

CSCI 468 Capstone Portfolio

Compilers: CSCI 468

Spring 2023

Justin Kerr, Billy Wood

Section 1: Program

<https://github.com/jikerr15/csci-468-spring2023-private/blob/main/capstone/portfolio/source.zip>

Section 2: Teamwork

Our team consisted of two members, referred to as Team Member 1 and Team Member 2, who collaborated effectively to complete this project. Each team member contributed a significant amount of time and effort towards the project. The team members had different and specific jobs to complete and the effort by both led to a successful outcome. The contributions are as follows:

Team Member 1 Contributions (Myself: Justin Kerr)

As the primary engineer, Team Member 1 was responsible for tokenizing, parsing, evaluating, and compiling the CatScript language. They meticulously worked through the phases of coding and testing, ensuring that the parser could accurately interpret and execute a diverse array of CatScript programs. Their focus was on constructing a robust and efficient parser capable of handling various language constructs, data types, and expressions.

Team member 1 developed and implemented three test cases to test against his own Catscript language but also to test against team member 2 Catscript language.

1. Complex Expression Parsing Test (parseComplexExpression()): This test case assessed the parser's ability to accurately parse and interpret complex mathematical expressions involving arithmetic operations and comparisons. The test verified the correct formation of equality, additive, and integer literal expressions within the parsed expression tree.
2. Function Call with Complex Arguments Test (parseFunctionCallWithComplexArguments()): This test case evaluated the parser's ability to parse function calls containing a mix of different argument types, including integer literals, nested function calls, and list literals. The test ensured that the parser correctly identified the function name, the number of arguments, and the argument types.
3. Nested List Literal Expression Parsing Test (parseNestedListLiteralExpression()): This test case verified the parser's ability to accurately parse nested list literals containing integer literals and other nested lists. The test examined the correct formation of list literal expressions, the number of values within each list, and the types of values contained in the nested lists.

Team Member 2 Contributions

Team Member 2 was responsible for generating detailed technical documentation outlining the syntax, structure, and features of the CatScript language. This documentation served as a crucial resource for understanding the language's underlying principles and guided the development of the parser.

Moreover, Team Member 2 developed and implemented three test cases to evaluate the parser's functionality:

1. Fibonacci Sequence Test (fibonacciSequence()): This test case assessed the parser's ability to execute recursive functions by calculating the Fibonacci sequence up to the 10th element.
2. String Concatenation in For Loop Test (stringConcatenationInForLoop()): This test case evaluated the parser's ability to process string concatenation within for loops, ensuring the parser's capacity to handle complex expressions.
3. Sum Square Difference Test (sumSquareDifference()): This test case verified the parser's ability to compute the difference between the sum of squares and the square of sums within a specified range of numbers.

Our team's collaborative efforts in documentation, test case development, and parser implementation resulted in a highly functional and reliable parser for the CatScript language. Team Member 2's contribution in generating technical documentation and developing essential test cases played a critical role in ensuring the parser's accuracy and dependability, while Team Member 1's work on tokenizing, parsing, evaluating, and compiling provided the foundation for the project's success.

Section 3: Design Pattern

In my capstone project, I used the Memoization design pattern to optimize the performance of a certain part of the code. Specifically, I implemented memoization in the CatscriptType class to avoid the expensive creation of new ListType instances whenever the getListType method was called with the same CatscriptType argument.

```
public static CatscriptType getListType(CatscriptType type) {
    ListType listType = memoizedListTypes.get(type);
    if (listType == null) {
        listType = new ListType(type);
        memoizedListTypes.put(type, listType);
    }
    return listType;
}
```

I chose to use the Memoization pattern in this case because creating new ListType instances for every call to getListType was computationally expensive, and it was common for the same CatscriptType to be used multiple times. By implementing the Memoization pattern, I significantly reduced the overhead of creating new ListType instances by storing previously created instances in a HashMap and returning them when the same CatscriptType was requested again.

This design pattern led to improved performance, lower memory usage, and a more efficient use of resources in the capstone project. Using the Memoization pattern allowed the program to focus on executing other parts of the code, thus providing a better overall user experience.

Section 4: Technical Writing

Catscript Guide

Introduction

Catscript is a simple scripting language. Catscript is a compiled language, and is compiled into JVM bytecode. Catscript is not an object-oriented language, but a procedural language. The concepts of classes, methods, and data structures are not present in Catscript, however the goal is to follow a sequence of tasks and to perform each task in order.

Features

Data Types

There are six data types within Catscript. They are:

- `int` - a 32-bit integer
- `string` - a Java-style string
- `bool` - a boolean value
- `list` - a list of value(s) of type 'x'
- `null` - null type
- `object` - a value of any type

There are no floating-point numbers, only signed integers.

Strings in Catscript are similar to strings within Java. They are immutable and placed within a string pool.

The two boolean values are `true` and `false`.

Lists are the only data structure within Catscript. All elements within a list must all be of the same type.

Comments

There are only line comments in Catscript. A line comment can be created by placing two forward slashes directly next to each other "//".

```
// This is a comment in Catscript and will not be seen as executable code.
```

Any code written on the same line as a comment will not be compiled

```
// print("This code will not print.")  
// The for loop below will result in an error  
  
for(x in [1, 2, 3]){  
    print(x)  
//}
```

Line endings

Catscript does not require a semicolon (;) or any other character to terminate a line of code.

Variables

Variables are declared using the `var` keyword. Variable types can be declared explicitly, but are not required.

Variable Declaration with Implicit Type

```
var x = "foo" // the type of 'x' is string  
  
var y = true // the type of 'x' is bool  
  
var x = "bar" // This is an illegal declaration. A variable with name 'x'  
already exists  
  
// Assign existing variable a new value  
  
x = "bar" // is a legal assignment, and is now bool type
```

```
x = 1 // is also a legal assignment. 'x' is no longer bool type and is now
int type
```

Variable Declaration with Explicit Type

```
var x : string = "foo" // variable 'x' must be assigned a value of string
type

var y : int = 1

var yy : int = "bar" // will raise an error for an illegal assignment

var z : object = "" // type object can have any value

var a : list<int> = [1, 2, 3] // list type must be followed with the type
of elements to hold

var b : bool = true

//Variables keep their explicit type declaration during reassignment

var num : int = 1

num = 2 // is a legal assignment

num = "bar" // is an illegal assignment. 'num' was expecting a value of int
type
```

Scope

Catscript uses static scoping, and variables can be global or local.

```
var x = 10 // this is a global variable

if(true){
  var y = 5 // this is a local variable
    // 'y' can only be seen within this if statement

  print(x) // 'x' is visible and accessible within the if statement
```

```
    print(y)
    x = x - y
}

print(x)
print(y) // 'y' is not able to be seen or accessed
          //outside of its declaration in the if statement
          // This will result in an error
```

Mathematics

Catscript supports addition(+), subtraction(-), multiplication(*), and division(/). Parentheses can be used in an expression for operation precedence.

```
2 + 2 // evaluates to 4

2 - 2 // evaluates to 0

2 * 2 // evaluates to 4

2 / 2 // evaluates to 1

2 * (2 + 2) - 2 // evaluates to 6
```

Math operations can be used during declaration or reassignment

```
var val = 1 + 1

val = val * 2
```

Boolean Expressions

There are three boolean expressions in Catscript:

1. Boolean Literals
2. Value Comparison
3. Equality

Boolean Literals

Literals values are `true` and `false`

Value Comparison

There are four comparison operators:

1. Greater than (>)
2. Less than (<)
3. Greater than or equal to (>=)
4. Less than or equal to (<=)

```
2 > 1 //evaluates to true  
  
2 >= 2 //evaluates to true  
  
2 > 2 //evaluates to false  
  
2 > 3 //evaluates to false
```

Equality

Two equal signs (==) are used to check for equality.

```
2 == 2 // evaluates to true  
  
2 == 3 // evaluates to false
```

Not Operators

There are two not or negation operators: the keyword `not` and `!` (bang).

```
not is used with boolean literals  
  
not true // evaluates to false  
  
not false // evaluates to true  
  
not can be used recursively  
  
not not true // evaluates to true
```



```
not not false // evaluates to false
```

! is used to negate equality expressions and to check for inequality

```
2 != 2 //evaluates to false
```

```
2 != 1 // evaluates to true
```

Boolean Variables

Variables can be assigned with boolean expressions. The value of the variable is the result of the expression.

```
var boolresult = 2 > 1 // the value of 'boolresult' is true
```

```
boolresult = 2 != 2 // the value of 'boolresult' is false
```

Strings

Strings are immutable and cannot be indexed or sliced. Strings can be concatenated using the addition(+) operator.

```
var hello = "Hello"
```

```
var world = "World"
```

```
var greeting = hello + " " + world // "Hello World"
```

Strings can be concatenated with any data type to create a new string.

```
1 + "a" // "1a"
```

```
"a" + 1 // "a1"
```

```
null + "a" // "nulla"
```

```
"a" + null // "anull"
```

For Loops

For loops begin with the `for` keyword. The following expression must be contained within parentheses, and the body of statements must be contained within braces.

The expression must contain:

1. Variable name
2. `in` keyword
3. expression indicating when the loop will terminate

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

The for loop above will loop three times and print each element in the list [1, 2, 3].

If Statements

Catscript only supports if and else statements. There is no if-else statement within Catscript. The expression must be a boolean expression. An `if` statement can be by itself or joined by an `else` statement. An else statement must be preceded by an if statement.

```
//equality  
if(1 == 1){  
    print("1 is equal to 1")  
}  
  
if(1 != 2){  
    print("1 is not equal to 2")  
}  
  
//comparison  
if(2 > 1){  
    print("2 is greater than 1")  
}  
  
if(2 <= 1){  
    print("1 is greater than or equal to 2")  
}
```

```
//true or false
if(true){
    print("true")
}
else{
    print("false")
}

if(false){
    print("false")
}
else{
    print("true")
}
```

Creating Functions

Functions are created using the `function` keyword, followed by the name of the function and ending with a left parenthesis and a right parenthesis.

Function declarations must contain a left brace and a right brace, between which statements are written for the function body.

```
function myfunction(){
    print("This is my function!")
}
```

Functions can have arguments or parameters. The **type** of the argument or parameter is **not** required.

```
function myfunction1(arg1, ...){
    print("This is my function with arguments!")

    print("Here is argument number 1:")
    print(arg1)
}
```

The argument or parameter of a function can have an explicitly declared type.

```
function myfunction2(arg1 : int){  
    print("This is my function with explicit argument types!")  
  
    print("arg1 is an integer")  
    print(arg1)  
}
```

Functions can have an explicitly declared return type

```
function myreturnintfunction() : int{  
    return 1  
}
```

Functions with object return type can return any data type

```
function myreturnfunction1() : object{  
    return 1  
}  
  
function myreturnfunction2() : object{  
    return "Hello World!"  
}
```

Functions can be recursive.

```
function myrecursivefunction(arg1 : int){  
    if(x >= 0){  
        myrecursivefunction(x - 1)  
        print(x)  
    }  
}
```

This recursive function will count up from 0 to 10 and print each number to the user's screen.

Calling Functions

Functions that have been declared are called with the name of the function followed by a left parenthesis and a right parenthesis. Any function arguments go in between the parentheses in the proper order, and are comma separated.

```
myfunction()

myfunction1(arg1, ...)

var x : int = 10
myfunction2(x)

var invalidArg : string = "foo"
myfunction2(invalidArg) // 'invalidArg' is an illegal argument type.
                        // Expected int but string was given

myrecursivefunction(x)
```

Section 5: UML Diagram

The Catscript Multiplication Sequence Diagram demonstrates the interaction between various components of a simple arithmetic expression evaluation in the Catscript language. The diagram shows the flow of events when a user inputs an expression, "10 * 3", and the system processes it to return the result, "30".

The main components participating in the sequence are:

1. User
2. CatScript
3. Lexer
4. Parser
5. ParseTree

The process begins with the User providing an arithmetic expression to the CatScript component. In this case, the expression is "10 * 3". CatScript is responsible for coordinating the activities of the other components. Upon receiving the expression, CatScript activates the Lexer component to tokenize the input. The Lexer processes the expression and returns a set of tokens to CatScript. Next, the Parser component is activated by CatScript, which sends the tokens to the Parser for parsing. The Parser uses various parsing functions (e.g., `parseExpression()`, `parseFactorExpression()`, and `parseIntegerLiteral()`) to build a parse tree representing the input expression. Once the parse tree is created, CatScript receives it from the Parser and proceeds to execute it by activating the ParseTree component. The ParseTree component executes the parse tree and returns the result, "30", back to CatScript. Finally, CatScript delivers the result to the User, completing the process. The Catscript Multiplication Sequence Diagram effectively illustrates the series of interactions and steps involved in evaluating a simple arithmetic expression using the Catscript language. This diagram can be

useful in understanding the flow of information and control among the different components involved in the process.

Section 6: Design Trade-offs

In the development of the Cat Script parser for our capstone project, we faced a critical design trade-off: whether to create the parser by hand or to use a parser generator. We ultimately chose to hand-code the parser and employ a recursive descent algorithm to construct it. This decision was driven by our desire to gain a deeper understanding of the grammar and to tackle potential issues that may arise during the development process.

The recursive descent algorithm is a top-down parsing method that starts with the highest level grammar rule and recursively applies production rules to match the input string. This approach allows for greater control over the parsing process and enables the implementation of specific error handling mechanisms. Moreover, it offers a clear and structured representation of the grammar rules, which simplifies debugging and maintenance.

There were several advantages to hand-coding the parser. Firstly, it provided us with an enhanced understanding of the Cat Script grammar, which was crucial in identifying and addressing potential parsing challenges, such as dealing with ambiguous or left-recursive rules. Secondly, developing the parser manually gave us greater control over the parsing process, allowing us to fine-tune the parsing algorithm and implement custom error handling and recovery mechanisms. This led to a more robust and user-friendly parser. Thirdly, hand-coding the parser enabled us to optimize it specifically for the Cat Script language, potentially resulting in better performance compared to a parser generated by a general-purpose parser generator. Lastly, the process of creating a parser by hand served as a valuable learning experience, deepening our understanding of parsing algorithms, grammar rules, and language design principles.

However, there were also disadvantages to hand-coding the parser. Creating a parser from scratch is a time-consuming process, which could have been reduced by using a parser generator. Additionally, manually developing the parser increases the chances of introducing errors in the implementation, which may affect the parser's correctness and robustness. Furthermore, modifying the grammar in the future may require significant manual updates to the parser, whereas a parser generator can automatically adapt to grammar changes. In conclusion, the design trade-off of creating a Cat Script parser by hand using a recursive descent algorithm proved to be beneficial for our capstone project. The advantages, including enhanced understanding of the grammar, greater control over the parsing process, optimized performance, and the educational value, outweighed the drawbacks. This decision facilitated the development of a robust and efficient parser tailored specifically for the Cat Script language, leading to a successful Catscript Parser.

Section 7: Software development life cycle model

In our capstone project, we utilized the Test-Driven Development (TDD) model, which is an iterative software development process that emphasizes writing tests before implementing the actual code. Adopting this model played a significant role in ensuring the robustness and accuracy of our cat script programming language projects. Our professor supplied us with the initial tests, which provided a solid foundation for our work. Additionally, we created some of our own tests to challenge each other, ensuring that both projects met high-quality standards. The TDD model provided numerous benefits to our team, including improved code quality. Utilizing the tests supplied by our professor and creating our own tests allowed us to establish a strong basis for our programming language, ensuring that it met the desired specifications and behaved as expected. The model also enhanced collaboration among team members. By exchanging tests with one another, we were able to review and provide feedback on each other's work. This process facilitated learning and enabled us to improve the overall quality of our individual projects.

Another advantage of the TDD model was easier debugging. This approach helped us identify and fix bugs early in the development process. By running tests frequently, we could isolate issues and resolve them promptly, minimizing the impact on our project timelines. Moreover, the TDD model led to faster development, as we could develop our projects incrementally, ensuring that each new feature was thoroughly tested before moving on to the next. This approach reduced the likelihood of encountering major issues later in the development process and accelerated overall progress.

The test suite created during the TDD process, including the tests supplied by our professor and those developed by ourselves, also served as valuable documentation, making it easier to maintain and update our projects in the future. However, the TDD model also posed some challenges for our team. Writing tests before implementing code can be time-consuming, particularly at the beginning of the project. Luckily the professor provided most of these which mitigated some of that issue.

Another challenge was achieving complete test coverage. While we strove to create comprehensive tests, it was difficult to ensure that every possible scenario was covered. As a result, there may have been some edge cases that our tests did not account for.

In conclusion, the Test-Driven Development model greatly benefited our team in the development of our capstone projects. It allowed us to create robust, accurate, and maintainable cat script programming language implementations while fostering effective collaboration and learning from one another. Despite the initial time investment and potential for incomplete test coverage, we believe that the advantages of TDD, combined with the guidance of our professor and our own testing efforts, far outweighed the challenges.