CSCI 468 CAPSTONE PORTFOLIO

Spring 2024

William Cade -Team Member 1
Lucas Rowsey – Team Member 2

Section 1: Program

See attached zip file in /capstone/portfolio/source.zip

Section 2: Teamwork

For the development of the Catscript compiler, each team member creates a brief documentation of the Catscript language. In addition, each team member designs three individual tests with the objective of assessing various parts of the other team member's compiler.

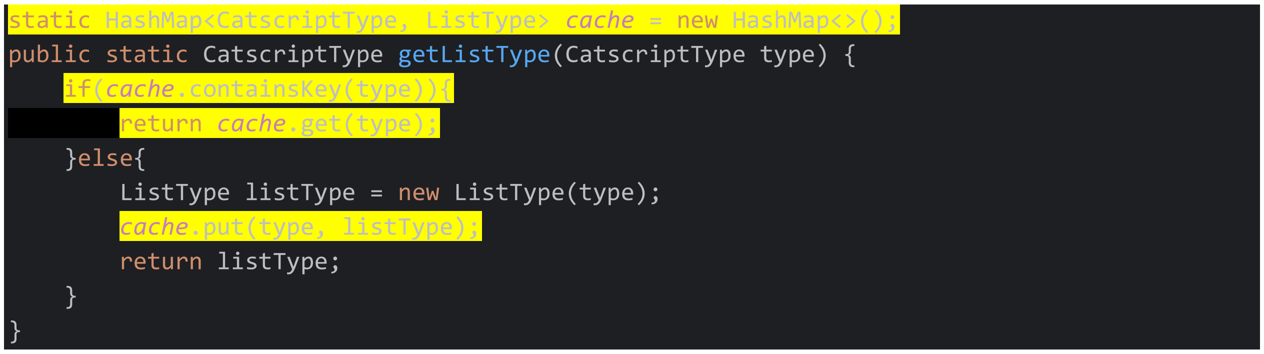
Team member two's tests were made to gauge the functionality of team member one's list literal expressions containing unary and boolean values, and if statements using expressions and function calls. The first test checks a variety of parts of the list literal expressions, including the size of the expression, and the proper usage and computation of unary and boolean values within the list. The second test evaluates whether expressions can be used properly as conditional expressions for if statements. The third test also pertains to if statements, and if function calls can be properly parsed and executed within the statement body.

Section 3: Design Pattern

The design pattern implemented in our Catscript compiler program is the process of memoization. Memoization is a technique used to optimize the program by cutting down on unnecessary or redundant function calls by storing data in a HashMap or an array, so that those results can possibly be reused if the same function call occurs more than once. The process of memoization starts with a recursive function call, When the function is called, it first checks a HashMap to see if the function call has already been made and stored. In the case that the function call has already occurred, the result will be returned directly from the HashMap, rather than needlessly re-computing the function. If the result is not found in the HashMap, then the function is computed normally, and then stored in the HashMap for future function calls.

For the purpose of our program, memoization has been implemented in the getListType function in *CatscriptType.java*. The usage of the memoization design pattern is to reduce unnecessary function calls and improve performance rather than running the same code multiple times over.

Included below is a snapshot of the design pattern as it appears in our code:



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

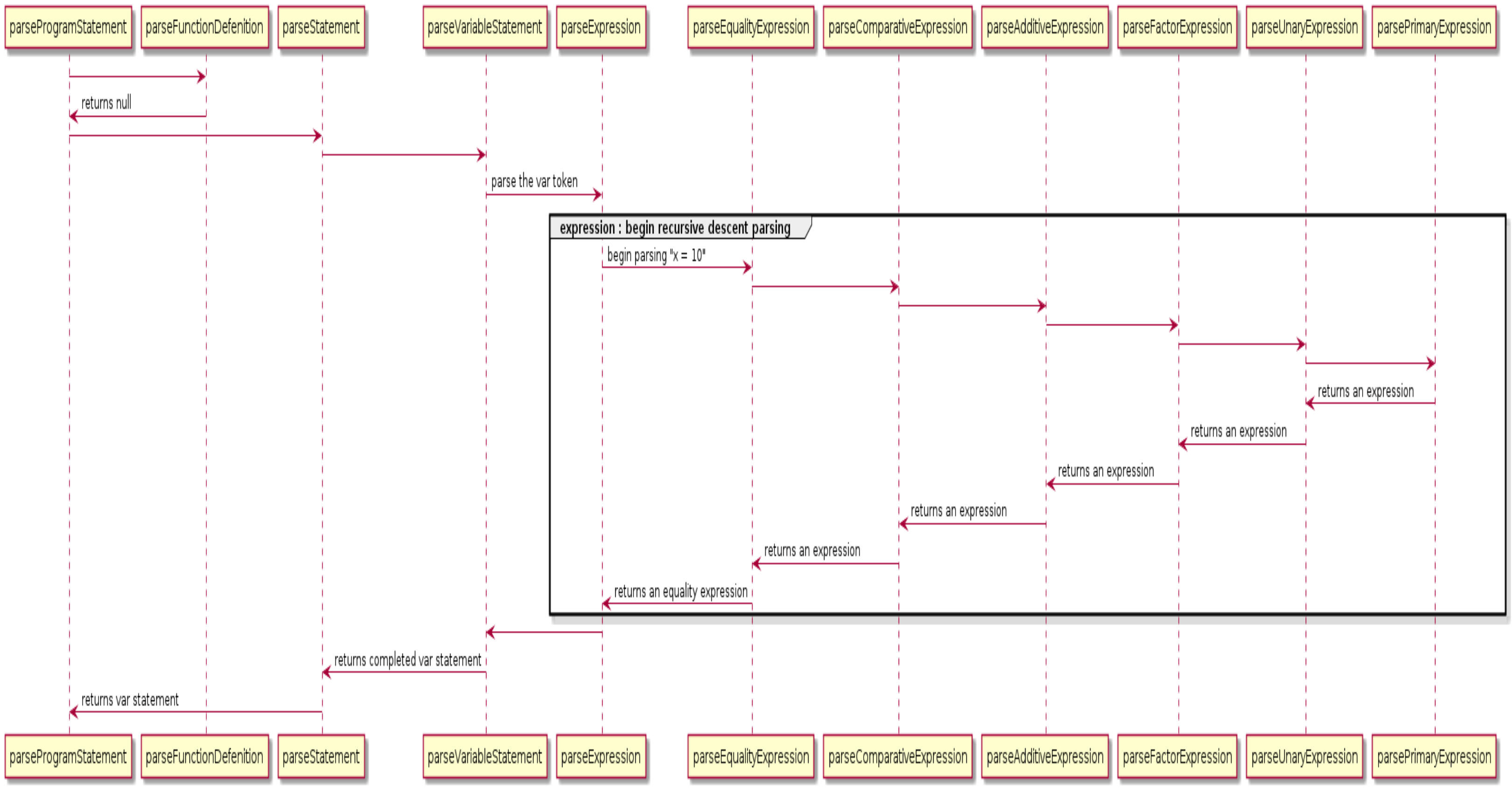
The design pattern, highlighted above in yellow implements a HashMap that stores a key-value pairing of Catscript Type, and List type. First the program checks if the function output already exists within the HashMap, and if not then it computes the list type as expected and stores the response within the HashMap for future calls.

Section 4: Technical writing

Included at the end of this document is the catscript documentation.

Section 5: UML

Below is a UML sequence diagram detailing function calls during the parsing process for the variable statement “var x = 10”.



Section 6: Design trade-offs

Recursive descent parsing and parser generators are two different applications for the creation of a parser component in the Catscript compiler. In the Catscript compiler, the parser was designed using recursive descent parsing, rather than a parser generator. Comparatively, a recursive descent parser is easier to implement and troubleshoot, as well as being able to complement the recursive nature of the Catscript grammar. A parser generator is a tool that can take a BNF grammar rule set and automatically generate a powerful and efficient parser. However, the decision to develop a recursive descent parser not only gives the developer a deeper understanding and higher level of control with parsing, but also works well with the simplicity of the catscript language. While a parser generator can generate a parser for large and complex grammars, this feature is not especially necessary when considering the simplicity of the Catscript language. In addition, parser generators are complex for newer programmers due to difficult syntax, a harsh debugging process, and overall giving the programmer less control of the parsing process. Instead, the recursive descent approach does the exact opposite: a simpler implementation means that there is less unnecessary struggle with debugging and error checking, and the development is closer to the inner workings of parsing. For all intents and purposes, recursive descent parsing strongly suits the niches of the Catscript language and the development of its compiler, at the cost of some program efficiency. To summarize, each application of parser development offers unique advantages and trade-offs. The recursive descent approach better suits the scope of the compiler project, as the catscript language is a simple scripting language that does not suffer from the loss of efficiency.

Section 7: Software development life cycle

The software development cycle for the Catscript compiler was based on Test Driven Development (TDD). Before the project was started, there was a large test suite built around assessing the completion of the compiler. At each step of the way, these tests were run to both verify that functions run as expected, but also to troubleshoot and debug during development. As a software development cycle, Test Driven Development prompts the developer to design code at making tests pass, and only making modifications or additions to the code where the tests fail. This approach ensures that there is not an excess of unnecessary code.

In the context of the Catscript compiler, there were 4 checkpoints, with each checkpoint holding a different focus for the compiler. First was tokenization, the second was expression parsing, and then statements and evaluation, and last was bytecode. Every checkpoint had a diverse test suite based around the assessment of each part of the compiler. This development model was highly useful in the completion of the compiler, as it gave direct insight into what parts work well and what parts don’t. From there, debugging was simple since the exact problems with the program were known.

Catscript Guide

This document should be used as a guide for catscript

Introduction

Catscript is a simple scripting language that is easy to read and understand. It supports programming functions such as control flow statements, functions, variables, and different data types.

Here is an example:

```
function foo() : list { return [1, 2, 3] }
print(foo())
```

Features

For Statement

A for statement is a control flow statement that is used to execute an expression a specific number of times. The number of times the code is repeated is based on the expression put into the for statement. An identifier is set to the value of the expression. One implementation of this is to put in a list of integers and print out the values.

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

If Statement

An if statement is a control flow statement that evaluates an expression and executes expressions based on the results. In the case that the expression is true a true statement is executed, if the expression is false the true statement is not executed. An else statement can also be added, in the case of the expression being false if there is an else statement it will be executed.

```
if(true){ print(1) }
if(true){ print(1) } else { print(2) }
```

Print Statement

A print statement outputs the result of an expression.

```
print(1+1)
```

Var Statement

A var statement is given an identifier declares and initializes a variable with the value of an expression. A var statement can be given a type or can be given an explicit type, if an explicit type is not given it will assign the type implicitly.

```
var x = 10
var x : int = 10
```

Assignment Statement

An assignment statement takes an identifier and sets the value of that variable to the value of an expression.

```
x = null
```

Function Call Statement

A function call calls a function name and an argument list. The function will then run the function with the list of arguments given.

```
foo(1, 2, 3)
```

Function Declaration

A function declaration creates a new function with a list of parameters and body, a function can be given a return type. The function body consists of a statement or statements that the function will run when it is called.

```
function foo(x, y, z) { print(y) }
function foo() : object { return true }
```

Return Statement

A return statement evaluates an expression and sets that as a function's value when called.

```
function foo(x : int) : int { return x + 1}
print(foo(9))
```

Equality Expression

An equality expression is used to check for the equality of two values. An equality expression can check if the values are equal (==) or not equal (!=) depending on the operator. An equality expression will evaluate as a boolean the expression will be true if the values given are equal for == and not equal for !=. The expression will be false if the values given are not equal for == and equal for !=.

```
1 == 1
1!=1
```

Comparison Expression

A comparison expression is used to compare different values. A comparison expression will evaluate to a boolean based on the values and the operator. A comparison expression will be true if the first value is greater than the second for >, the first value is less than the second for <, the first value is greater than or equal to the second for >=, and the first value is less than or equal the second for <=. A comparison expression will be false if the first value is less than the second for >, the first value is greater than the second for <, the first value is less to the second for >=, and the first value is greater than the second for <=.

```
2 > 1
2 >= 1
2 <= 1
2 < 1
```

Additive Expression

An additive expression is used to add or subtract values. An additive will evaluate to a value that is value based on the operator used. A (+) operator will add integer values together and combine strings, while a (-) operator will subtract values.

```
1+1
1-1
1+1-2
```

Factor Expression

A factor expression is used to multiply or divide values. A factor expression will evaluate to an integer that is value based on the operator used. A * will multiply integers and / will divide integers.

```
1*2
4/2
9/3*2
```

Unary Expression

A unary expression is used to change a value. Not will flip a boolean value and - will make an integer negative

```
not true
-1
```

Type Expression

A type expression evaluates the data type.

```
int
string
```