CatScript

CSCI 468 Compilers

Spring 2024

Joshua Bowen, Racquel Bowen

Professor Carson Gross

Section 1: Program

The source code for the project is in a zipped folder included with this document.

Section 2: Teamwork

Team member 1: responsible for the implementation of the compiler and the successful completion of all tests. Also responsible for writing most of the capstone document. Estimated time spent: 95%

Team member 2: responsible for creating 3 high-level tests to ensure the correct functionality of the compiler. Also responsible for providing the technical writing portion of the capstone document, which was the documentation of the Catscript programming language. Estimated time spent: 5%

Section 3: Design pattern

In the Catscript type system (in `CatscriptType.java`), I implemented a memoization pattern in the `getListType()` method that caches list types after calculating them to avoid repeatedly recalculating the same list type over and over. My implementation is highlighted in yellow below. I also included a non-highlighted version for more comfortable viewing. For clarification, `existingListTypes` is a static map from component types to list types in the `CatscriptType` class.

```
public static CatscriptType getListType(CatscriptType type) {
        if (!existingListTypes.containsKey(type)) {
                existingListTypes.put(type, new ListType(type));
        }
        return existingListTypes.get(type);
}
```

```
public static CatscriptType getListType(CatscriptType type) {
    if (!existingListTypes.containsKey(type)) {
        existingListTypes.put(type, new ListType(type));
    }
    return existingListTypes.get(type);
}
```

There are two main reasons that I chose to memoize the `getListType()` method.

First, for most applications, programmers will use only a handful of list types in a given program. For example, a programmer working with processing a list of strings will likely create several lists of strings in their code, but they will not likely create a triple-nested list of integers. Because programmers will likely create only a few lists with similar types, there is a performance improvement in storing previously created list types in case the same list type shows up again later in compilation. In the same line of thought, very complex list types like creating a quadruply nested list of objects rely on simpler list types like a list of objects simply by their nature. So, caching simpler list types can aid in the construction of more complex list types and vice versa.

Second, strictly speaking, in a programming language there should only be a single instance of any type expression. If two variables are lists of integers, they should have the **same exact** type, not copies or similar

instances of a type. The memoization of the list type construction guarantees that each list type is stored after creation, ensuring that there is only one list of integer type and only one list of objects type. I could have achieved the same functionality with the singleton pattern, but memoization had another benefit that I previously described.

Section 4: Technical writing.

Introduction

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

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

Expressions

Catscript is a language with statements and expressions. Since most statements have internal expressions, we should look at Catscript expressions first.

The evaluation and precedence of Catscript expressions closely match those in the Java programming language.

To ensure expression precedence works correctly in Catscript, all of the expressions are broken into different categories. We will examine them from lowest precedence, to highest.

Equality Expression

The equality expression operators are `!=` (for not equal) and `==` (for equal).

They are both binary operators, and evaluate to truth/boolean values, and work on any other expressions except equality expressions (unless they are parenthesized). For future reference, the expressions that an expression can operate on will be ignored because they will always be the expressions of higher precedence.

```
19 == 19     // True
7 != “7”     // True

null == “foo”  // False
true == false  // False
```

Comparison Expression

The comparison expression operators are `>` (for greater than), `>=` (for greater than or equal), `<` (for less than), and `<=` (for less than or equal).

```
1 > 1   // False
1 >= 2  // False

1 < 2   // True
1 <= 1  // True
```

Additive Expression

The additive expression operators are `+` (for addition) and `-` (for subtraction). It is worth noting that we can concatenate strings with the `+` operator as well.

```
1 + 1   // 2

13 - 4  // 9

“Cat” + “script”   // “Catscript”
```

Factor Expression

The factor expression operators are `*` (for multiplication) and `/` (for division).

```
5 * 3   // 15

12 / 2  // 6
```

Unary Expression

The unary expression operators are `-` (for mathematical negation) and `not` (for boolean negation).

```
not true  // false
not false // true
not not true // true

- 13  // negative 13
```

Primary Expression

Primary expressions are the bits and pieces that make up all of the other expressions in Catscript. We have been dealing with them in all our examples.

The list of primary expressions is:

- strings
- integers

- booleans (true and false)
- null
- lists of expressions
- parenthesized expressions
- identifiers (names of variables or functions)
- function calls (can be either a statement or expression)

```
// strings
“foo”
“bar”
“hello world”

// integers
21
55
0

// booleans
true
false

// null
null

// lists
[1, 2, 3]
[“foo”, null, 13]

// parenthesized expressions
(1 - 2 + (4 * 4))
(not true)
(“I” + “ like ” + “to” + “ code!”)
```

We will look at identifiers and function calls later on in the statements section.

Statements

Every Catscript program has zero or more statements and zero or more function definitions. Let's look at the statements available in Catscript and then examine function definitions.

Print Statements

Catscript's built-in `print` function can print any expression in Catscript. Examples are shown below.

```
print(86)

print("hello world")

print([3, 2, 1])
```

```
print(not true)
```

The `print` statement can also print variables (there is an example shown below). The `print` statement is used a lot to show the functionality of other features in Catscript.

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

If you are confused on what `var x = "hello"` means read on.

Variable Declaration

To declare variables in catscript use the `var` keyword. Along with declaration, you can also assign any expression to your new variable after declaring it. Catscript will infer the type of the variable for you! More on this later.

```
var w   // declaring w
var x   // declaring x

var y = 3       // declaring and assigning y
var z = "foo"   // declaring and assigning z
```

After variables are declared they can be assigned. Read on to see how.

Variable Assignment

Like most programming languages, Catscript uses the `=` operator for assigning values to variables. We already saw this when we declared and assigned variables all in one step. But we can assign variables at any point after they are declared.

```
var x
x = 54
x = 7
x = -5

var y = 32
y = 8
```

Some assignments will cause errors when the types of the variable and value do not match the same types.

```
var myString = "hi"
myString = [1, 4, 7]       // causes an error
```

We can always define our variables to have broader types if we want. See the documentation on types in Catscript.

If-Statements

Conditional control flow is present in Catscript as if-statements.

```
if (true) {
    print("This text will ALWAYS print")
}

if (false) {
    print("This text will NEVER print")
}
```

Catscript also has support for if-else-statements. You can make as many if-else branches and can end with an else clause.

```
if (1 >= 4) {
    print("IF body")
} else {
    print("ELSE body")
}

if (1 > 3) {
    print("first IF")
} else if (2 > 3) {
    print("second IF")
} else if (3 > 3) {
    print("third IF")
} else {
    print("optional ELSE body")
}
```

For more complex programs, you can nest if statements within the bodies of other if statements.

For-Loops

Catscript also provides support for clean and simple for-loops similar to Python. It does not support the older C-style for-loops, which means that a list expression will always be needed to "power" the loop.

```
for num in [11, 30, 44, 87] {
    print(num)
}

var myList = ["Catscript", "can", "be", "fun!"]
for str in myList {
```

```
    print(str)
}
```

For-loops are not limited to hard-coded list expressions. List variables can be iterated over, allowing for more flexibility.

```
var myNums = [-1, 95, 81]
for num in myNums {
    print(num)
}
```

Additionally, nested lists can be traversed naturally.

```
var nestedList = [[1, 2], [3, 4], [5, 6]]
for innerList in nestedList {
    for num in innerList {
        print(num)
    }
}
```

Function Definitions

To create functions in Catscript use the `function` keyword followed by the function name and a list of parameters to take in.

```
function myFunction(x, y) {
    print(x + y)
}

function multiplication(x, y) {
    print(x * y)
}
```

Catscript Functions are very flexible. Functions do not have to take parameters and they can return an expression if needed.

```
function fruitfulFunction() {
    if (3 != 4) {
        print("They are NOT equal")
        return false
    } else {
        print("They ARE equal")
        return true
```

```
    }
}
```

Catscript supports recursion (fruitfully or not).

```
function recursiveFunction(num) {
    if (num <= 0) {
        print(num)
        return      // no expression provided (so its not fruitful)
    } else {
        recursiveFunction(num - 1)
    }
}
```

Unfortunately, Catscript does *not* support inner-functions (closures) yet.

```
function outerFunction() {

    function illegalInnerFunction() {           // causes an error
        print("I am not syntactically valid")
    }

}
```

In Catscript functions are typed by their return type or lack thereof (more on this later).

Now that we can define functions, let's see how to call them. We already got introduced with the recursive function example!

Functions Calls

If we have a function `myFunc` defined somewhere in our program, we can call that function by just using its name and whichever parameters it requires.

```
myFunc() // calling the already defined function with no arguments
```

If a function requires two arguments, they can be listed between the parenthesis in the function call.

```
twoArgFunc(4, "foo")
```

Catscript allows functions to be called before they are defined!

```
myFunc()

function myFunc() {
    print("This could be handy!")
    print("This could also be useful!")
}
```

CatScript Types

CatScript is statically typed (meaning types cannot change), 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

Catscript will infer the type of variables and functions if they are not specified. When in doubt, it will infer the general type of object for variables and void for functions.

```
var myInt = 3 //myInt will be inferred as an int

var myString = "goodbye" //myString will be inferred as a string

var myList = [78, "water"] //myList will be inferred as list<object>

function printer(x) { //printer will be inferred as void
    print(x)
}
```

If you want to specify types you do so with a colon followed by a type expression.

```
var x : object = 24 //x will be typed as an object

var myString : string = "summer2024" //myString will be typed as a string

var myList : list<list> = [[10, 20], [1, 3]] //myList will be inferred as a
list<list<int>>

function identity(x) : object { //identity will be typed as object
    return x
}
```

You might wonder why we would want to specify types. Other than being specific, sometimes specifying general types allows your code to be more flexible.

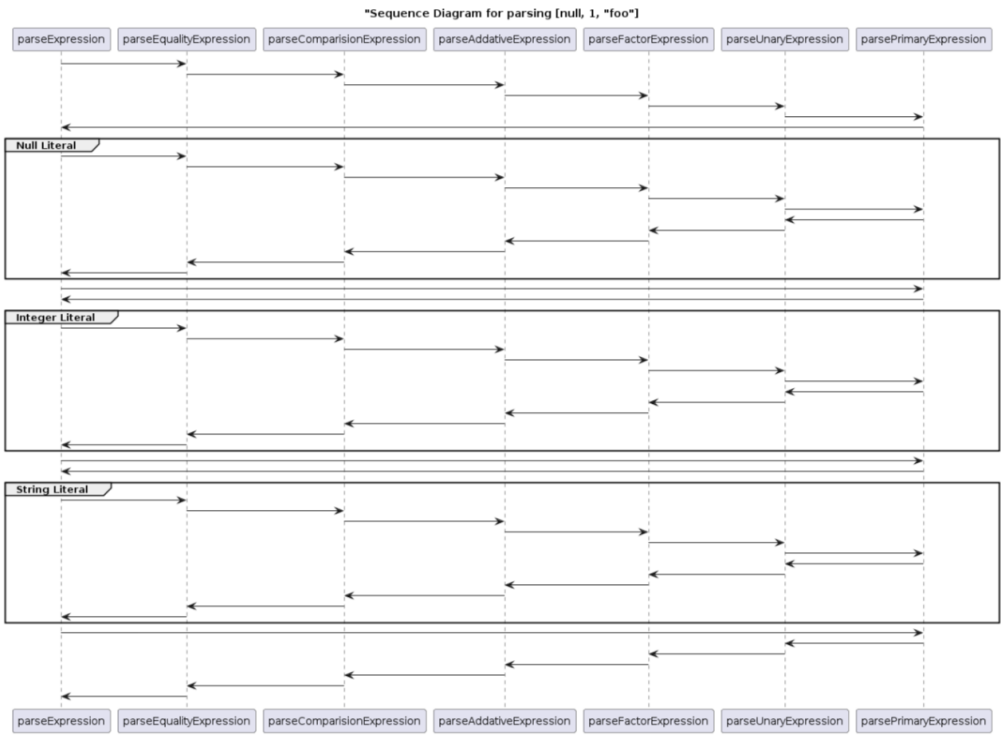
```
var x : object = 7

x = "foo"                      // valid since "foo" is still an object

x = [1, 2]                     // valid since [1, 2] is still an object

x = [[77, 88], "firstname"]    // valid since [[77, 88], "firstname"] is still
considered an object
```

We hope you enjoy using Catscript!

Go test it out!

Section 5: UML.

The majority of the design decisions were made before I was able to work on the project. My work focused on implementing the recursive descent algorithm. So, instead of showing UML about the system structure, I will show a sequence diagram of how the list literal expression `[null, 1, "foo"]` parses using recursive descent.



The three boxes in the figure show the process of parsing the literal expressions `null`, `1`, and `"foo"`. In each of the boxes, it is clear that the each parser function passes of to another parser function of higher precedence until finally the `parsePrimaryExpresion()` function is call where `null`, `1`, and `"foo"` can each be parsed and passed back up the chain of parser functions to the top-level `parseExpression()` function call.

Outside of the parsing of each literal expression, we see that there is a similar pattern for parsing the list literal. The parsing starts at `parseExpression()` and works its way down to `parsePrimaryExpression()`. The only difference is that within parsing the list, `parseExpression()` is called to parse each list element. Finally, at the end, the list literal finishes parsing and is passed up the chain of parser function calls to `parseExpression()` where it can then be returned for future use.

Section 6: Design trade-offs

Although I did not have input to the main design decisions in the project, I will still discuss the key design decision of the entire project: choosing to implement a recursive descent parser instead of using a parser generator.

In my opinion, the benefits of implementing the recursive descent parser by hand far outweigh the benefits of learning to use a parser generator. Similarly, the disadvantages of implementing the recursive descent parser are more bearable than the disadvantages of using a parser generator. The four advantages of implementing a recursive descent parser are that I have a better understanding of the recursive nature of grammars, I became more familiar with programming in Java, the focus of my time and energy was spent on understanding parsing, which is a good focus for a compilers class, and the algorithm can be used in other parsing contexts (like parsing file formats or regular expressions). On the other hand, the only benefit of using a parser generator is that I would have had to write less code overall. Clearly, implementing a recursive descent parser is more beneficial than learning to use a parser generator. In the same way, implementing a parser also has fewer disadvantages than using a parser generator. The only disadvantage of implementing a recursive descent parser by hand is that it is more time consuming. Unfortunately, I was not able to work on transpiling during the semester because all of my time was spent implementing the parser. However, using a parser generator can potentially be more time consuming because the generated parse code is extremely difficult to understand. Also, learning a single parser generator tool would not be an easily transferable skill in other contexts since they are specifically meant for parsing languages.

Section 7: Software development life cycle model

Our development life cycle model was Test-Driven Development (TDD) with the tests first.

In general, TDD was a beneficial development model because it made the task of programming a large project granular enough to tackle in bite-sized pieces. Furthermore, our grade in the class was largely determined by how many tests we were able to pass at a given checkpoint, which made the project expectations and timelines clear and manageable. The main disadvantage of using TDD was that our implementation was only as good as the tests required them to be. For example, if the `+` operator is defined to work on integers and strings in a language's grammar, but there are only tests that ensure the `+` operator works correctly for strings, there is nothing ensuring that the string functionality will be implemented. Several times in the semester, I would pass a checkpoint with all tests passing and later realize that I did not fully implement something in a certain expression or statement subclass. The solution to this problem is to continuously add tests to ensure all functionality is working correctly. My partner and I have created three tests that test functionality that was never tested, and hopefully our tests will be added to the tests for the project next year, so that future students will be guided to implement all functionality correctly.