Montana State University Computer Science
Senior Team Portfolio

Compilers CSCI 468
Spring 2024

Colton Parks

Nicholas Weigand

Section 1: Program.

Source.zip for my Catscript compiler is located in the same directory as this pdf.

Section 2: Teamwork.

I, Colton Parks, was responsible for the implementation of code used in the tokenizer, parser and bytecode in Catscript, my partner, Nicholas Weigand, contributed cat script documentation and 3 additional tests. So, like all the other groups the workload was about 95% on me in Catscript and 5% on the partner for writing the tests and documentation. You will find the documentation in Section 4 technical writing and the tests here. This PartnerTests test file is located under demo under test under src.

```
public class PartnerTests extends CatscriptTestBase {
    @Test
    public void forStatementWithNestedVarParses() {
        ForStatement expr = parseStatement("for(i in [10, 20, 30]){
print(i) var x : int = 10 }");
        assertNotNull(expr);
        assertEquals("i", expr.getVariableName());
        assertTrue(expr.getExpression() instanceof
ListLiteralExpression);
        assertEquals(2, expr.getBody().size());
        LinkedList<Statement> list = (LinkedList<Statement>)
expr.getBody();
        assertTrue(list.get(0) instanceof PrintStatement);
        assertTrue(list.get(1) instanceof VariableStatement);
        assertEquals(CatscriptType.INT,((PrintStatement)
list.get(0)).getExpression().getType());
    }
    @Test
    void nestedIfTest(){
        assertEquals("1\n", compile("if(true){ if(true){ print(1) }
}"));
    }
    @Test
    public void whitespaceHandlingTokenizerAndEmptyListArray(){
        assertTokensAre("1\t+\t1", INTEGER, PLUS, INTEGER, EOF); //
Tabs
        assertTokensAre("1 +   1", INTEGER, PLUS, INTEGER, EOF); //
Multiple spaces
        assertTokensAre("1\n+\n1", INTEGER, PLUS, INTEGER, EOF); //
Newline characters
        assertEquals("[]\n", compile("print([])"));
    }
}
```

Section 3: Design pattern.

A pattern we use is memoization which is found in catScriptType.java and the code can be found below. We do not want to code directly here because we do not want to run a specific input more than once. To store the specific input results, we can store each result in a static hash map. Before we run an input through this method, we can use the hash map to check if we already have the results of that specific input, if we do already have the results then we don't need to run it again we just return those results. If we coded this directly, every time we run the method on the same input, we could be running it after we already have ran it, which would be redundant.

```
static HashMap<CatscriptType, ListType> cache = new HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
    //check if the result is already in the cache
    if (cache.containsKey(type)) {
        return cache.get(type);
    }
    ListType listType = new ListType(type);

    // Put the result into the cache
    cache.put(type, listType);
    return listType;
}
```

Section 4: Technical writing

(Catscript markdown file containing documentation)

Catscript Documentation

This document outlines all the features in Catscript

Introduction

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

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

Features

We have split the features, first the expressions and then the statements

Expressions

Additive Expressions

```
var result = 5 + 3
print(result)  // Output: 8
```

Represents addition or subtraction operations between two integers or concatenates if they are not both integers.

Boolean Literal Expressions

```
var is_true = true
print(is_true)  // Output: true
```

Represents boolean literals, which can have two possible values: true or false. Boolean literals are used to represent logical values in Catscript.

Comparison Expressions

```
var result = 5 > 3
print(result)  // Output: true
```

A comparison expression compares two expressions using relational operators such as ">", ">=", "<", and "<=". It evaluates to true if the specified condition is true and false otherwise.

Equality Expressions

```
var result = 5 == 5
print(result)  // Output: true
```

An equality expression compares two expressions for equality using
the "==" or "!=" operators. It evaluates to true if the expressions are equal or false if they are not.

Factor Expressions

```
var result = 10 / 2
print(result)  // Output: 5
```

A factor expression performs multiplication or division operations between two expressions using the "*" or "/" operators. It evaluates to the product or quotient of the expressions.

Function Call Expressions

```
function add(a, b) {
    return a + b
}
var result = add(3, 5)
print(result)  // Output: 8
```

Represents multiplication or division operations between two expressions.

Identifier Expressions

```
var x = 10
print(x)  // Output: 10
```

Represents identifiers, which are names used to identify variables, functions, and other entities in Catscript.

Integer Literal Expressions

```
var num = 42
print(num)  // Output: 42
```

Represents integer literals in Catscript.

List Literal Expressions

```
var myList = [1, 2, 3]
print(myList)  // Output: [1, 2, 3]
```

A list literal expression creates a list containing one or more elements. It allows you to initialize lists with specific values, which can be accessed and manipulated in your Catscript code.

Null Literal Expressions

```
var myNull = null
print(myNull)  // Output: null
```

Represents the null literal value, indicating the absence of a value in Catscript.

Parenthesized Expressions

```
var result = (5 + 3) * 2
print(result)  // Output: 16
```

Represents expressions enclosed within parentheses, used to control the order of operations and clarify the structure of complex expressions.

String Literal Expressions

```
var str = "Hello, World!"
print(str)  // Output: Hello, World!
```

Represents string literals in Catscript.

Unary Expression

```
var result = -5
print(result)  // Output: -5
```

Represents unary operations like negation or logical negation on expressions.

Statements

For loops Statements

```
var myList = [1, 2, 3]
for (item in myList) {
    print(item)
}
// Output:
// 1
// 2
// 3
```

This statement initiates a loop that iterates over elements in a list or range. It allows you to execute a block of code repeatedly for each item in the specified collection or range.

Assignment Statements

```
var x = 10
x = x + 5
print(x)  // Output: 15
```

An assignment statement is used to assign a value to a variable. It allows you to modify the value stored in a variable by assigning a new value to it.

Function Call Statements

```
function greet(name) {
    print("Hello, " + name + "!")
}
greet("Alice")  // Output: Hello, Alice!
```

This statement invokes a function by its name, passing arguments as input parameters. It allows you to execute the code defined within the function's body and optionally return a value.

Function Definition Statements

```
function add(a, b) {
    return a + b
}
print(add(3, 5))  // Output: 8
```

This statement defines a new function in Catscript, specifying its name, parameters, return type (optional), and body. It allows you to encapsulate reusable blocks of code and execute them by calling the function.

If Statements

```
var x = 10
if (x > 5) {
    print("x is greater than 5")
} else {
    print("x is not greater than 5")
}
// Output: x is greater than 5
```

An if statement allows conditional execution of blocks of code based on the evaluation of a specified condition. If the condition evaluates to true, the code within the "if" block is executed, otherwise, if an "else" block is provided, the code within the "else" block is executed.

Return Statements

```
function add(a, b) {
    return a + b
}
print(add(3, 5))  // Output: 8
```

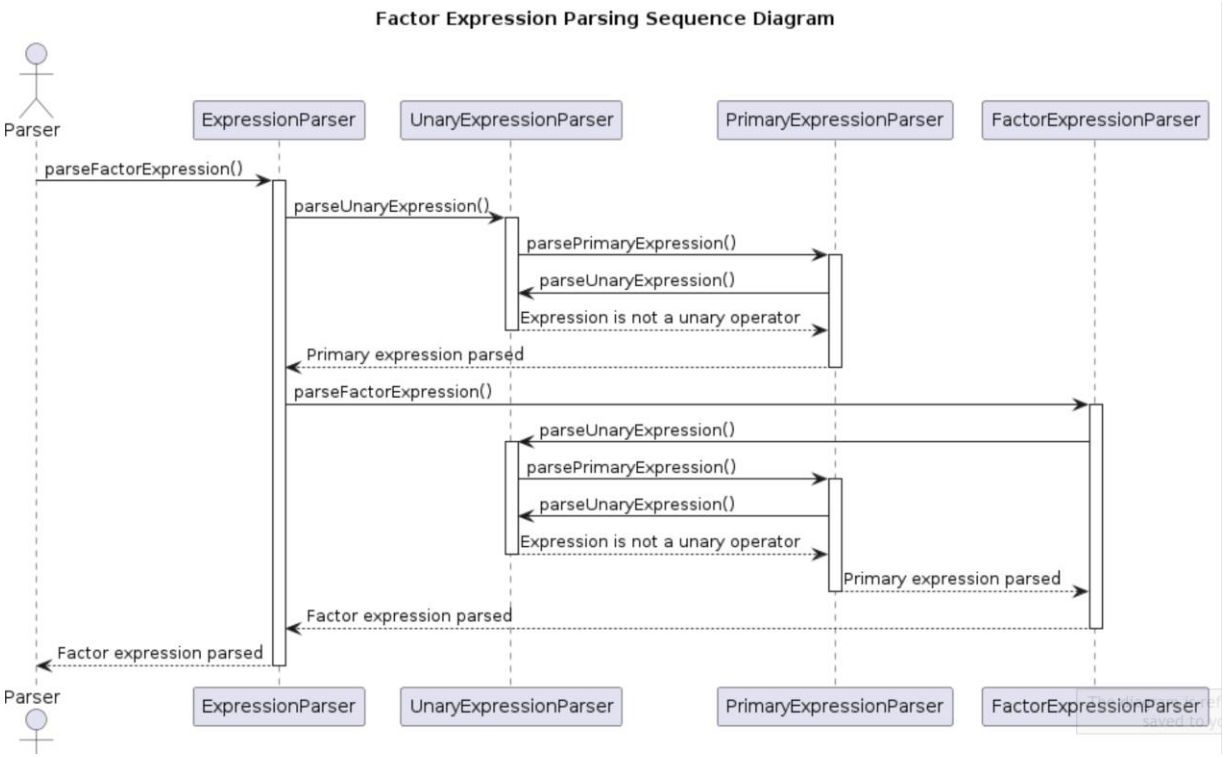
The return statement is used within a function's body to specify the value that should be returned when the function is called. It allows functions to produce output values that can be used in other parts of the program.

Variable Statements

```
var x = 10
print(x)  // Output: 10
```

This statement is used to declare variables and optionally assign initial values to them. It allows you to define variables with optional type annotations, specifying the data type of the variable's value.

Section 5: UML



The sequence diagram illustrates the parsing process for factor expressions within a larger parsing system. At the start, the Parser actor initiates the parsing process by invoking the ExpressionParser to handle the factor expression. The sequence unfolds as each parsing component, including UnaryExpressionParser, PrimaryExpressionParser, and FactorExpressionParser, plays its role in dissecting the expression. The recursive descent aspect becomes apparent as we observe the delegation of parsing tasks from higher-level parsing components to lower-level ones.

For instance, the ExpressionParser initially calls the UnaryExpressionParser, which subsequently calls the PrimaryExpressionParser. If the expression encountered is not a unary operator, the PrimaryExpressionParser communicates this back to the UnaryExpressionParser, reflecting a recursion point where the parsing hierarchy returns to a previous level to explore alternative parsing paths. This recursive descent pattern continues as each parsing component delegates tasks, delving deeper into the expression structure until terminal symbols are reached, or parsing errors are encountered. The delegation and subsequent return of control between parsing components exemplify the recursive nature of the parsing process, where parsing tasks are broken down hierarchically until the entire factor expression is successfully parsed.

Section 6: Design trade-offs

CatScript is designed to use recursive descent instead of using what is called a parser generator. Recursive descent is something that is very profound yet simple and easy to understand. It can be described as a top-down parser. A parser generator can have less written code than doing it by hand, and certainly less infrastructure. Parser generators automatically generate a parser based on a formal grammar specification. A parser generator can be more difficult to understand, they may be better for grammars that have complex structures or ambiguous rules. In a parser generator there is obscure syntax for things that are obvious compared to when you do it by hand. Parser generators can support multiple target languages, allowing developers to generate parsers in their language of choice. For students, a parser generator does not give a good feel of the recursive nature of grammars and parsing.

Section 7: Software development life cycle model

We used Test-Driven Development, which means that we were supplied with tests that we needed to pass before we wrote the code. There was a suite of test files, containing tests to pass in each part of making CatScript. There were test files for the tokenizer, the parser, and bytecode. If the tests are thorough enough, passing all of them generally means that your code does everything it is intended to do correctly. When tests are very specific like we had, it was very helpful. If you are supplied with tests that are very vague, non-descriptive, or confusing, that could be more of a hindrance than a help.

I liked this model of development because when I was not sure of what to do next or if thought there was something that I may have forgotten, the tests provided me with some direction. The tests had very specific names that described what they were testing for, so just by seeing the name of the test that is failing you could determine where you should look. Another way of finding your shortcomings was to see where the test failed and why the test failed. I would not mind using this method of development again in the future. Testing after implementation seems fine as well but I do like to have this kind of testing before implementation.