Cooper Senior
CSCI 468
04/30/2024

Compilers Capstone Portfolio

Section 1: Program

A source.zip file of the final repository can be found in the capstone directory.

Section 2: Teamwork

I worked with one teammate to complete the capstone project. He was responsible for creating the technical writing document explaining the features of the Catscript programming language. This document didn't take very long to complete in comparison to the code implementation. He was also tasked with creating addition unit tests to verify the accuracy of my implementation of the compiler. He created 3 additional tests that I ran against my code to see if any additional bugs could be found. These tests followed existing format, so they didn't take very long to create.

I was responsible for the whole implementation of the compiler. This majority of the time to complete since there was a lot of work to be done throughout the semester. The first section was to work with the skeleton code and complete the tokenizer. The next step was completing the implementation for the parser. Once those basic sections were completed the next tests to complete involved evaluating the expressions using the tokens and parse elements generated by the functions written previously. The last step of the parser was generating bytecode primarily by writing a compile method on each of the expression and statement classes.

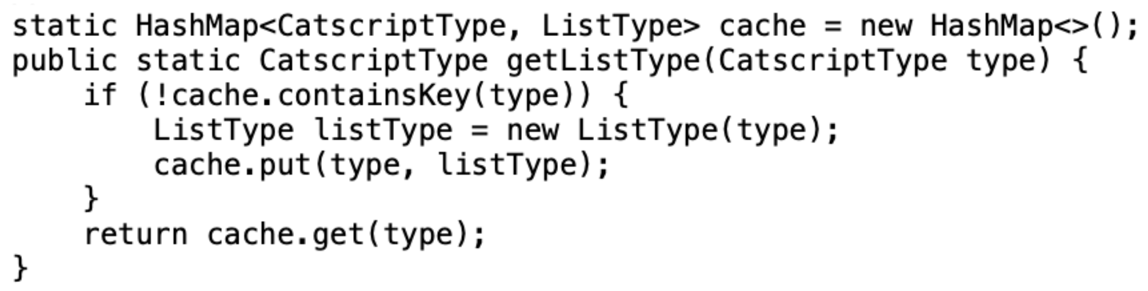
Partner Provided Tests:

```
package edu.montana.csci.csci468.demo;

import edu.montana.csci.csci468.CatscriptTestBase;
import edu.montana.csci.csci468.parser.CatscriptType;
import edu.montana.csci.csci468.parser.expressions.*;
import edu.montana.csci.csci468.parser.statements.*;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

public class CapstonePartnerTest extends CatscriptTestBase {

    @Test
    public void parseDoubleParenthesizedExpression() {
        FactorExpression expr = parseExpression("5 * (4 * (2 + 3))", false);
        assertTrue(expr.isMultiply());

        //check outer paren is parsed properly
        ParenthesizedExpression exprRight = (ParenthesizedExpression) expr.getRightHandSide();

        // get expression from paren
        FactorExpression exprRightIn = (FactorExpression) exprRight.getExpression();

        // check inner paren is parsed properly
        assertTrue(exprRightIn.getRightHandSide() instanceof ParenthesizedExpression);
        assertTrue(exprRightIn.isMultiply());
    }

    @Test
    public void forStatementWithIfStatementParses() {
        ForStatement expr = parseStatement("for(i in [6, 5, 4]){ if(i <= 5) { print(\"less\") } else
{ print(\"greater\") } }");
        assertNotNull(expr);
        assertEquals("i", expr.getVariableName());

        // verify instance types
        assertTrue(expr.getExpression() instanceof ListLiteralExpression);
        assertTrue(expr.getBody().get(0) instanceof IfStatement);
        assertEquals(1, expr.getBody().size());

        // check if statement body's
        assertTrue(((IfStatement) expr.getBody().get(0)).getTrueStatements().get(0) instanceof PrintStatement);
        assertTrue(((IfStatement) expr.getBody().get(0)).getElseStatements().get(0) instanceof PrintStatement);
    }

    @Test
    void functionWithParametersAndReturnType() {
        FunctionDefinitionStatement expr = parseStatement("function x(a : int, b : bool) : int  {return a}");

        //check params
        assertNotNull(expr);
        assertEquals("x", expr.getName());
        assertEquals(2, expr.getParameterCount());
        assertEquals("a", expr.getParameterName(0));
        assertEquals("b", expr.getParameterName(1));
        assertEquals(CatscriptType.INT, expr.getParameterType(0));
        assertEquals(CatscriptType.BOOLEAN, expr.getParameterType(1));

        //check return
        ReturnStatement returnStmt = (ReturnStatement) expr.getBody().get(0);
        assertNotNull(returnStmt);
        assertTrue(returnStmt.getExpression() instanceof IntegerLiteralExpression);
    }

}
```

Section 3: Design pattern

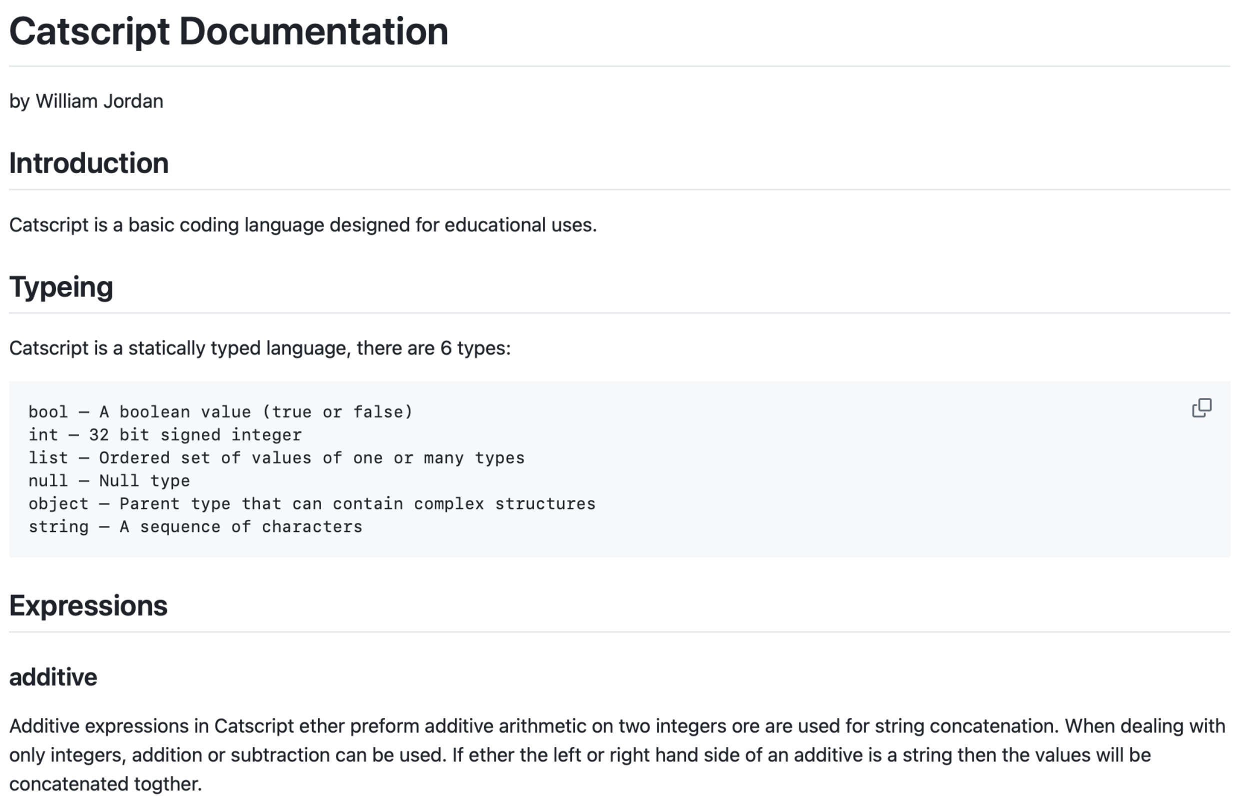
One design pattern I used in the implementation of the compiler was Memoization Pattern. This was used to memoize calls to CatScriptType getListType(). I used this pattern so that I could cache the types into a HashMap so that frequent calls to getListType() would be more efficient and wouldn't have to be created repeatedly if they already existed in the current scope. The Memoization Pattern is an appropriate choice for this scenario since it ensures that the method does not execute more than once for same inputs by storing the results.

Below is my implementation for the Memoization pattern. This code segment can be found in CatScriptType.java.



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

Section 4: Technical writing.



Catscript Documentation

by William Jordan

Introduction

Catscript is a basic coding language designed for educational uses.

Typeing

Catscript is a statically typed language, there are 6 types:

```
bool — A boolean value (true or false)
int — 32 bit signed integer
list — Ordered set of values of one or many types
null — Null type
object — Parent type that can contain complex structures
string — A sequence of characters
```

Expressions

additive

Additive expressions in Catscript ether preform additive arithmetic on two integers ore are used for string concatenation. When dealing with only integers, addition or subtraction can be used. If ether the left or right hand side of an additive is a string then the values will be concatenated togther.

```
//addition
1+1        // 2

//subtraction
2-1        // 1

//mixed type concatenation
"a" + 1     // a1

//string concatenation
"a" + "b"   // ab
```

factor

Factor expressions in Catscript ether preform multiplication or division on two integers.

```
//multiplication
4 * 2   // 8

// division
4 / 2   // 2
```

comparison

Comparison expressions in Catscript evaluate inequalities between integers. Using < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to) a boolean value is returned. This expression allows for expanded control flow options when used in an if statement.

```
// greater than
2 > 3       // true

// less than or equal with a var
var x = 3
x <= 1      // false
```

list literal

List literal expressions in Catscript are how lists are defined. A list can contain all of one type, sub lists or mixed types. Lists allow for programmers more efficient and higher level ways or storing and organizing data.

```
//all ints
x = [2, 2, 3]

//list of lists
x = [[1, 2] [3,4]]

//mixed list
x = [1, true, "a"]
```

equality

Equality expressions in Catscript checks where to values are equal or not equal to each other. Using the == for equality and != for not equal to programmers can check the values of variables or see if two variables are equal.

```
// equality check
1 == 1      // true

//variable equality check
var x = 1
x  != 2     // false

//variable compared to variable
x == x      // true
```

parentheses

In Catscript parentheses within an expression allows for overriding traditional order of operations and have part of an expression evaluate first in Catscript. This allows for programmers to implement much more complicated math expressions and adjust the values of variables more precisely.

```
//parenthesized expression
4 * (2 + 2)     // 16
```

unary

Unary expressions in Catscript evaluate to the negative value of an integer or the opposite value of a boolean. This allows for programmers to efficiently manipulate variables within their code.

```
//negative integer
-1

//negative variable
-x

//flip boolean
not true
```

Statements

print

In CatScript, print statements are used to display information to the user. It's a fundamental tool for communicating information from the program to the user, facilitating interaction and conveying results or messages during program execution. Programmers can pass variables, strings, or expressions to a print statement.

```
//variable
print(x)

//expression
print(1+2)

//string
print("hello world!")
```

if

In CatScript, if statements are fundamental for controlling program flow. Each if statement includes a boolean expression, which determines whether the subsequent code block executes. If the expression evaluates to true, the body of the if statement is executed. Optionally, an else statement can be included to handle cases where the boolean expression evaluates to false. This allows for conditional execution of different code blocks based on specific conditions within the program.

```
//basic if statement
if(true) {
    print(1);
}

// if statement with else
if(1 == 2) {
    print(1)
} else {
    print(2)
}
```

for

For statements in CatScript iterate over a sequence of elements, such as lists or ranges. They execute a block of code repeatedly, each time with a different element from the sequence. This facilitates tasks like processing each item in a list or executing a block of code a predetermined number of times.

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

function

In CatScript, function statements are essential for organizing and encapsulating code. Each function typically includes a block of code that performs a specific task or calculation. When invoked, the function executes its code block and may optionally return a value using a return statement. Parameters can be specified to functions to customize their behavior as well as return types.

```
//basic function
function foo() {
    print("hi")
}

//function with a paramater
function foo(x: int) {
    print(x)
}

//function with paramaters and type
function foo(x: int, y: int) : int {
    return x + y
}
```

return

In CatScript, return statements are fundamental for controlling program flow. Each return statement typically includes a value or expression to be returned from a function or method. When executed, the return statement immediately exits the function or method and passes the specified value back to the caller. Optionally, multiple return statements can be used within a function or method to handle different conditions or scenarios. This allows for conditional return of values based on specific conditions within the program.

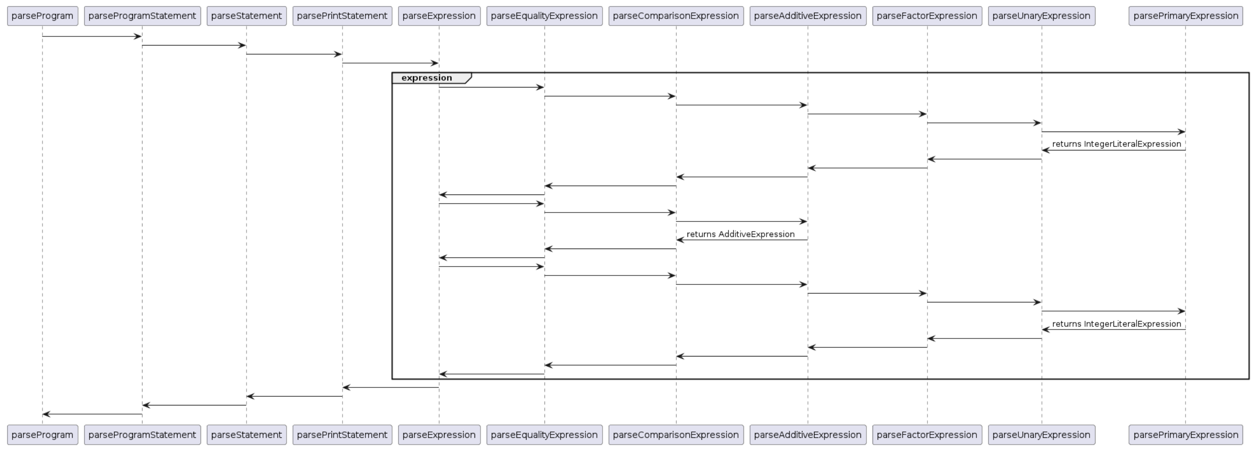
```
//variable
function foo(x: int) : int {
    return x
}

//string
function foo(x: int) : int {
    return "success"
}

// return just to exit
function foo(x: int) : int {
    return
}
```

var

In CatScript, var statements define and initialize variables. They consist of declaring a variable name and optionally a type and or an initial value. These statements allow for dynamic data management, memory allocation, and state maintenance within the program.

```
//var no type specified
var x = 1

//var with type
var x : int = 10
```

Section 5: UML.



This is the sequence diagram for parsing the expression “print(1 + 1)”. The sequence diagram is effective for showing how recursive decent works in the compiler. The program calls parseProgram, parseProgramStatement, parseStatement then parsePrintStatement. Then it recursively calls the expressions searching for a match to the first element. Since the first element in the expression is 1, it is a primary expression. This section starts by calling parseExpression followed by parseEqualityExpression, parseComparisonExpression, parseAdditiveExpression, parseFactorExpression, parseUnaryExpression and finally parsePrimaryExpression, where a match is found. An integerLiteralExpression is returned down the stack until parseExpression since this method loops until all the tokens in the expression are matched to a literal.

The next token is a plus, so it follows the same method calls going from parseEqualityExpression to parseComparisonExpression and then parseAdditiveExpression where the token is matched to a plus. This returns an additiveExpression down the stack back to parseExpression. The last element is another 1 so it follows the same logic as before. This once again returns an integerLiteralExpression down the stack back to parseExpression. Since all the elements in the expression have been tokenized, it recurs back to the original call to parseProgram.

Section 6: Design trade-offs

When it comes to designing a compiler, one of the fundamental decisions is choosing between using a Parser Generator or Recursive Descent parsing techniques. Parser generators, such as YACC and ANTLR offer a high-level of abstraction and automation in generating parsers from a formal grammar specification. These parsers design benefit from being able to handle complex grammars efficiently and often produce parsers with good performance. However, they come with the overhead of learning and configuring the tool, which may require more time and effort. These generated parsers can also be difficult to debug and modify, since we may be unfamiliar with the underlying generator.

The other design option is Recursive Descent parsing which involves hand-writing the parser code based on the grammar rules of the language. While this approach requires more manual effort upfront, it offers several advantages. Recursive Descent parsers tend to be more transparent and easier to understand, as they directly reflect the grammar rules in the code. This makes them easier to debug and modify, as we have full control over the parsing process. Additionally, Recursive Descent parsers are often more flexible, allowing for easy integration of error handling and special cases into the parsing logic.

Since we were given the CatScript programming language grammar we used recursive descent for our parser so that we could have more flexibility and understanding of the code that we were writing. Recursive Descent parsing provides clarity, flexibility, and ease of debugging which works well with our project since we were using Test Driven Development which relies heavily on debugging to write the methods and investigate errors. Since we were given the structure for the parser and had to work with existing code, it was best to use a design that would allow for the most flexibility and understanding so that we could implement the parser correctly. It is also worth noting that the CatScript programming language is relatively simple, so we didn’t require using a parser generator which is more apt to handle complex grammars.

Section 7: Software development life cycle model

While implementing our parser, we followed the Software Development Life Cycle. This helped us navigate through the complexities of building our project step by step. One important method we employed within this cycle was Test Driven Development (TDD).

Test Driven Development is a methodology where tests are written first before writing the code itself. It basically creates a checklist of what the code should do, and then the code is written to meet those requirements. However, in our project, we didn't start from scratch with our tests. We were handed a pre-made set of tests along with the structure of the parser we were building. We were responsible for creating most of the logic for parsing the expressions following the recursive descent design. This setup meant that we weren't strictly doing "test first" development as traditional TDD may suggest. We couldn't dictate the tests as they were already set for us. This might seem like it takes away some of the freedom and creativity from the development process, but I enjoyed this structure. With the tests provided upfront, we had a clear understanding of what our code needed to accomplish. It provided a solid foundation and helped us stay focused on meeting the expected outcomes.

I found that debugging is much more straightforward when using TDD because failing tests pinpoint exactly where the code isn't working as expected. When a test fails, it serves as a clear indicator that there's a problem with the code. This allowed us to focus our debugging efforts on the specific area of the code that needed attention, rather than searching through the entire code base for the issue. Since we didn’t create the whole codebase, this would be a tedious task. By fixing all the failing tests, we are confident that we have addressed the issues in the code. Also, by writing additional tests to be used on our partners codebase we were able to find remaining bugs in the code that may have been missed by the provided test cases. This iterative process of writing tests, watching them fail, and then fixing the code until the tests pass, helps streamline the debugging process and leads to more robust and reliable software. Using TDD helped my team create a fully functional parser for the CatScript programming language.