CSCI-468 COMPILERS

Capstone Portfolio

Timothy Bender

Sierra Stephens

April 20, 2021

Section 1 - Source Listing:

Source Code

All project source code and files are contained within the included Csci-468-spring2021-private.zip folder.

Development Environment

Catscript was developed within the Intellij Idea IDE ecosystem and compiled using a Maven script with JDK 14.

Section 2 - Teamwork:

Work on this project was divided into four primary sections: tokenizer, parsing, evaluation, and bytecode generation. While source code creation for each of these phases was done on an individual level, with each individual responsible for the creation of each distinct attribute of this compiler; cooperation between both individuals in the team was a vital part of testing, validation, debugging, and the creation of documentation. Both members of the team submitted appropriate tests to the other member for the purpose of validating and debugging the member's compiler.

Total Estimated Hours: 130

Member 1:

Primary software developer, code management.

Estimated Hours: 100 hours - 76.1% contribution in time.

Member 2:

Technical documentation, software tests, debugging.

Estimated Hours: 30hours - approximately 23.9% contribution in time.

Section 3 – Design Pattern:

The primary pattern used in the creation of Catscript was the Memoization Pattern. This pattern was used to increase the efficiency of the "getListType" function and reduce the number of new objects being created. By implementing this pattern, only one ListType is generated for each corresponding CatscriptType, this improves runtime speed by removing the need for redundant object creation, as well as lowering overall memory usage. To directly code this function means creating and storing a new ListType, as well as performing unnecessary comparison operations to determine the proper ListType to create.

The Memoization Pattern can be seen in the getListType function, within CatScriptType.java located at ~\csci-468-spring2021-private\src\main\java\edu\montana\csci\csci468\parser\CatscriptType.java. The pattern comprises the entire function, its control flow, and a static final HashMap used to store CatscriptTypes with their ListType equivalent.

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

Section 4 - Technical Documentation

Introduction

Catscript is a small statically typed scripting language that compiles to JVM bytecode featuring both evaluation and compilation. It is designed to take the best features of Java and combine it with features inspired from other languages to create an ideal language to work with.

Here are some features:

Features

Comments

Comments are used in Catscript using the "//" for single line comments and "/* */" for block comments

```
/*
This is
a fat comment
*/
var x = 8
//This is a single line comment
print(x)
```

Types

Catscript supports basic types for variables. These types can be both explicitly declared and inferred. Catscript also has a shared namespace, variable names from all fields and sub-scopes are stored in the same location. Here is the small list of Catscript's type system:

- int - a 32 bit integer
- string - 'Java style' string object
- bool - boolean value
- list - a list of objects
- null - a null type
- object - any value

For loops

Using the 'in' keyword like most other languages, for loops in Catscript are used to iterate through contents in an array as shown here:

```
var list = ["a", "b", "c", "d"]
for(i in list){
print(i)
}
```

This returns "abcd". i is assigned to each character within the array and printed as the loop iterates through the array.

If statements

If statements in Catscript are the same as those in Java. Using the if, else if, and else format, the user uses boolean literals or expressions that evaluate to a boolean as well as comparison operators to evaluate if statements.

```
var x = "yes"
if(x == "yes"){
print("accepted")
}
else if(x == "no"){
print("denied")
} else{
print("who are you")
}
```

The above returns the string "accepted"

Function Definitions

Functions are defined by including the keyword "function" before your identifier and parameters. The return type is void, unless otherwise stated.

Explicitly Typed

This function uses explicit types for the parameters and returns an int

```
function foo(num: int, truth: bool) : string {
//do something
return num
}
```

Type Inferred

This function has a default return type of void. It also uses type inference for the arguments.

```
function foo(num, truth){
//do something
}
```

Unary Expressions

Catscript uses the negate (-) unary operator on integers and booleans take the "not" keyword to negate its value

```
var x = 8
print(-x) //This will print "-8"
```

> not true //This evaluates to false

Comparison

Catscript uses the basic comparison operators less than, less than or equal, greater, greater than or equal, as shown below:

> 1 < 0 //false
> 1 <= 0 //false
> 1 > 0 //true
> 1 >= 0 //true

Equality

Catscript also check for equality using the basic equality operators on objects of the same type. The operands used are "equal equal" and "bang equal"

> 1 == 0 //false
> 1 != 0 //true

Variable Statements

Variables are declared using the "var" keyword

Explicitly Typed

You can explicitly define an object's type by including a ": < type >" after var but before your "="

> var num: int = 8
> var hello: string = "Hello World!"
> var truth: bool = false
> var nums: list<int> = [1,2,3,4,5]

Inferred Type

If you don't explicitly give your variable a type, the type will be inferred while the statement is parsed

> var num = 8 //int
> var hello = "Hello World!" //string
> var truth = false //bool
> var nums = [1,2,3,4,5] //list of type int

Catscript also allows for lists of type object with multiple dimensions:

> var list = [8,"Hello World!", [8,40,320]]

Print Statements

Catscript print statements are similar to those in other languages where the keyword "print" is used to return data to the user.

```
var num = 8
print(num) //prints out "8"
print("Hello World!") //prints out "Hello World!"
```

You can also use the "+" operator to concatenate integers to your strings

```
var num = 8
print("Hello player " + num)
//returns "Hello player 8"
```

Math Operators

Catscript uses basic division and multiplication operators to calculate integers

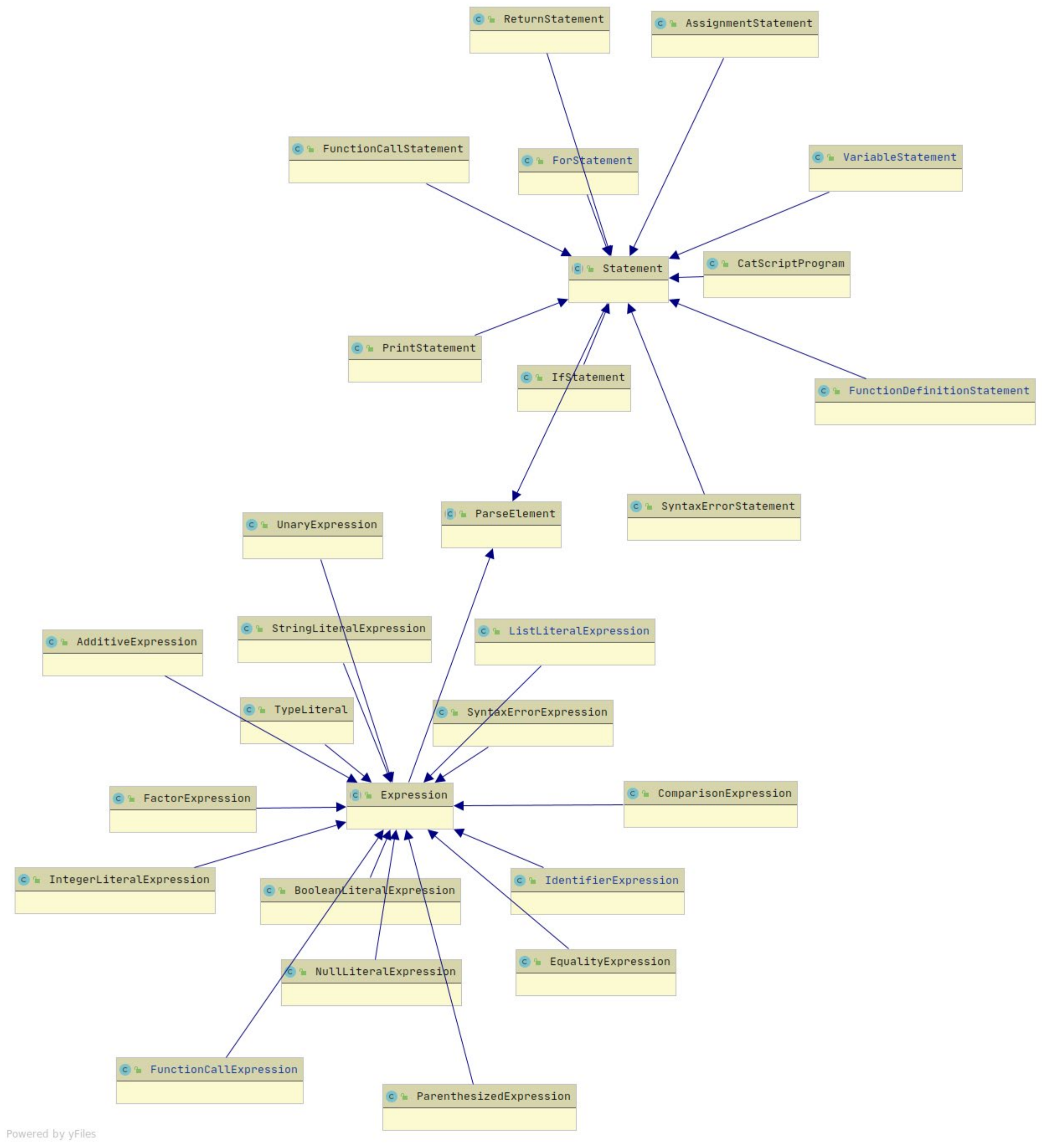
```
var x = 8
var y = 40
print(x*y) // returns 320
print(x/2) // returns 4
```

Catscript also uses basic addition and subtraction operators

```
print(8+40) //evaluates to 48
print(8-40) //evaluates to -32
```

Section 5 – UML

Catscript Expressions Parsing UML



Section 6 – Design Tradeoffs

The primary design tradeoff decision made for the Catscript project was the decision to use a Recursive Descent parser. Recursive Descent parsers feature several pros and cons when compared to other parsing solutions such as Bottom up parsing. To begin, recursive descent parsers are not the fastest method of parsing available and make error message generation difficult. In contract recursive descent parsers feature very simple implementation, and full featured parsers can be generated very quickly. Additionally, due to the nature of recursive descent parsers, it is exceptionally easy to add features within the parser.

Recursive descent parsers also express the natural recursive nature of a programming language's grammar, making that grammar almost parallel with the actual execution of the parser. This leads to easy understanding of the parser's functions by any programmer who can interpret a language's grammar. Recursive descent parsers also contain a generated parse tree, making it exceptionally easy to determine the nature of, and fix any bugs. Recursive descent parsers feature relatively fast execution while featuring simple implementation and expansion, making them the perfect choice for a time constrained project such as this one.

Section 7 – Software Development Life Cycle Model

The Catscript project was developed using iterative test development. Development was separated into four major milestones': Tokenization, Parsing, Evaluation, and Bytecode Generation. Completion of each of these milestones was measured by numerous Junit tests intended to determine the integrity of our software solution. Each individual test featured a section of Catscript source code, which was then passed through the appropriate Catscript functions and tested for validity.

The iterative test development model significantly aided this project by allowing the team to proceed at its pace, while ensuring that each step of implementation was performed correctly. By allowing the team members the option to work on the project when time was available to them, this approach reduced cramming, increased team member satisfaction and productivity and allowed for the parallel completion of other course work.