

MONTANA STATE UNIVERSITY

CSCI 468 - Compilers
Carson Gross

Computer Science Capstone Portfolio

May 5, 2023

by
Max Kuttner

TABLE OF CONTENTS

	Page
SECTION I. PROGRAM	3
SECTION II. TEAMWORK	4
SECTION III. DESIGN PATTERN	5
SECTION IV. TECHNICAL WRITING	6
SECTION V. UML	11
SECTION VI. DESIGN TRADEOFFS	12
SECTION VII. SOFTWARE DEVELOPMENT LIFECYCLE	14

SECTION I. PROGRAM

1.1 Catscript Source Code

The full project code can be found within the source.zip file accompanying this portfolio.

Link: <https://github.com/Maximus1198/csci-468-spring2023-private/capstone/portfolio/source.zip>

SECTION II. TEAMWORK

This course is designed such that each student implements their own compiler, where traditionally it was done in teams. This decision ensures everyone has an equal opportunity to develop a compiler from start to finish, but requires team contributions be accommodated elsewhere. Peer discussions were facilitated through discord for anyone seeking help, while dedicated teamwork happened through test driven development and producing project deliverables. Team communications were done primarily through discord and email.

2.1 Team Member 1

Team member one was in charge of code development and maintenance.

Estimated Time Contribution: 80 hours or roughly 83%

2.2 Team Member 2

Team member two produced the Catscript documentation and supplied member one with auxiliary tests.

Estimated Time Contribution: 16 hours or roughly 17%

SECTION III. DESIGN PATTERN

Design patterns are a use tool that provide solutions to frequent programming problems and promote code readability and maintainability. One design pattern that was utilized in the Catscript compiler is a technique called memoization.

3.1 What is Memoization?

Memoization is a pattern used for optimizing the execution of a program. It works by saving the results of a heavy function then reusing that output when faced with the same inputs. The target function keeps track of what arguments it has seen, then returns the cached result when it sees a repeated one.

3.2 Implementation

The memoization pattern was utilized inside the CatscriptType class in the function called getListType(). This function returns the list type for a component type, e.g return list boolean for boolean and list int for int. The previous implementation returned a new listType for each function call. This would be costly over time and could be improved by storing each type once.

```
private static final Map<CatscriptType, CatscriptType> LIST_TYPES =
    new HashMap<>();

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

To implement this approach a HashMap is used to store the different catscript types for the duration of the program. When the getListType() function is called, the Map is queried for that type passed in the argument. If no entry exists for that type, insert and update the map, otherwise return the cached type.

SECTION IV. TECHNICAL WRITING

Introduction:

The Catscript scripting language is based on Java and is designed to be simple and intuitive. The program contains basic mathematical, logical operations, iterative statements, conditionals, functions and recursion. In the example below, we use a string variable to print hello world:

```
var str = "Hello World!"  
print(str)
```

TypeSystem:

Simple:

In Catscript, there are five primitive types: strings (same as java strings), booleans (true/false), integers (-2^{31} to $2^{31}-1$), objects (any type of value), and null types (null type). All Java types can be referenced by their underlying Java classes when comparing their assignability to primitives.

```
string // same as java strings  
int    // 32 bit integer ( $-2^{31}$  to  $2^{31}-1$ )  
bool   // true/false  
null   // null type  
object // any type of value
```

Complex:

Among the Catscript types, there is only one Intricate type, "list". The list type can contain multiple types, but it still maintains its invariance by preserving the relationship between each component type. This allows for the assignment of multiple types of data to a single list, as long as the component types are consistent.

```
list // list <int> list with integer values
```

Features:

Print Statement:

Using print statements, data may be displayed visually on a console or other output device. It works by taking some input value such as a string or integer and sending it to the terminal. This also allows you to display results and debug code. It is a helpful tool for developers as it makes it easier to track and debug code.

```
var str = "Hello World!"  
print(str)
```

```
>> Hello World!
```

If Statement:

An if statement is a programming construct that checks if a certain condition is met, and if it is, it executes certain code. If the condition is not met, the code is not executed. It is a way of controlling the flow of a program. By using multiple if statements, the programmer can create complex logic structures that respond differently depending on the program conditions. The "else" statement specifies what should happen if the if statement is not met.

```
var animal = cat  
  
if (animal = dog) {  
    print("Bark! Woof Woof!")  
}  
else {  
    print("Meow Meow! Purr!")  
}
```

```
>> Meow Meow! Purr!
```

For Loop:

Catscript for loop allows a user to iterate through a range of values. It can be used to execute a set of instructions multiple times, using a variable as a counter. This makes it ideal for performing an action multiple times on a range of values, such as looping through an array of items or printing out a set of numbers. Example Code:

```
for (i in [1,2,3])
{
    if (i<3)
    { print("true") }
    else
    { print("false") }
}

>> true
>> true
```

Variable Statement:

Statements that are used to represent a value or set of values that can be changed over time. They are designed to store data and can represent or manipulate values in a program. Catscript allows only one variable to be declared under a specific name at a time. Some Variables are declared implicitly and some explicitly. Written below are examples of implicit and explicit type of declaration. Example Code:

```
var animal : string = Lion //declared explicitly
var NumOfPeople = 10      //declared implicitly
var NumOfClasses = 4
```

It is not necessary to redeclare a variable's type when changing its value, and can simply be redefined like the following (keeping in mind that the new value is of the same type as the original declaration). Example Code:

```
NumOfPeople = 25
```


Mathematical expressions and non-void functions can also be used to set variables.

Example Code:

```
var NumOfPeople = 10 + 30
var NumOfStudents = NumOfPeople * NumOfClasses
```

Equality Expression:

In Catscript there are mainly 2 types of Equality Expression that can be used to see if the values of two variables (or) a variable and <type> entered in the *if* are equal.

NOT_EQUAL_TO (!=) and *EQUAL_EQUAL_TO* (==)

```
X = 6
if ( i == 6)    //EQUAL_EQUAL
{print(true)}
else if (i != 12) // NOT_EQUAL
{print(false)}
```

Comparison Expression:

There are 4 types of Comparison Expressions that can be used to compare two values.

GREATER_THAN(>), *LESSER_THAN*(<), *GREATER_THAN_EQUAL*(>=) and *LESSER_THAN_EQUAL*(<=)

```
X = 20
if ( i <= 5)    //LESS_THAN_EQUAL
{print(true)}
else if (i >= 30) // GREATER_THAN_EQUAL
{print(false)}
```

Function Statement:

In Catscript this enables a specific block of statements to be used repeatedly. One or more parameters may be passed to the function to provide information that will be used as variables within the body of the function. Following the variables are the parameter types, a return type may be declared, with a (:) followed by its type (<type>). Example Code:

```
function foo(x : bool) {  
    print(x)  
}  
foo(true)  
  
>> true
```

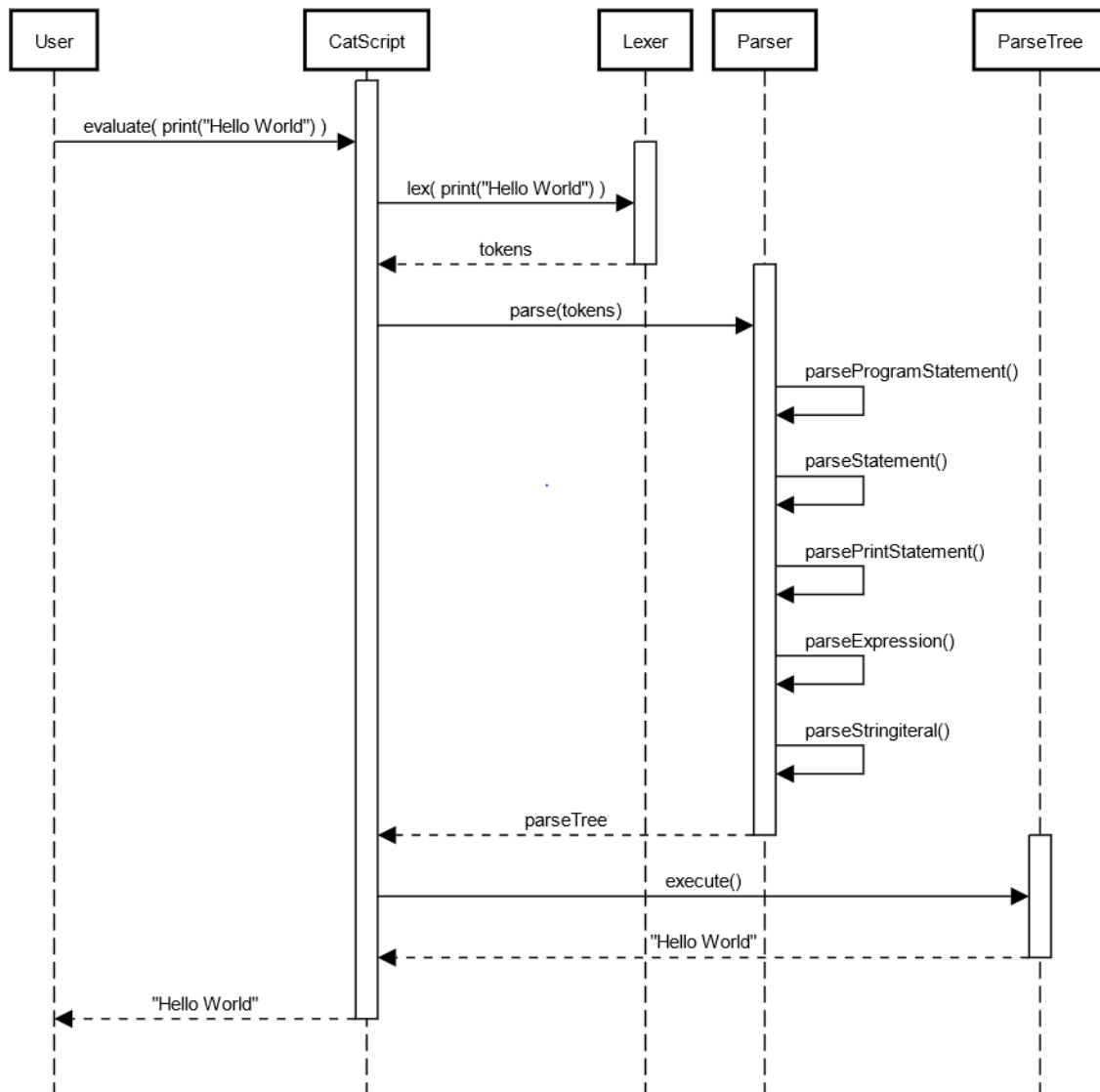
Return Statement:

An expression may be returned by a return statement at the end of the execution of a function. Once returned, control is returned to where the function was called. Example Code:

```
function max(a : int, b : int) : int {  
    if (a < b) {  
        return true  
    }  
    return false  
}  
print(foo(4, 10))  
  
>> true
```

SECTION V. UML

Catscript Print Statement Sequence Diagram



SECTION VI. DESIGN TRADEOFFS

Parsing is the magical step that gives meaning to random strings of characters. A hands-on parser implementation grants a deeper look at how languages work under the hood. A major design decision made for this project was choosing to build a parser from the ground up instead of using a parser generator. Both are popular industry methods for writing compilers and each has their own benefits and downsides. The intuitive and relatively simple nature of recursive descent parsing made it suitable for Catscript's small grammar.

6.1 Recursive Descent Parsing

Recursive descent parsing is a top-down method that is optimal when dealing with a plain and unambiguous grammar. Each production or rule can be represented via expressions and statements as a set of recursive functions. The program creates a parse tree by repeatedly descending from the top grammar until all non-terminals have been parsed. This approach fits nicely when built around Extended Backus-Naur Form and maps grammars to methods intuitively.

6.2 Parser Generators

Parser generators such as the look-ahead LR parser use bottom-up parsing which in contrast, start with the lowest grammar level and work up to the highest. Look head parsers can predict what the next input symbol will be, based on the current state and inputs. Generators are also better at handling ambiguity. They have less difficulty determining the correct parse tree given an input with multiple representations.

Bottom-up parsers are generally attributed to languages with large and complex grammars for their ability to help mitigate backtracking. However, recursive descent parsers are fast and extensible with the ability to support intricate error handling. Complex compilers such as GCC and Roslyn have implemented RDP's and shown its efficacy in production languages. Descent parsing is an excellent choice for Catscript in its current form. Although, if Catscript were to be fleshed out with a larger, more

complex and ambiguous grammar, it would be worth considering a bottom-up parser instead.

SECTION VII. SOFTWARE DEVELOPMENT LIFECYCLE

Development lifecycles are an important consideration for any large-scale project or application. In industry this means creating quality, low-cost software in as little time as possible. In the context of this course, quality was prioritized, leading to the design decision for Catscript to be produced using Test Driven Development.

7.1 Test Driven Development

Test driven development is a testing methodology that employs tests to define and verify code behavior. A test suite is created around the requirements of the project, where pass/fail checks are built for each functionality. The process of TDD boils down into three simple steps. First write a test and run it, then make changes to the code so that it passes, and refactor the code to make it right (if necessary). Benefits of test driven development include flexibility, improved collaboration, and improved code quality. Some drawbacks to TDD are the upfront development cost, learning curve, and maintenance requirements.

7.2 TDD in Catscript

Test driven development was utilized to build the Catscript language from start to finish. Each team began with a repository containing the basic outline for a compiler. Inside the repo was a test suite containing over 140 total tests, broken down into 4 project checkpoints. The four checkpoints were based around implementing the core concepts of the compiler, including tokenizing, parsing, evaluation, and bytecode generation. The team was responsible for gradually fixing all the tests before each deadline.