CatScript

CSCI 468 Compilers

Spring 2024

Joshua Bowen, Racquel Bowen

Professor Carson Gross

Section 1: Program

The zipped folder `source.zip` accompanying this document contains all the source code of the compiler project.

Section 2: Teamwork

Team Member 1 (95%): implemented the tokenizer, recusive descent parser, evaluation, and compilation to Java bytecode. Also created most of this capstone document.

Team Member 2 (5%): documented the Catscript programming language and wrote three tests for team member 1's implementation.

Section 3: Design pattern

I implemented the memoization design pattern in the `getListType()` of the `CatscriptType.java` class. A screenshot of the code is shown below. For reference, `HshMapListType` is a hash map of list component types to list types.

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

I chose to use the memoization pattern in the `getListType()` method because of two reasons. First, there is only one instance of the types integer, string and object in Catscript, so it would be inconsistent to generate new instances of list types. Instead memoization creates a single list type and then returns it when needed. Second, memoization can slightly increase the performance of the compiler because it will cache smaller list types that can aid in the creation of larger, more complex list types. For example, if the type list of integers is already cached, then creating list of list of integers is one step faster.

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).

As can be expected, they are both binary operators 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.

```
1 == 1    // True
1 != 2     // 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
“Cat” + “script” // “Catscript”

13 - 4  // 9
```

Factor Expression

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

```
3*4  // 12

12 / 2  // 6
```

Unary Expression

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

```
not true  // false

- 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”

// integers
1
13
0

// booleans
true
false

// null
```

```
null

// lists
[1, 2, 3]
[“foo”, null, 13]

// parenthesized expressions
(1 + 3 - 4)
(not true)
(“I” + “ love ” + “Catscript”)
```

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

Features

At the core, every Catscript program has zero or more statements and zero or more function definitions. Let's start by looking 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. Consider the examples below.

```
print(1)

print("foo")

print([1, 2, 3])

print(not true)
```

The `print` function can also print variables as shown before. The `print` statement is used a lot to demonstrate the functionality of other features in Catscript.

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

If you are confused on what `var x = "foo"` 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 langauges, Catscript uses the `=` opperator for assigning values to variables. We already saw this in action 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 = 3
x = 43
x = -5

var y = 32
y = -4
```

Some assignments will cause errors when the types of the variable and value do not play well together.

```
var myString = "foo"
myString = [1, 2]       // this will 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 will always print")
}

if (false) {
    print("this will never print")
}
```

Catscript also has support for if-else-statements. You can make as many if-else branches and optionally 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 clause")
}
```

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 [1, 3, 4, 8] {
    print(num)
}

var myList = ["Catscript", "is", "awesome!"]
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 = [12, 9, 4, 8]
for num in myNums {
    print(num)
}
```

Additionally nested lists can be traversed naturally.

```
var nestedList = [[1, 5], [2, 7], [3, 9]]
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)
}
```

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) {
        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() {           // will 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 a sneak peak in 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(1, "foo")
```

Catscript allows functions to be called before they defined!

```
myFunc()

function myFunc() {
        print("Wow, that is a nice feature!")
        print("I am looking at you C")
    }
```

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

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 infered as an int

var myString = "foo" //myString will be infered as a string

var myList = [1, "foo"] //myList will be infered as list<object>

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

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

```
var x : object = 3 //x will be typed as an object

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

var myList : list<list> = [[2, 3], [3, 7]] //myList will be infered as a
list<list<int>>

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

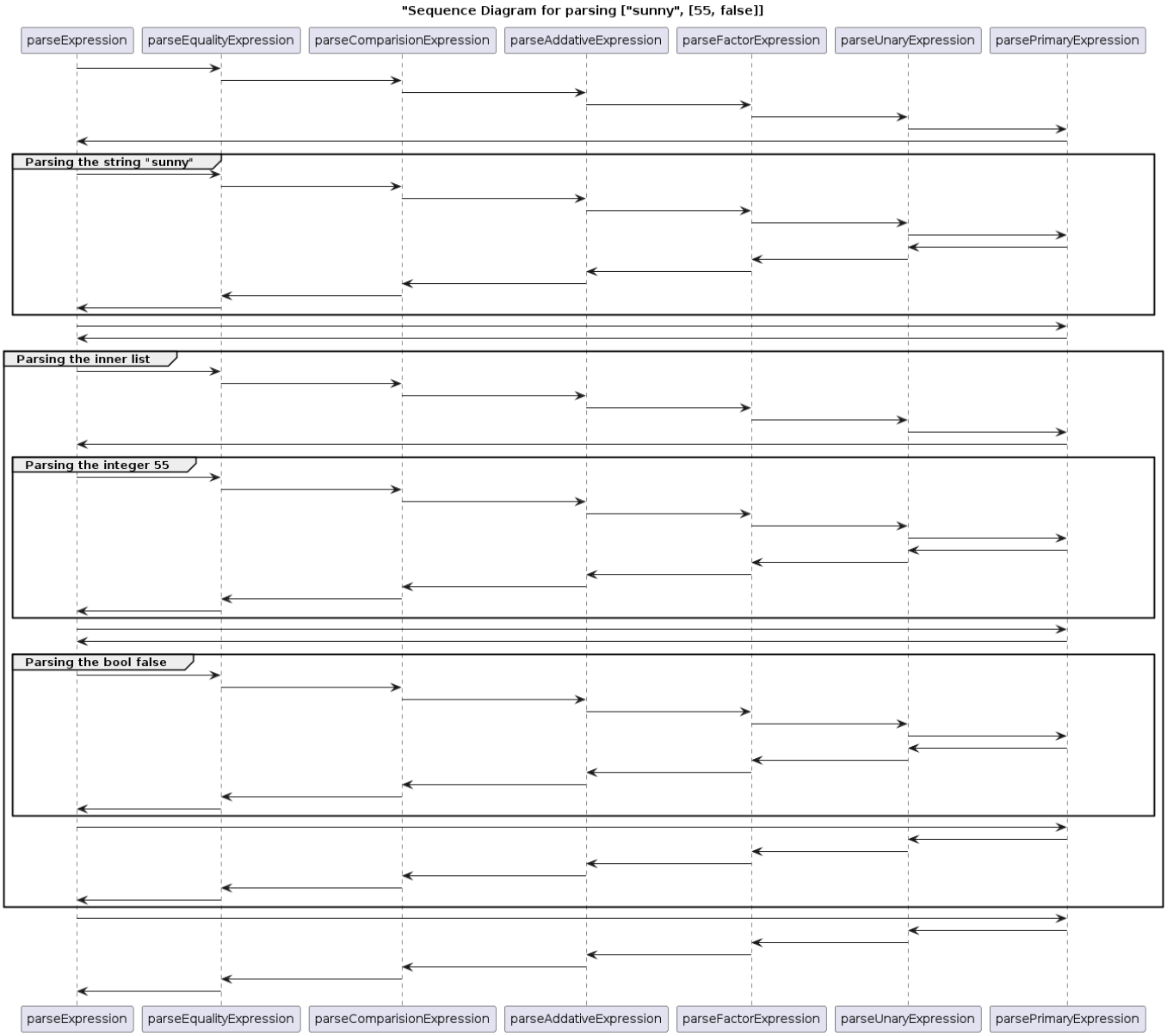
You might wonder why we would want to specify types. Other than being specific, sometimes specifying general types allows your code to be more flexible.

```
var x : object = 3

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

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

We hope you enjoy using Catscript!

Section 5: UML.

Many of the high-level design choices were made before I began working on the project. However, I implemented almost all of the recursive descent parsing algorithm. So, instead of commenting on UML related to the structure of the compiler, I will discuss how `["sunny", [55, false]]` is parsed by my recursive descent algorithm. The sequence diagram of how it is parsed is shown below.



The algorithm begins parsing the outer list with a call to `parseExpression()`. That call passes off to `parseEqualityExpression()` and it keeps getting passed down until `parsePrimaryExpression()` is called. Within that call, `parseExpression()` is called twice (since there are two inner expressions -- the string `"sunny"` and the list `[55, false]`). Parsing the string is very similar to parsing the outer list except once `parsePrimaryExpression()` finishes it passes the parsed string literal expression all the way back up the chain of calls to `parseExpression()`. When parsing the inner list, the same process is followed as the outer list. Once the string and inner list are parsed, the outer list is passed back up the chain of parser calls to `parseExpression()` and the algorithm finishes. From this diagram it is very clear how recursive the nature of expressions are.

Section 6: Design trade-offs

The most significant decision in the design of the project was the choice of implementing recusive descent parsing by hand instead of using a parser generator. There are several reasons why implementing a parser by hand is better. First, coding the parser myself, taught me a lot more about how parsing actually works. If I had simply written code to generate a parser, I would be an abstraction level away from knowing how parsing

really works. Second, implementing the parser by hand gave me more exprience coding in Java, which is more applicable to my career than learning the language of a parser generator. The only disadvantage to implementing the parser by hand was that it took more time. Although the argument could be made that since parser generated code is so difficult to debug, then parser generators are truly the more time-consuming of the two options.

Section 7: Software development life cycle model

I followed a Test Driven Development (TDD) life cycle during the entire project. The two advantages to this development model are that they broke up the implementation of the project into small chunks and clarified what I should be focusing on.

Creating a compiler seemed like a large and difficult project at the start of this semester. However, when I actaully got started coding, I was able to focus on working on one small part of the parser at a time. For example, instead of trying to code all of the recursive descent algorithm in one go and then test my work, I started with parsing literals, and then worked my way up, function call by function call, to parsing expressions. At each step, I only had to focus on passing the next test. Without the tests to guide me, I could have easily gotten lost in the scope of this project.

The disadvantage to TDD is that its easy to rely too much on the tests. A few times in the project, I only implemented enough of the compiler to pass the next set of tests. A few weeks later, I had to go back and finish my implementation because the tests had not covered all of the functionality that was needed in the next phase of the project. To help combat this issue, I tried to write solid tests for my partner.