CSCI 468: Compilers Capstone Portfolio

Jonathan Neuman, Cameron Oberg

Spring 2024

1 Program

Included in this directory is a compressed directory containing the source code for the project.

2 Teamwork

Team member 1 and team member 2 each designed three test cases for testing Catscript as well as the technical document describing Catscript and exchanged them with each other. All of the code was written individually by each member for their own projects, which were individual assignments. This code was implementing the Catscript tokenizer and parser, as well as the evaluation and compilation methods required to create the Catscript interpreter and compiler. Below are the tests that member 2 wrote for member 1 to test their code with.

```
    public class PartnerTest extends CatscriptTestBase {
    //TODO Are nested list elements evaluated properly?
    @Test
    public void listElemtsEval(){
        ListLiteralExpression ll = parseExpression("list a = [1, (1+3),5]]");
        String expected = "[1, 4, 5]";
        assertEquals(expected, ll.getValues());
    }

    //Concatenate nested String and int?
    @Test
    public void concatAdv(){
        Expression ex = parseExpression("(Hello+5)+(10/2)");
        assertEquals(CatscriptType.STRING, ex.getType());

    }

    //Can the parser support function calls within a function call?
    //This test checks to ensure that the arguments
    //within the function are evaluated:
    @Test
    public void nestedFuncCall(){
        FunctionCallExpression fcall = parseExpression("bar(foo(),1)");
        assertEquals(2, fcall.getArguments());
    }
}
```

3 Design pattern

A design pattern I used in my project is the memoization pattern. Memoization is used when a method is frequently called with the same arguments, leading to the same result. Instead of having to rerun that method each time, you can use memoization to cache the results and return the cached results instead. Below is where I implemented the memoization pattern in my code. The getListType method is called each time the type of a list is checked, which happens frequently in the evaluation step of the compiler. This method originally would return a new ListType object each time it was called, which was inefficient. Instead, now the code uses the memoization pattern and will cache any new ListTypes that are created. Subsequent calls of the getListType method will return the cached result instead of creating a new ListType.

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

4 Technical writing

Catscript Guide

Introduction

Catscript is a simple scripting language. Here is an example:

```
var x = "foo"
print(x)
```

Features

Additive and Factor Expressions

CatScript supports basic arithmetic expressions. Like other high level programming languages, it supports addition, subtraction, multiplication and division.

```
1+1
15-5
3*3
12/2
```

CatScript also supports the + operator on strings as well. Doing this concatenates them:

```
string a = "A"
string b = "yellow"
string c = "bus"
string d = a + b + c
```

Primitive Types

CatScript supports the following types: integers, strings, booleans, lists, objects and null. Here is an example of how one may wish to define them:

```
int a = 2
string b = "Hello World"
boolean c = true
list d = [1,5,9]
```

```
object e = null
```

Where an object can be any value, and null represents the lack of a value.

Unary and Conditional Operators

CatScript supports the ! and − operators. Additionally, the standard conditional less than <, less than or equal to <=, greater than >, greater than or equal to >=, not equal to != and equal to == are all supported.

Print Statements

CatScript print statements display a value to the screen. Here are some examples of its use:

```
string y = "Red"
int x = 4
print(x)
print("This value is: "+x)
print("A " + y + " apple")
```

As seen in the last example, string concatenation can also occur within a print statement.

Variables

CatScript also features type inference with the var keyword. Here are some examples:

```
var x = 5
var y = "Type inference is a nice feature"
```

Where x will be an integer and y will be a string.
For var expressions of a list type, type casting is also possible:

```
var a list<int> = [1,2,3]
var b list<string> = ["Strawberry","Pineapple","Mango"]
```

For loops

In CatScript, for loops can be defined as follows:

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

Just like other languages, some element will iterate over a collection of objects and execute statements within the local scope, denoted by the brackets ''. This will continue until every element in y has been reached.

If Statements

The If conditional statement is supported in CatScript. Here is an example:

```
if(x > 10){
    print(x)
 } else if (x == 5){
   print("Alternative branch")
 } else {
   x = 10
 }
```

Example without any else ifs:

```
if(x > 10){
    print(x)
} else {
    print(x+ "Was not greater than 10.")
}
```

Several else ifs are also supported.

```
if(x > 10){
    print(x)
} else if (x == 5){
    print("Alternative branch")
} else if (x == 4){
    print("Alternative branch1")
} else if (x == 3){
    print("Alternative branch2")
} else {
    x = 10
}
```

Like If statements from other programming languages, the else if and else branches are not required.

```
if(x > 10){
    print(x)
}
```

Functions

Likewise, functions can be defined as seen below:

```
function bar (){
    print("Hello World")
}

function foo(a : int, b : int) : int{
    int c = a+b
    return c
}
```

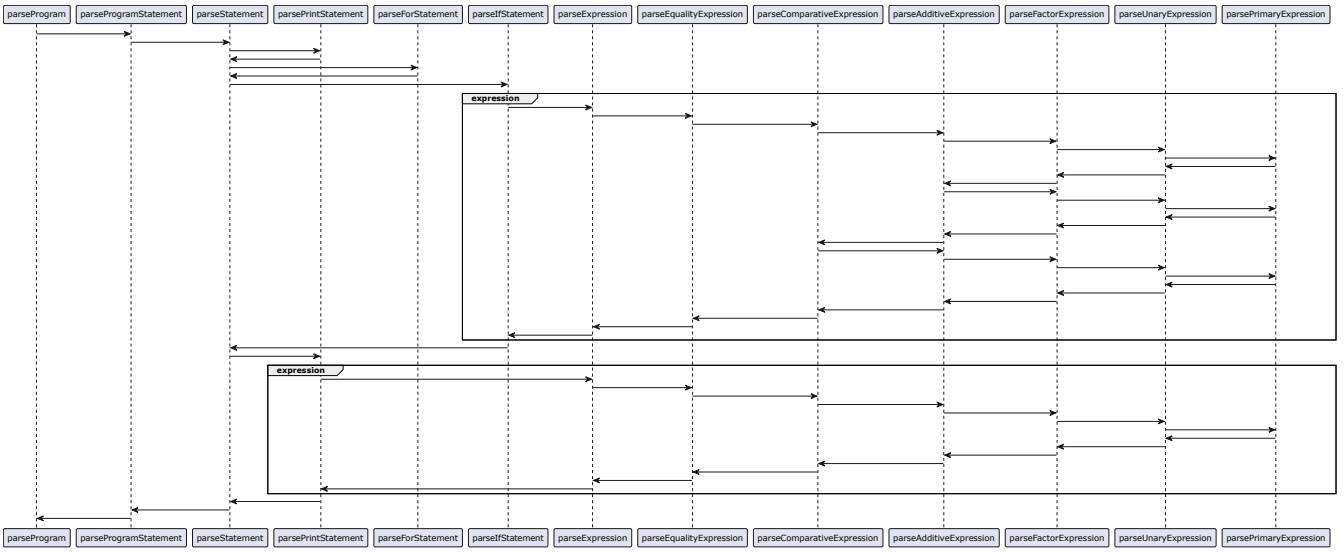
Where the first example gives a function without a specified type or return, and the second demonstrates a more complex definition complete with strict typing and a return type. Likewise, a function call appears as follows:

```
bar()
foo(1,2)
```

5 UML

Included is a UML sequence diagram that shows how the parser parses the following Catscript statement:

```
if(x + 10 > 20){
    print(x)
}
```



This UML sequence diagram shows how the recursive descent parser parses source Catscript code. The parser will start by calling the parseProgram method, which will call the parseProgramStatement method. From here, the tokenized Catscript code is sent through each type of Catscript statement parse method (printStatement, forStatement, ifStatement, etc...), until a valid statement is found. If no valid statement is found, a parse error is returned instead.

In this case, the Catscript code starts with an if statement so once the parser reaches the parseIfStatement method, the method will recognize the beginning of the if statement and begin parsing it. It does this by consuming the token "if" and the next token "{" and then parsing the next token as an expression. Expression parsing is where the recursive nature of the recursive descent parser is most obvious. When parsing an expression, the expression is recursively parsed until it is parsed down to a literal expression such as an integer or String literal. These literal values are then returned back up the chain of recursive method calls to the original method. In the case of this example, the "x" expression will be parsed down into whatever type literal it evaluates to, (in this case an integer), and then this value will be returned up the recursion chain until the parseAdditiveExpression method is reached. This method will see that the next token is a "+", so it will consume that operator and then parse the right side of the expression. In this case, that expression is the comparative expression, $10 > 20$. Once that expression is fully parsed, the parseIfStatement method will continue by parsing each statement within its body. This will continue until the entire statement is parsed.

6 Design trade-offs

The main design trade-off considered in this project was the use of a recursive descent parser or a parser generator to parse the tokenized Catscript code. Recursive descent is a method of parsing that uses recursion to generate an abstract syntax tree from an array or list of tokens. Recursive descent is accomplished by writing a method for each type of production in the language's grammar, then within that method, recursively call the method from the right hand side of that production in the grammar. These methods will recursively go through each production in the grammar, parsing the token when applicable. The end result will be a fully parsed abstract syntax tree. The main benefits of recursive descent parsing include gaining a deeper understanding of the recursive nature of programming language grammars, and learning to implement one is valuable experience since most parsers written in industry are recursive descent parsers. The big downside of recursive descent parsing is the amount of code that needs to be written to create one. The parser itself, along with the infrastructure required to make it work, requires thousands of lines of code, far more than other methods of creating a parser.

Parser generators are programs that take in a lexical grammar of a programming language, either in the form of regular expressions or extended Backus–Naur form (EBNF), and generates a parser for that language. This parser is used in the same way the recursive descent parser is used, an input array or list of tokens from the tokenizer is fed to the parser, and the parser will parse the tokens and return an abstract syntax tree. The main benefits of using a parser generator to make your parser are having to write less code to make the parser, and less infrastructure is required to make the parser work correctly. The downsides of the parser generator approach are the fact that a parser generator parser is less efficient than a hand written parser. Additionally, parser generator parsers are not that common outside of academia, in industry, recursive descent parsers are much more common.

The parser in this Catscript compiler was written using the recursive descent technique. This was chosen over the more common approach of using a parser generator for a number of reasons, mainly for learning how recursive descent parsing works and also the prominence of recursive descent parsers in industry makes learning how to make one a good choice for preparing for a career in technology.

7 Software development life cycle model

This project was developed using Test Driven Development (TDD). Tests were written before any code was written, allowing for automated testing and test coverage for any new features added to the project. This model was helpful to the development as having the tests made it easier to debug the codebase. Test Driven Development also made it easier to have a more comprehensive view of the project, knowing which features needed to be implemented in advance by looking at what tests were written.

The tests for the project are written using JUnit, which is a testing library built for testing Java code. JUnit allows for automated testing of the code for which the tests are written for. For example, JUnit can be configured to automatically run tests on source code when the code is pushed to Github. This allows for easier verification of source code when using version control such as Github. In the case of this project, Carson Gross implemented a custom autograder script, which would run the tests for each checkpoint of the project on the code that I had pushed to my Github repository, which let me know how many tests were passing.