Compiler Implementation

CSCI 468: Compilers

Carlee Joppa (main contributor, team member 1), Emily Ingbertsen (team member 2)

Section 1: Program

The zip file of my final repository is in the directory called `/capstone/portfolio/source.zip` . This program is an implementation of a compiler for the programming language Catscript.

Section 2: Teamwork

For this project, team member 1 implemented the compiler, which includes tokenization, parsing, evaluation, and compilation. Team member 2 contributed the technical documentation (Section 4) and wrote additional tests (shown below). These sections are integral to broader understanding of the programming language of Catscript and its compiler. Team member 1 spent around 95% of the time on this project and team member 2 spent around 5%.

```
@Test
void AssignmentStatementWorksProperly() {
assertEquals("5\n", compile("var x = 1\n" +"x = x + 4\n"+ "print(x)"));
assertEquals("10\n", compile("var x = 1\n" +"x = -3194 \n"+"x = 10"+ "print(x)"));
assertEquals("Hello!\n",compile("var x = \"Goodbye!\"\n "+"x = \"Hello!\"\n"+"print(x)"));
}

@Test
void IteratingOverListLiteralWorks(){
    assertEquals("1\n2\n3\n",compile("var x : list<int> = [1, 2, 3]\n"+"for( i in x ) {\n" +
    "  print(i)\n" + "}\n"));
    assertEquals("true\nnull\nfalse\n",compile("var x : list<object> = [true, null, false]\n"
    +"for( i in x ) {\n" +
    "  print(i)\n" + "}\n"));
    assertEquals("Harry\nJoe\nBob\n",compile("var x : list<string> = [\"Harry\",\"Joe\",\"Bob\"]\n"
    +"for(i in x){ print(i) }"));
}

@Test
void GlobalsAccessibleInsideFunctionsAndStatements(){
    assertEquals("36\n", compile("var x = 6\n"+
            "function foo() : int {\n" +
            "  x = x * 6\n" +
            "  return x\n" +
            "}\n" +
            "print( foo() )\n"));

    assertEquals("Hello World!\n",compile("var x = \"Goodbye World!\" \n" +
        "if(true){ x = \"Hello World!\" \n print(x) }\n"));
}
```

Section 3: Design pattern

One design pattern used in my implementation of the Catscript compiler is the memoization pattern. The implementation of this pattern can be seen below and is located in the file `CatscriptType.java` in the source code. This pattern is primarily employed for optimization of programs, especially in cases of expensive operations.

Therefore, I used this pattern in order to optimize the creation of list types. Instead of creating a new list type everytime this function is called, which may be used be frequently, I used memoization. Using memoization allows me to store the result of an expensive operation (in this case creating new a list type) and use it when needed. When that type of list is first used, the new list type will be created and stored in a HashMap. The next time that list type is needed, the HashMap will be consulted and that list type will be found and returned, which has an average time complexity of O(1).

```
static HashMap<CatscriptType, ListType> cache = new HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
    cache.computeIfAbsent(type, k ->  new ListType(type));
    return cache.get(type);
}
```

Section 4: Technical writing

Introduction

CatScript is a simple scripting language that is statically typed, has familiar syntax, and has basic programming capabilities. It is important to note that CatScript does not support classes and is only a functional programming language.

Here is an example of some CatScript:

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

Features

CatScript consists of a series of simple "features" that allow for easy comprehension of the language.

Literal Types

The following literal types are the available types that can be used in the CatScript language.

String Literals

String literals are a data type that is represented using text. String literals can be created by wrapping a set of quotation marks around any data.

Here are some examples of strings in CatScript:

```
"Hello World!"
"My number is (123) 456-7890"
```

Boolean Literals

Boolean literals are a data type that represent the boolean values of true and false. These literals can be used as conditional arguments for some statements like the if statement.

Here are examples of boolean values:

```
true
false
if(true){ print(5)}
```

Integer Literals

Integer Literals are the numerical data type in CatScript. Integers can only represent whole numbers and not decimals.

Here are examples of integer values:

```
5
2141975162
-214
```

List Literals

CatScript supports List literals that can hold similar types of objects. List literals can be defined with a type or without a type. If no type is specified, then the type of the list will be inferred. List literals can additionally be used to iterate over in for-loops.

Here are some examples of how to declare lists in CatScript:

```
var myList = [1 , 2 , 8, -32]
var myOtherList : string = ["Jessica", "James", "Peter"]
```

Null Literals

The null literal is used to indicate something of no value. The null literal is assignable to all type literals.

Here are some examples of the null literal:

```
null
var x : string = null
var y = null
```

Object Type

Object literals is used as an overarching type in CatScript. As such, all types are assignable to a variable with the Object type.

Here are some examples of the Object type:

```
var x : object = "Hello World!"
var y : object = -298
```

Expressions

Additive Expressions

An additive expression handles the operations of addition and subtraction between two other expressions. This can only be used between integer and string variable types, however Strings can only be used with the addition operator for concatenation. Between integer types, the result of an additive expression is also an integer. However, if at least one side of the expression is of type String, then the result will also be of type String.

Here are some examples of an additive expression in CatScript with integers:

```
var a = 1 + 7
var b = a - 10
```

Addition in CatScript can also be used to concatenate strings:

```
var c = "Hello " + "World!"
var d = 10 + " is a fun number!"
```

Comparison Expressions

Comparison expressions in CatScript handle the operations of >, <, <= , and >= to compare two expressions. This can only be done between two integer values. The result of a comparison expression is boolean type.

Here are some examples of comparison expressions in CatScript:

```
1 < 10
20 + 5 >= 35
```

Equality Expressions

Equality expressions in CatScript use the operators == and != to determine if the two sides of the expression are equal (or not equal) to each other. The result of this expression returns a boolean type.

Here are some examples of equality expressions in CatScript:

```
10 == 9 + 1
"Hello" != "Hi"
```

Factor Expressions

Factor expressions use the operators of * and / to multiply and divide two expressions. This can only be done between two integer values.

Here are some examples of factor expressions in Catscript:

```
var a = 81 / 3
var b = a * 5
```

Identifier Expressions

Identifier expressions are singular identifiers that represent some data that has been assigned to it previously. As such, identifier expressions can be used in the place of the data that it represents.

Here are some examples of identifier expressions in CatScript:

```
var a = true
var b = 16
if(a){print(b)}
return b
```

Unary Expressions

The unary operators in CatScript are - for negative integer values and not to negate boolean expressions.

Here are some examples of unary expressions in CatScript

```
-3
not false
not (-3 < 0)
```

Statements

Variable Statements

In CatScript, in order to store a value to some identifier we must use a variable statement. Variable statements can be defined with or without a specified type. This type can be specified by including a colon and type after the variable's name. If a type is not provided, the type will be inferred.

Here are some examples of variable statements in CatScript:

```
var name : string = "Susan"
var x = "Hello, my name is: " + name
```

Assignment Statements

Assignment statements are used to reassign values to previously declared variables. Assignment statements will also type check to ensure the user is not illegally assigning values to type incompatible variables.

Here is an example of an assignment statement in CatScript:

```
var x = 21481
x = 64 + 123492
```

For Loops

For loops in CatScript use a list literal and some iterating identifier to iterate over and repeat the code within the loop until done iterating.

In CatScript, with each iteration through the loop, the iterating identifier will take the value of the data in the iterated list at the location of the current iterating factor. In the case of the example below, on the first iteration through the for-loop, x will take the value of 1. Then the next iteration, it will take the value of 2, and finally 3 on its last iteration.

Here is an example of a For loop in CatScript:

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

If Statements

If statements in CatScript take in some boolean condition and only execute the code within them if that condition results to true.

Additionally, If statements can also have else statements attached to them such that if their condition is not true, then the code within the else statement will be executed.

Here is an example of an If statement in CatScript:

```
if(x >= 22){
print("woohoo!")
}
else{
print("oh no!")
}
```

Print Statements

Print statements in CatScript take in some expression and prints it to console. Print statements can print simple variables or print the results of expressions.

Here are some examples of print statements in CatScript:

```
print(x < y)
print("Hello World!")
print(x + 27)
```

Return Statements

Return statements can be used to return from a function. Return statements can be used with or without a return value depending on the function return type.

Return coverage must be ensured for all if statement branches in order to properly compile.

Here is an example of return statements in CatScript:

```
return
return x + 19
return "The end!"
return x >= 214
```

Functions

Functions in CatScript can be used to execute some code upon the user calling the function. Functions can take in some number of parameters that can then be used as local variables to the function. Parameters can be given specified types similar to variable statements, otherwise if not specified they are objects.

Functions have return types and can return values upon being called. If a function's return type is not specified, the return type will be void.

Function Definitions

Here is an example of defining a function in CatScript:

```
function add (num1 : int, num2 : int) : int {
    return num1 + num2
}
```

Function Calling

Here is an example of calling the above function in CatScript:

```
var sum = add(14, 36)
```

Section 5: UML

Figure 1 is a sequence diagram for the statement `print("Result: " + (4 + 7))`. The figure is called `CapstoneSequence.png` and can also be found in the same folder as `source.zip` and this file.

The sequence diagram below shows the recursive nature of this recursive descent parser. As seen in Figure 1, we start with parseProgram and start recursing down, eventually reach parsePrintStatement where the program will recognize that this statement is a print statement (black arrows). Then the expression inside the print statement will be parsed. The program will recurse downward towards simpler statements until it reaches parsePrimaryExpression where `Result:` will be recognized as a StringLiteralExpression (in purple arrows). The StringLiteralExpression will be returned up the chain until reaching AdditiveExpression where the `+` operator will be found (purple arrows). This operator will instigate the search for the right side of the expression, recursing down to parsePrimaryExpression again (shown in blue arrows).

Shown in green arrows, the search for the right hand side will result in the discovery of a ParenthesizedExpression `(4 + 7)`, which will call parseExpression to parse the inside of the expression. The program will again recurse down the expression parsing, finding an IntegerLiteralExpression (`4`), and similar to earlier will return the IntegerLiteralExpression until reaching AdditiveExpression where the `+` operator is found. Then the right side of the AdditiveExpression must be found, so the program will recurse down until reaching parsePrimaryExpression and will find the IntegerLiteralExpression `7`. The AdditiveExpression `4 + 7` is then returned up until escaping the ParenthesizedExpression.

The ParenthesizedExpression will be returned up through the program until AdditiveExpression where `"Result: " + (4 + 7)` will be put together as an AdditiveExpression (shown in blue arrows). The AdditiveExpression will be returned until parsePrintStatement (shown in purple arrows), where it will be fully parsed and returned all the way up as a PrintStatement (black arrows).

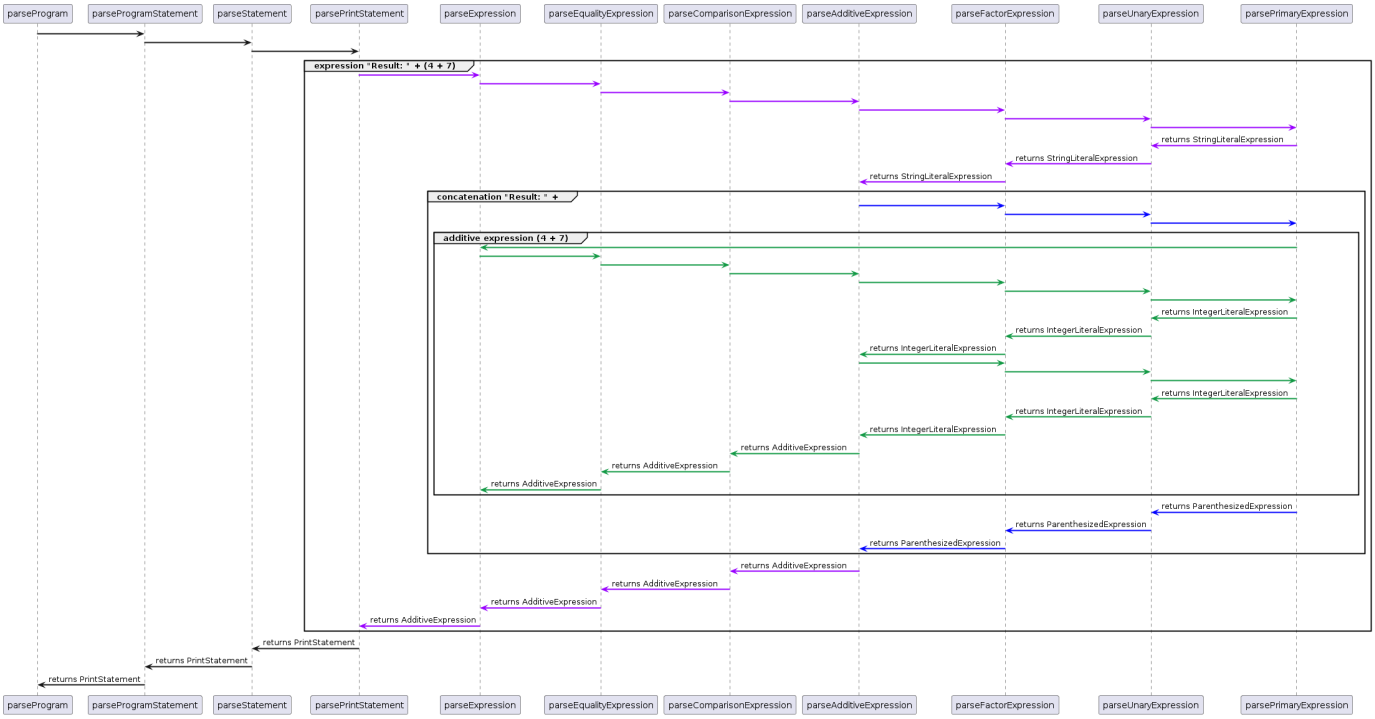

Figure 1. Sequence diagram for print("Result: " + (4 + 7))

Section 6: Design trade-offs

The major design trade-off decision in this project is the choice to make a recursive descent parser instead of using a parser generator. A recursive descent parser is a top-down parser, so it starts with the biggest picture, with each element calling the next smallest element down through the grammar (which can be seen in Figure 1) until the current token is matched with one of these elements. In calling the next 'smallest' token, the non-terminals with the least precedence are called first, and the non-terminals with the most precedence are called later on. This type of descent through the different levels creates a parse tree that has the correct precedence, and deals with ambiguity in the grammar. In implementing a recursive descent parser, the level of control over parsing and the grammar is high.

Parser generators by contrast are usually third party programs that generate a parser given a grammar. This approach has several advantages, including decreased time for writing the parser itself. Parser generators will also exactly implement the grammar specified. Exactness based on a grammar can be great, but there may be times that implementation of a grammar may have unintended side effects and the parser needs some flexibility. This need for flexibility arose in the implementation of my Catscript parser. If the Catscript grammar was followed exactly, return statements would have produced an error if they were inside an if statement. Due to the hands-off nature of parser generators, this side effect may not have been noticed and eliminated the ability to return different values based on conditionals. Since the code for parsing is output of a program instead of done by hand, the level of control and understanding on the part of the developer is greatly reduced.

Section 7: Software development life cycle model

For this project, I used Test Driven Development(TDD). This means that comprehensive tests were written before I truly started development, and that the program developed was essentially a response to the tests written. The majority of these tests were written by the instructor of this course, and some of them were written by my partner (team member 2).

This model was overall extremely helpful to me, as plans for development were already laid out, making it easier to understand exactly which sections were completed and which were unfinished. Instead of rummaging around through files to find functions that were uncompleted, the tests previously written allowed me to focus in on specific functions and fix them quicker. TDD also allowed me to ensure my compiler worked for more obscure cases that could have easily been forgotten without this model, like the null case.

While the TDD model was great in a lot of ways for this project, there are some aspects about this model that were a hindrance. This model is built on the idea that all code is covered by tests, which is a great ideal, however in the real world it is too easy to forget to cover a small function. This may lead to undeveloped holes in a project, that without close attention, may fall through the cracks. This was a problem I encountered in this project, as I realized quite late that AssignmentStatement was lacking proper test coverage for compilation. The other hindering aspect of TDD I found is that code may pass the tests, but it may not be fully 'correct' and cause problems in other areas.