions can be of any type.

Statements

Variable Statements:

Variables in Catscript act as placeholders capable of storing changing values throughout the execution of a program. However, while variables can undergo changes during execution, they are constrained to maintain their initially declared type. For instance, a variable declared as type int can be assigned different integer values but cannot be assigned a boolean value. Here's an example of a variable statement in Catscript:

```
var x : int = 871
```

Catscript is able to infer variable types, so declaring a variable type is not needed. An example like this would work:

```
var x = 871
```

Where Catscript infers that the value 871 is an int.

Assignment Statements:

Assignment statements in Catscript are utilized to alter values for a designated variable. Prior to assigning a value to a variable, it must be initially defined. Here's an example of an assignment statement in Catscript:

```
Var1 = 756
```

If Var1 was not defined it will not be able to be assigned a value. An example of variable definition would look like this:

```
var Var1 = 99
Var1 = 2
```

This sequence would lead to "thisVariable" being declared as an integer type with a value of 99, followed by assigning the integer value 2 to "thisVariable" on the subsequent line.

Print Statements:

Print statements in Catscript take any input and display that input on the console. Here's an example of a print statement in Catscript:

```
print("i like cars")
```

output:

```
i like  cars
```

For Statements:

For statements in Catscript are control flow statements designed to iterate through lists. Here's an example of a for loop in Catscript:

```
for(x in [1,2,3,4]){
      print(x)
}
```

output:

```
1 2 3 4
```

If Statements:

If statements are conditional control flow statements that execute code conditionally based on whether a boolean expression evaluates to True or False. Here's an example of an if statement in Catscript:

```
var x = 44
if(x > 2){
    print("x is greater than 2"
} else {
    print("x is less than or equal to 2")
}
```

Where the output is:

```
x is greater than 2
```

Return Statements:

Return statements in Catscript are used to return values from functions. It's crucial to note that return statements must be the last instruction of any function; otherwise, parsing errors will occur. Here's an example of a return statement in Catscript:

```
function func() : bool {
    return false
}
print(func())
```

Which would output:

```
false
```

Functions

Functions are segments of code intended to be reusable, enabling users to perform specific actions without duplicating large code blocks. In Catscript, functions can be defined as follows:

```
Function moreThanZero(num : int){
       if(num > 0){
            print("This number is larger than 0")
       } else {
            print("This number is smaller than 0")
       }
moreThanZero(15)
```

This would output:

```
This number is larger than 0
```

Expressions

Expressions in Catscript yield a value upon evaluation. They can result in strings, booleans, integers, null, and Lists.

Parenthesized Expressions

In Catscript, parenthesized expressions are any expressions enclosed within parentheses. Multiple sets of parentheses can be used. Here's how parenthesized expressions appear in Catscript:

```
(expression)
```

A more explicit parenthesized expression:

```
(433625)
```

or

```
((((433625))))
```

Both of these parenthesized expressions would evaluate to 433625 in Catscript.

Additive Expressions

Additive expressions in Catscript are utilized for both adding and subtracting integer values and performing string concatenation. Here's how additive expressions appear in Catscript:

```
integer1 + integer2 + integer3 + ....
```

In Catscript, additive expressions can be continued infinitely. Here's a more explicit example of additive expressions:

```
9 + 10
```

Which would evaluate to:

```
21
```

just kidding

```
19
```

Subtraction works similarly:

```
21-9
```

evaluates to:

```
12
```

In Catscript, addition and subtraction can be used together, and they evaluate from left to right associatively, like this:

```
1+2-3+4-2-2
```

Which evaluates to:

```
0
```

Parenthesis can also be used with additive expressions like this:

```
4 + 5 - 8 - (2+2-4)
```

Which would evaluate to:

```
1
```

String concatenation in Catscript works like this:

```
someString1 + someString2 + ...
```

Where a less general string concatenation would look like this:

```
"summer"+"is"+"soonish
```

and would output this:

```
summer is soonish
```

Integer Literal Expressions

Integer literal expressions in Catscript represent 32-bit integer values. Here's an example of an Integer Literal Expression:

```
55
```

This Expression would evaluate to:

```
55
```

String Literal Expressions

String literals in Catscript are arrays of characters that can be of any size, and they are enclosed within double quotes to indicate a string. Here's an example of a string literal expression:

```
"vroom goes car"
```

Which would evaluate to:

```
vroom goes car
```

Factor Expressions

Factor expressions are used for multiplying and dividing values. Factor Expressions in catscript look like this:

```
Value1 * Value2 * ...
```

A more explicit example may look like this:

```
5 * 1 * 2
```

Which would evaluate to:

```
10
```

Division would look like this:

```
10 / 2
```

evaluates to:

```
5
```

Using additive, factor, and parenthesis expressions together, Catscript can yield full mathematical equations, like this:

```
4 + (6*2)-12
```

Which would evaluate to:

```
4
```

Unary Expressions

Unary expressions in Catscript can take one of two different operands. The first operand is a minus (-) sign, which converts the value to its opposite, like this:

```
-19
```

evaluates to:

```
-19
```

The second operand in Catscript unary expressions is the "Not" operator. Using the "Not" operator in Catscript would look like this:

Or more explicitly:

```
not false
```

evaluates to:

```
true
```

Boolean Literal Expressions

Boolean literals in Catscript always evaluate to a true or false value. In Catscript, boolean literals look like this:

```
false
```

evaluates to:

```
false
```

Or:

```
true
```

evaluates to:

```
true
```

List Literal Expressions

List literal expressions in Catscript are similar to arrays in other programming languages. They can store any type of value, and multiple types can exist within the same list literal. Here are examples of list literals in Catscript:

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

Which would evaluate to:

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

As previously mentioned, different types can coexist within the same list literal in Catscript, as demonstrated below:

```
[null,2,"cars", true]
```

Which would evaluate to:

```
[null,2,"cars",true]
```

Null Literal Expressions

A null literal in Catscript represents the absence of a value. It is used as follows:

```
null
```

evaluates to:

```
null
```

Equality Expressions

Equality expressions in Catscript compare two values to check if they are equal or not equal to each other, resulting in a boolean literal (true or false). Here's how an equality expression would look in Catscript:

```
expression == expression
```

evaluates to:

```
true
```

or:

```
expression != expression
```

evaluates to:

```
false
```

Comparison Expressions

Comparison expressions in Catscript evaluate a condition and return a boolean literal based on the evaluation. Here's how Where the < sign can be <,>,<=, or >= comparison expressions would look in Catscript:

```
1 > 10000
```

evaluates to:

```
false
```

Recursion

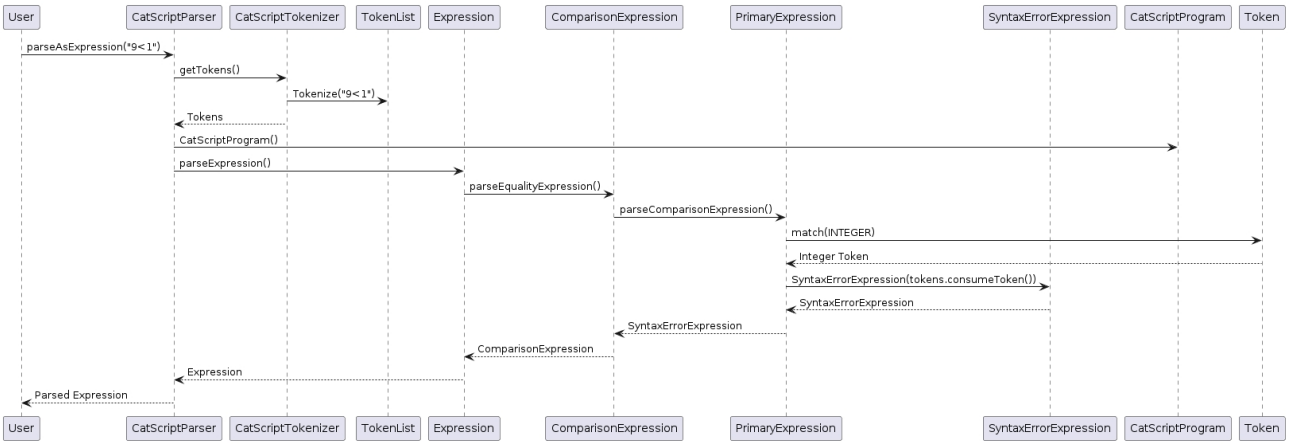
Recursion, a programming convention where a function calls itself as part of its execution until a base case is met, is possible in Catscript. Here's an example of recursion in Catscript:

```
function recursiveFunction(x : int){
        print(x)
        if(x>7){
            recursiveFunction(x - 1)
        }
}
recursiveFunction(10)
```

Which would evaluate to:

```
10 9 8 7
```

Section 5: UML.



Section 6: Design trade-offs

Optimization of Execution Time vs. Space Requirements

In developing the Catscript compiler, a key trade-off decision is between optimizing for execution time or minimizing space requirements during compilation. For instance, choosing aggressive optimization techniques like inlining can speed up runtime performance but may result in larger compiled code and longer compilation times. The decision hinges on factors such as the target platform, performance requirements, and developer experience. A balanced approach might involve implementing moderate optimizations to strike a compromise between runtime efficiency and compilation speed and code size.

Considering Catscript's intended use case, a balanced approach may be preferable. This could involve implementing moderate optimization techniques that provide noticeable performance improvements while keeping compilation times and code size within acceptable limits.

Section 7: Software development life cycle model

Test Driven Development

Test Driven Development (TDD) is a software development approach where developers, or in this case professors, write tests before writing the corresponding code. This is a great way to learn about the program you are intending to write, since to pass each test you have to know exactly how each component works leading to a better understanding of the whole program. You also get instant feedback so you know exactly where your code is faulty and where it works well.

While it is great to learn, you lose a portion of the "creative liberties" as you are constrained to focus on the specific deliverables that are assigned as tests.