CSCI 468 - Compilers Portfolio

Spring 2024

Treyton Grossman
Colby Roberts

Section #1: Program

Source code is in the attached zip folder

Section #2: Teamwork

Team member two implemented the parser, which was approximately 95% of the project. Team member one contributed documentation of the parser, as well as three additional tests to certify completion. Team member two contributed about 5% of the overall project.

The first of the three tests contributed by team member one, ensured that the project could parse a for loop with multiple lines of code to execute. This made sure the parser could accurately complete the logic of a For Statement. This test passed. The second test made sure that print statements processing an unterminated list would throw an unterminated list error. This test passed. The final contributed test looked at return statements, and checked if they functioned as intended when given an equality statement. This test also passed.

Section #3: Design Pattern

In parser/CatscriptType.java, I used a memoization design pattern. The code is as follows:

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

Memoization is the process of ensuring a method does not execute with the same inputs multiple times, by saving the already computed results. This is achieved in this case by using a static hashmap. In this case, the computeIfAbsent method checks if an existing value is at the key “type” in the cache Hashmap. If one exists, this means the computation has already taken place, so we can just use the saved instance. If an object does not exist at key “type” that means the calculation has not already taken place, so we make a new ListType instance, with type being passed in as t. We then save the instance in the cache corresponding with type, so we do not make the calculation again in the future, before returning the generated ListType.

I implemented this memoization to save memory and run time. This segment of code will be much faster as when a duplicate calculation is asked for, it only needs to execute a lookup and not resave a duplicate ListType to memory.

Section #4: Technical writing

The following documentation was written by team member one.

Introduction

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

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

Features - Expressions

Additive Expressions

Additive expressions in Catscript use either the "+" (add) or "-" (subtract) operator. If the "+" operator is used, Catscript will compute the sum of both sides of the expression only if both sides are integers. If both sides are not integers, Catscript will concatenate the string value of the two sides. If the "-" operator is used, Catscript will compute the difference of the two sides in the order: right - left.

```
2 + 4
7 - 3
"hello " + "world"
```

Boolean Literal Expression

Boolean literal expressions in Catscript are used to denote binary values "true" and "false". These values are commonly used in decision logic, and are returned by equality expressions and comparison expressions.

```
true
false
```

Comparison Expression

Comparison expressions in Catscript use either the "<" (less than), ">" (greater than), "<=" (less than or equal to), or ">=" (greater than or equal to) operator. Using any of the comparison expression operators will attempt to cast both sides of the expression to an integer, then evaluate to a boolean value depending on if the comparison is true or false.

```
5 < 3
9 > 1
4 <= 4
20 >= 21
```

Equality Expression

Equality expressions in Catscript use either the "==" (equals) or "!=" (does not equal) operator. The Catscript equality expression operators can be used regardless of type, and will evaluate to a boolean value depending on if the equality is true or false.

```
6 == 2
2 != 6
```

Expression

Expressions in Catscript are strings of symbols that may be evaluated to determine its value. Expressions are a parent class to more specific expressions, such as AdditiveExpression and ComparisonExpression.

```
4 + 4
false
"hello world"
```

Factor Expression

Factor expressions in Catscript use either the "*" (multiply) or "/" (divide) operator. Using either of the Catscript factor expression operators will attempt to cast both sides of the expression to integers, and compute the product or quotient respectively.

```
3 * 3
9 / 3
```

Function Call Expression

Function call expressions in Catscript are denoted by an identifier followed by a list of arguments enclosed in parentheses. Function call expressions are used to invoke functions with passed parameters if required.

```
foo()
bar(5, "hello", false)
```

Identifier Expression

Identifier expressions in Catscript are a sequence of characters (or just one) used to identify things in memory. Identifier expressions are things like variable names and function names.

x
foo

Integer Literal Expression

Integer literal expressions in Catscript are one or more numerical symbols used to represent quantitative values. Integer literal expressions in Catscript, much like other languages do not contain extra precision past the ones digit, and can only represent whole values.

0
283

List Literal Expressions

List literal expressions in Catscript are one or more comma-separated values enclosed in square brackets. Values in the list may include types such as bool, string, int, or object.

[1, 2, 3]
["hello", false, null]

Null Literal Expressions

Null literal expressions in Catscript are used to denote a null or "nothing" value. It is used with the keyword "null".

null

Parenthesized Expressions

Parenthesized expressions in Catscript are expressions enclosed in parentheses. They are used to group expressions together and prioritize evaluation.

(1 + 3)
5 - (1 + 3)

String Literal Expression

String literal expressions in Catscript are strings of symbols enclosed in double quotations. These expressions are most often used as human-readable outputs and messages.

"hello world"
"456 Catamount Street"

Syntax Error Expression

Syntax error expressions in Catscript are messages given to the user when unexpected inputs are detected during parsing expressions. It will output an error message describing the illegal input.

Error: Unexpected Token

Type Literal

Type literals in Catscript represent the various types available in the Catscript scripting language. Types such as 'bool' and 'int' are common type literals.

bool
string

Unary Expression

Unary expressions in Catscript use either the "-" or "!" operator. Adding the "-" operator to an integer will multiply the value by -1. Adding the "!" operator will attempt to cast the following expression to a Boolean value, then invert it.

-20
!(6 == 3)

Features - Statements

Assignment Statements

Assignment statements in Catscript use the "=" operator. Using this operator will save the evaluated expression on the right of the "=" operator to a variable whose name is declared on the left side.

```
x = 4 + 5
y = true
```

For Statement

For statements in Catscript begin with the "for" keyword, and are followed by an identifier, the "in" keyword, and an expression, all enclosed in parentheses. The for statement will execute all statements enclosed in braces n number of times depending on the expression. Current iteration is stored in the identifier preceding the expression.

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

Function Call Statement

Function call statements in Catscript begin with an identifier denoting the name of the function to invoke. Function call statements store function call expressions, which are then evaluated as function calls.

```
foo()
bar(5, "hello", false)
```

Function Definition Statement

Function definition statements in Catscript begins with the function keyword and stores all information about a function including name, arguments, argument types, return type, and body. Function definition statements contain the necessary information to execute and return values when the function is invoked.

```
function foo (x, y, z) { print(x) }
```

If Statement

If statements are conditional logic statements that start with the "if" keyword. Statements inside the body of the if statement will be executed if the parenthesized expression returns true.

```
if (3 == 5) { print("hello world") }
```

Print Statement

Print statements in Catscript begin with the "print" keyword. Print statements show the output of an expression contained within them to the user.

```
print("hello world")
```

Return Statement

Return statements in Catscript begin with the "return" keyword. Return statements are used in functions and allow functions to finish execution while also providing an output value.

```
return(5)
```

Statement

Statements in Catscript are instructions that express some action to be carried out. For statements and if statements are different types of statements.

```
for(x in [1, 2, 3]){ print(x) }
if (3 == 5) { print("hello world") }
```

Syntax Error Statement

Syntax error expressions in Catscript are messages given to the user when unexpected inputs are detected during parsing expressions. It will output an error message describing the illegal input.

```
Error: Unexpected Token
```

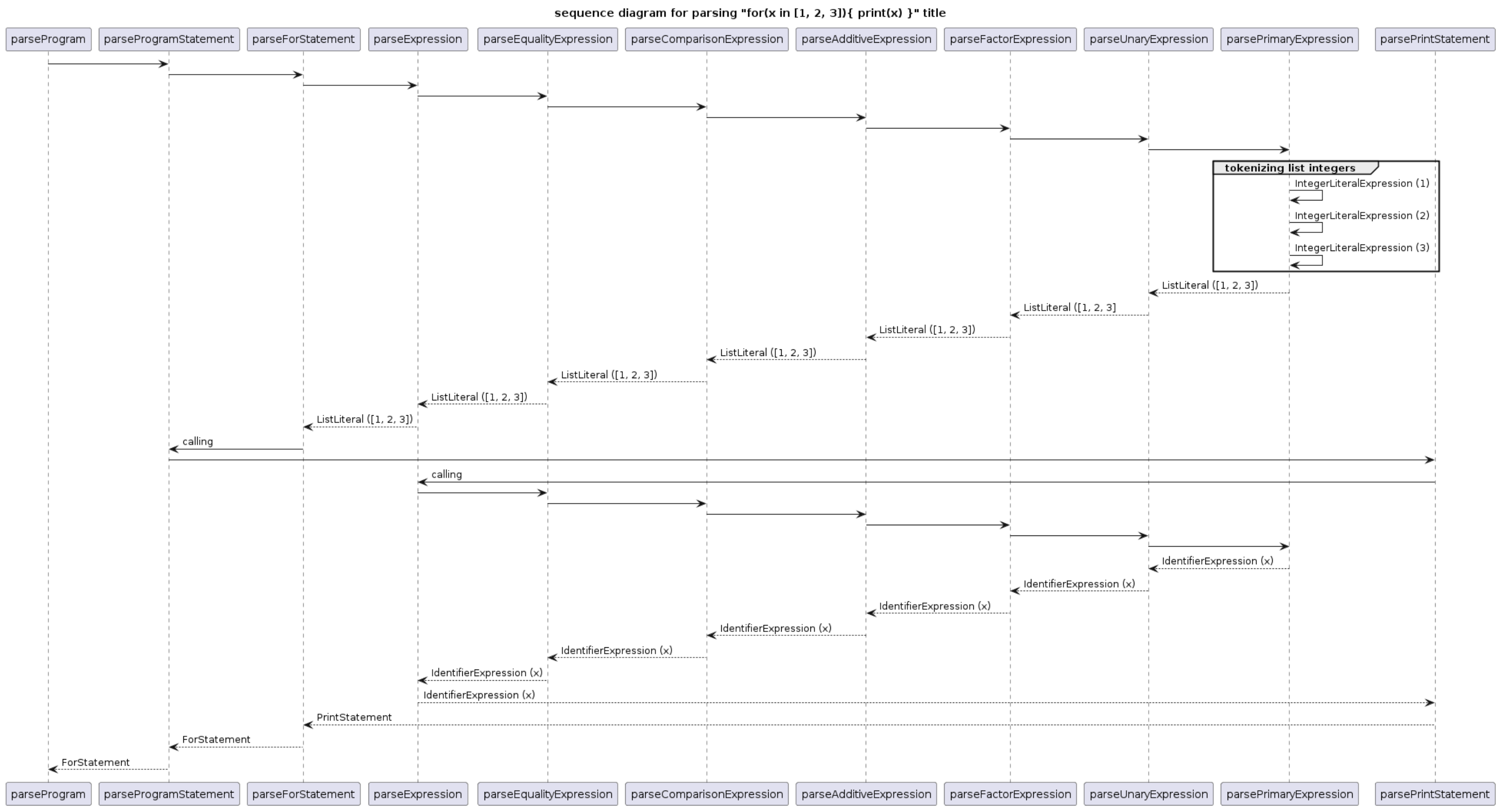
Variable Statement

Variable statements in Catscript begin with the "var" keyword. Variable statements initialize variables with a given name, type, and expression which is stored in memory.

```
var x = 5
```

Section #5: UML

The following Sequence diagram is associated with parsing “for(x in [1, 2, 3]) { print(x) }”



This sequence diagram illustrates the implemented Recursive Descent Algorithm. The idea is that each statement and expression might have elements that can be broken down into more basic expressions or statements. This works in the example given because in the sought after For Statement, we have an iterator that contains both an identifier and a List Literal Expression, and a body Print Statement that also contains an identifier. In the sequence diagram, you can see the hierarchy of the different features, and how parsing them in order can allow us to recursively descend towards our expected result.

For the sake of clarity, the sequence has any data being returned as a dotted line, and any action which calls another method as a solid line. Just going backwards doesn’t necessarily mean there was a return called, and we are breaking out of an execution.

For a broad summary of the sequence diagram, we begin by parsing out our Identifier Expression in parseForStatement(), before moving down all the way until we can put the given listed iterator values into instances of IntegerLiteralExpressions. At that point, we then throw them all into an instance of ListLiteralExpression, which we can return all the way to the method parsing the For Statement. At this point we need to parse the body of the loop, so we call parsePrintStatement(). This method will parse the parameter expression by moving all the way down to parsePrimaryStatement(). This method will sort the parameter as an IdentifierExpression, then send it all the way back to parsePrintStatement(). The parsePrintStatement() will then return an instance of PrintStatement to parseForStatement(). At this point the instance of ForStatement is complete, so we return that all the way back to finish the call hierarchy.

Section #6: Design Trade-offs

The compiler I helped to develop used a Recursive Descent design. This allowed for an incredibly simple, readable, and maintainable pattern. The lack of generated code made it very easy to understand and interact with. Recursive Descent also made the compiler design very relevant to the overarching grammar. When you look at the language grammar, you can very easily see what the process will be like to parse and compile a given statement or expression. Another key point to consider when looking at Recursive descent, is that it allowed for refraction of large segments of code. This makes it quite easy to understand what each piece of code does, and where to find the segments that you are looking for,

On the other hand, I know that there are tradeoffs to not using a parser generator. Most parser generators are made to be relatively easy to use. Most of the actual parser is generated for you, so it is less work when it comes to writing the compiler. Parser generators are also often optimized for performance, and can generate efficient parsing code without much input. Also parser generators are good in that they have good error handling. This isn't exclusive to parser generators but the error handling is guaranteed to be consistent. Unfortunately thanks to all code being generated and not written, the overarching design is very hard to understand at first glance.

Overall, I am happy with the compiler being designed using a Recursive Descent algorithm. The lack of generated code allowed for easy comprehension of the infrastructure. Also, the organization of the design is a massive point that makes it hard to consider using a parser generator instead.

Section #7: Software Development Life Cycle Model

The software development model I used was a Test-Driven Development(TDD) life cycle. This means I was given a number of tests to get passing in order to demonstrate key features have been implemented. Overall, I quite liked this development model. While it made the project quite hard to approach at first, the simple goal oriented learning process made it fun in a way. I knew exactly what I was aiming for with each edit I made, and after interacting with the project for a while, I knew how to get there. Also, being able to use the debugger to navigate through the project allowed for a more structured introduction into its infrastructure.

On the other hand, it is hard to say that I fully understand how to make a compiler when my real goal for the course was to just get all of the tests passing. While I did learn a lot about what a compiler is and how it functions, I am not sure if I could fully replicate the efficiency of the current project. I could definitely make a somewhat functional compiler now, but not one I would feel content submitting to a boss or customer. This may be a feeling that everyone has, but it's hard to say a quasi-Test First Development model didn't contribute to it.

In conclusion I thought TDD was a great choice for this particular project. It meant that I could rely on the debugger to help me understand my goals, and gave me the chance to practice coming into a project that is already in the process of being developed. It is not often that you will get the chance to make a software completely from scratch in a professional setting so I see this as good practice.