CSCI-468: Compilers

Spring 2024

Jacob Halsey, Kian Lynde

Section 1: Program

The source code for the program is attached to GitHub under /capstone/portfolio/source.zip.

Section 2: Teamwork

Team Member 1 wrote all the source code for all the base tests given. These tests were for the tokenizer, expression parsing, statement parsing, evaluation, and producing JVM byte code. Team Member 2 provided 3 extra tests for parsing to ensure that team member 1's compiler was parsing CatScript properly. Team member 2 also completed the documentation of the CatScript Language. The tests that team member 2 contributed to the program will be in a screenshot below and are also located in src/test/java/edu/montana/csci/csci468/parser/PartnerTest.java. Team member 1's time spent on the project is about ninety-five percent and Team member 2's time spent on the project is about five percent.

```
@Test
public void forStatementWithNestedIfVarPrintParses() {
    ForStatement expr = parseStatement("for(x in [1, 2, 3]){ if(x > 10){ print(x) } else { print( 10 ) } var y: int = 2 print(y) }");
    assertNotNull(expr);
    assertEquals("x", expr.getVariableName());
    assertTrue(expr.getExpression() instanceof ListLiteralExpression);
    assertEquals(3, expr.getBody().size());
    LinkedList<Statement> list = (LinkedList<Statement>) expr.getBody();
    assertTrue(list.get(0) instanceof IfStatement);
    assertTrue(list.get(1) instanceof VariableStatement);
    assertTrue(list.get(2) instanceof PrintStatement);
    assertEquals(CatscriptType.INT, ((PrintStatement) list.get(2)).getExpression().getType());
}
```

```
@Test
public void parseFunctionCallExpressionWithArray() {
    FunctionCallExpression expr = parseExpression("spoof([1,2,3])", false);
    assertFalse(expr.getArguments() instanceof IdentifierExpression);
    assertEquals("spoof",  expr.getName());
    assertEquals(1, expr.getArguments().size());
    Expression arg = expr.getArguments().get(0);
    assertTrue(arg instanceof ListLiteralExpression);
    assertEquals(3, ((ListLiteralExpression) arg).getValues().size());
    for (Expression value : ((ListLiteralExpression) arg).getValues()) {
        assertTrue(value instanceof IntegerLiteralExpression);
    }
}
```

```java
@Test
public void returnStatementExprWithNestedIfInFunction() {
    FunctionDefinitionStatement expr = parseStatement("function x(y : int) : int { if(y > 10){ print(y) } else { print(10) } return 22 }");
    assertNotNull(expr);
    assertEquals("x", expr.getName());
    assertEquals(CatscriptType.INT, expr.getParameterType(0));
    LinkedList<Statement> list = (LinkedList<Statement>) expr.getBody();
    assertEquals(2, list.size());
    assertTrue(list.get(0) instanceof IfStatement);
    ReturnStatement returnStmt = (ReturnStatement) list.get(1);
    assertNotNull(returnStmt);
    assertTrue(returnStmt.getExpression() instanceof IntegerLiteralExpression);
}
```

Section 3: Design Pattern

The implemented design pattern is memoization, utilized within the code located at /src/main/java/edu/montana/csci/csci468/parser/CatscriptType.java. Memoization optimizes the speed of determining the type of a list by employing a cache instead of directly coding the list type. This approach is beneficial because determining the type can sometimes be an expensive operation.

Memoization operates by storing the list type in a cache: if the type is not already cached, it is retrieved, stored, and then returned to the calling function. Subsequent calls for the same type retrieve it directly from the cache, rather than recomputing it. This caching mechanism significantly speeds up operations, particularly for frequently accessed types like integers, as they are readily available in the cache. Below is a screenshot of the implementation of memoization.

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

Section 4: Technical Writing

Catscript Guide

This document should be used to create a guide for catscript, to satisfy capstone requirement 4

Introduction

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

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

CatScript Types

CatScript is statically typed, with a small type system as follows

int - a 32 bit integer

string - a java-style string

bool - a boolean value

list - a list of value with the type 'x'

null - the null type

object - any type of value

Features

For loops

In Catscript, a for loop begins with the "for" keyword, signifying its nature as a loop construct. Following this keyword, the parser anticipates specific tokens such as opening and closing parentheses, as well as opening and closing brackets to encapsulate its components.

Within the parentheses, an identifier is mandatory, as it represents the variable that iterates through the loop. Following the identifier, the "in" keyword is expected, indicating that what follows is an iterable expression.

Inside the brackets, at least one statement is required. However, multiple statements can be included within the loop body, delineated by the opening and closing brackets.

A for loop is useful as it allows execution of a block of code repeatedly.

Below is an example demonstrating the syntax and structure of a for loop in Catscript:

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

If Statement

In Catscript, an if statement begins with the "if" keyword, signaling its conditional nature. Following this keyword, the parser anticipates specific tokens such as opening and closing parentheses, as well as opening and closing curly braces to encapsulate its components.

Within the parentheses, an expression is mandatory, as it represents the condition that the program evaluates. Following the expression, the opening curly brace is expected, indicating the beginning of the block of code to execute if the condition is true.

Inside the curly braces, at least one statement is required. However, multiple statements can be included within the if statement's body, delineated by the opening and closing curly braces.

Optionally, an if statement can be followed by an "else" clause, signifying an alternative block of code to execute if the condition is false. This "else" clause can contain another if statement or a block of code enclosed in curly braces.

The if statement is useful as it allows the execution of a block of code conditionally, depending on whether a specified condition evaluates to true or false.

Below is an example demonstrating the syntax and structure of an if statement in Catscript:

```
if(x < 10){
    print(x)
} else if (x > 10) {
    print(20)
} else {
    print(100)
}
```

Print Statement

In Catscript, a print statement begins with the "print" keyword, followed by opening and closing parentheses, encapsulating an expression. This expression represents the value to be printed.

The print statement is useful for displaying information to the user or logging output during program execution. It allows for dynamic output, as the expression inside the parentheses can be any valid Catscript expression, including variables, literals, or more complex expressions.

Below is an example demonstrating the syntax and structure of a print statement in Catscript:

```
print(10)
```

Variable Statement

In Catscript, a variable statement begins with the "var" keyword, followed by an identifier representing the variable name. Optionally, a colon and a type expression can be included to specify the variable's data type. The assignment operator "=" is then used to assign a value to the variable, which is represented by an expression.

Variable statements are useful for declaring and initializing variables in Catscript, allowing for storage and manipulation of data within a program.

Below is an example demonstrating the syntax and structure of a variable statement in Catscript:

```
var x: int = 20
```

Assignment Statement

In Catscript, an assignment statement involves an identifier (representing a variable name) followed by the assignment operator "=" and an expression. This expression assigns a value to the variable identified by the identifier.

Assignment statements are fundamental in programming as they allow for the modification of variable values, updating them based on the result of an expression.

Below is an example demonstrating the syntax and structure of an assignment statement in Catscript:

```
y = x / 2 + 5
```

Function Declaration

In Catscript, a parameter list is a comma-separated list of parameters enclosed within square brackets. Each parameter consists of an identifier representing the parameter name, optionally followed by a colon and a type expression to specify the parameter's data type.

Here's the updated description for the function declaration, including the parameter list:

In Catscript, a function declaration begins with the "function" keyword, followed by an identifier representing the function name. Then, an opening parenthesis is used to denote the start of the parameter list, which consists of zero or more parameters separated by commas. Each parameter can include an identifier for the parameter name and optionally a colon followed by a type expression to specify the parameter's data type. After the parameters, a closing parenthesis is used to signify the end of the parameter list.

Optionally, a colon followed by a type expression can be included to specify the return type of the function.

The function declaration is then enclosed in curly braces, with the function body containing one or more function body statements. These statements can either be a regular statement or a return statement, which is used to return a value from the function.

Function declarations are crucial in Catscript for defining reusable blocks of code that can be called and executed within a program.

Below is an example demonstrating the syntax and structure of a function declaration in Catscript:

```
function add(x : int, y : int){
    return x + y
}
```

Function Call

In Catscript, a function call invokes a function by specifying its identifier followed by a list of arguments enclosed in parentheses. The arguments are passed to the function as inputs, and they can be expressions, variables, or literals.

Function calls are essential for executing reusable blocks of code defined by functions. They allow you to pass data to functions and receive results or perform actions based on the function's behavior.

Below is an example demonstrating the syntax and structure of a function call in Catscript:

```
add(1, 2)
```

Type Expression

In Catscript, a type expression specifies the data type of a variable, parameter, or function return type. It can be one of the basic data types such as "int", "string", "bool", "object", or "list". Additionally, for generic types like "list", a type expression can include angle brackets "<" and ">" to specify the type of elements contained within the list.

Below is an example demonstrating the syntax and structure of a type expression in Catscript:

```
list <int>
```

Return Statement

In Catscript, a return statement signifies the end of a function's execution and optionally provides a value to be returned. This value can be of any valid Catscript expression type.

Return statements are essential for functions as they allow the function to produce output or results that can be used elsewhere in the program.

Below is an example demonstrating the syntax and structure of a return statement in Catscript:

```
return y
```

Equality Expression

In Catscript, an equality expression compares two values for equality or inequality. It consists of one or more comparison expressions separated by the "!=" (not equal) or "==" (equal) operators.

Equality expressions are used to check if two values are equal or not, and they evaluate to a boolean value (true or false) based on the comparison result.

Below is an example demonstrating the syntax and structure of an equality expression in Catscript:

```
x == y
```

Comparison Expression

In Catscript, a comparison expression evaluates whether one value is greater than, greater than or equal to, less than, or less than or equal to another value. It consists of one or more additive expressions separated by comparison operators such as ">", ">=", "<", or "<=".

By evaluating to a boolean value (true or false) based on the result of the comparison, comparison expressions enable you to implement logic that depends on the relationships between different values.

Below is an example demonstrating the syntax and structure of a comparison expression in Catscript:

```
x > y
```

Additive Expression

In Catscript, an additive expression combines multiple values using addition or subtraction operators. It consists of one or more factor expressions separated by the "+" (addition) or "-" (subtraction) operators.

Additive expressions are fundamental for performing arithmetic operations in your code. They allow you to add or subtract values, variables, or expressions, facilitating numerical calculations and manipulations.

By evaluating to a numerical value based on the result of the addition or subtraction operations, additive expressions enable you to perform arithmetic operations and obtain the results for further processing or display.

Below is an example demonstrating the syntax and structure of an additive expression in Catscript:

```
1 + 1
```

Factor Expression

In Catscript, a factor expression combines multiple values using multiplication or division operators. It consists of one or more unary expressions separated by the "*" (multiplication) or "/" (division) operators.

Factor expressions are essential for performing arithmetic operations involving multiplication or division in your code. They allow you to multiply or divide values, variables, or expressions, facilitating numerical calculations and manipulations.

Below is an example demonstrating the syntax and structure of a factor expression in Catscript:

```
x / y
```

Unary Expression

In Catscript, a unary expression performs operations on a single operand. It can consist of a unary operator such as "not" or "-" applied to another unary expression, or it can be a primary expression on its own.

Unary expressions are crucial for performing unary operations in your code, such as negation or logical negation. They allow you to modify the value or state of a single operand based on the specified unary operator.

Below is an example demonstrating the syntax and structure of a unary expression in Catscript:

```
not true
```

Primary Expression

In Catscript, a primary expression represents the most basic units of data or operations. It can consist of various components, including identifiers, string literals, integer literals, boolean literals ("true" or "false"), null literals ("null"), list literals, function calls, or expressions enclosed in parentheses.

Primary expressions are fundamental for representing individual values or operations in your code. They allow you to work with data at its simplest level, whether it's a variable, a constant value, or the result of a function call or expression evaluation.

Below are some examples demonstrating the syntax and structure of a primary expression in Catscript:

```
true
"string"
1
```

List Literal

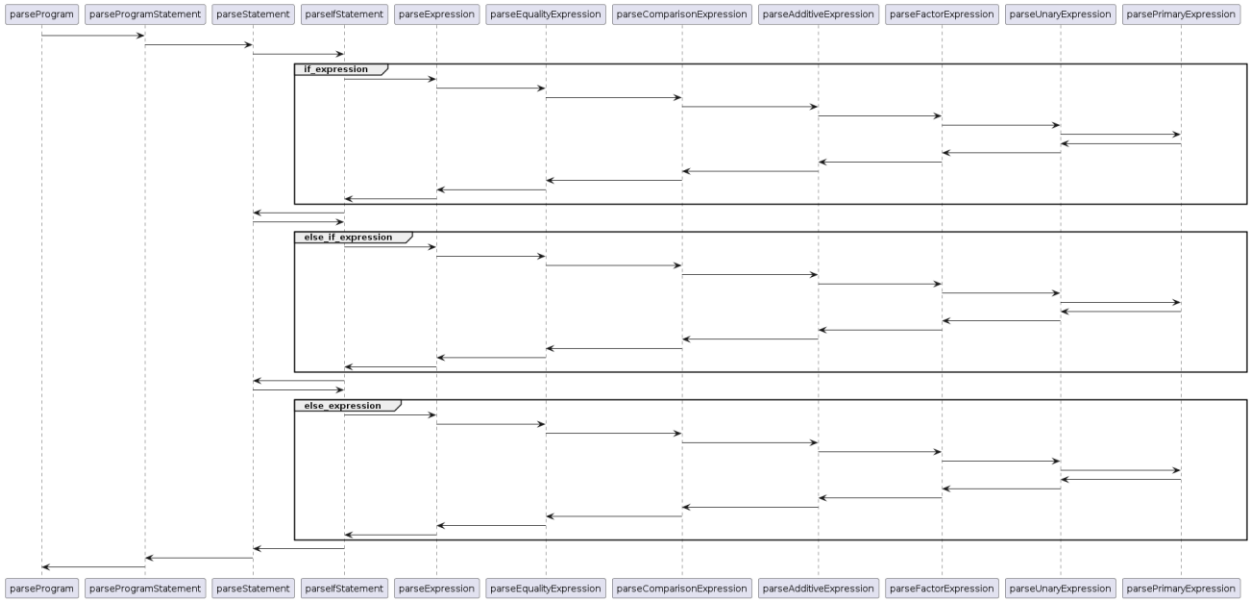
In Catscript, a list literal represents a collection of values enclosed within square brackets. Each value is separated by a comma, and the entire collection is enclosed within square brackets.

List literals are crucial for representing lists or arrays of values in your code. They allow you to create collections of data that can be indexed or iterated over, enabling various operations such as adding, removing, or accessing elements within the list.

Below is an example demonstrating the syntax and structure of a list literal in Catscript:

```
[1, 2, 3, 4]
```

Section 5: UML



The UML diagram above is a sequence diagram of parsing an if statement, that has an else if statement, and an else statement. The diagram begins when the Catscript program encounters an if statement. The program matches the keyword if and goes in to parsing the condition of the if block. This will move to parseExpression. This part will parse any type of expression in the grammar using recursive descent to get to the lowest level. Each expression will have parts that recurse down to primary expression. Once the expression is determined, it gets returned to the if conditional block. This process is repeated for the else if conditional block and the else block. The statements that are executed in each block are not included in this diagram, but the process will be almost identical. After the conditional expressions are parsed, parseStatement would be used to get all the statements in each block. This was omitted as it made the diagram illegible, and parsing the statements in the body will all call parse expression as that is how the parser was written. Other calls were also removed as they would have returned null and be skipped over completely.

Section 6: Design Tradeoffs

For Catscript, the main design used is recursive descent in the parser. Another option is a parser generator. Recursive descent was used instead because there is greater control over the parsing than a generator. It also makes it easier if a new feature is added to

Catscript. Since the tokenizer was built by hand instead, making a new token or token for the new feature being added is simple. If a new feature is a while loop, the token while is added. Then follow the grammar and add a new parsing method for the while loop, which makes the parsing simple with new features being added. Another reason recursive decent was used is that the code is more legible than a parser generator's output. With parser generators, the input is usually lexical grammar and regular expressions that comprise the tokenizer. The resulting code from the tokenizer is then run in a generator that produces the parser. Adding new features using a generator is more complicated as the tokenizer will have to be changed to incorporate the new feature. With lexical grammar and regular expressions being more complex to read, debugging can be more challenging than the tokenizer and recursive descent used for Catscript. Overall, using recursive descent is better than parser generation as Catscript is a simple language, and the ease of use of recursion follows nicely with how the grammar flows.

Section 7: Software Development Lifecycle

The model used to develop and test the compiler for Catscript was Test Driven Development. This model was also used with Test First Development as all the tests were given before starting the program. These 2 models together gave the project a defined set of requirements for getting each part of the compiler working properly. The main parts were the tokenizer, parser, evaluation, and producing JVM byte code that all had their separate tests to be completed. Since the compiler was built on previous parts, the test was useful for catching previous mistakes made in either the tokenizer or parser when doing tests further in the project. An example of this is producing JVM byte code for a list with no explicit type. This test failed because of a parsing error for the list type, and the test was able to help find this error saving a significant amount of time. However, there are some downsides of using this type of model for software development. One downside is it is not possible to test everything for a given software. This forces a decision on making enough tests that the software works most of the time, but not making too many tests that it becomes too expensive. Finding the balance between these two sides can be very challenging. Most of the tests given were a good balance, but there are some instances that there weren't any tests making it difficult to continue. One example is in the byte code section of the project. A lot of the expressions did not have specific tests to make them work, but other tests required them to work before the test could be passed. The tests would also not explicitly say which expressions need to be implemented, increasing the time it took to complete this section. Overall, I think test driven development is good as it is

simple and can thoroughly check a program is working as intended at the same time. The only part I would change for the compiler is to add extra tests to help in the further sections of the project.