CSCI468 Compilers Capstone Portfolio

Spring Semester 2024

Jack Hayward
Garrett Mullings

Section 1: Program

Here is a link to my github repository containing my source code that I wrote for my csci468 “Catscript” language compiler.
There is a zip file included in my /capstone/portfolio/ directory.

Section 2: Teamwork

We engaged in peer review for this project to satisfy the teamwork requirement for the capstone. Team Members each wrote their own compiler, and teammates helped debug and optimize source code on occasion. Each team member wrote new tests to add to the provided test suite to double check the functionality of our program in edge cases and scenarios that may have been overlooked in the original test suite. Additionally, team members published documentation for each other's programs, so that the code would be clear and understandable for anyone who wishes to use our Catscript language compilers. The teamwork section allowed both team members to get comfortable with coding collaboratively as well as thinking out of the box to find possible edge cases or gaps that may not be accounted for in the tests. Overall the teamwork section helped prepare us for a career experience.

In terms of what percentage of the work was completed by each team member, Team member 1 did 95% of the work for his own portfolio, and Team member 2 did about 5% of the work. This work was only supplementary and served the purpose of refining this portfolio. Team member 2 has their own portfolio, in which Team member 1 contributed about 5% of the total work. Due to the nature of this exchange, it is only reasonable to conclude that each team member did equal amounts of work on this project.

Section 3: Design Patterns

The design pattern we chose to implement was the visitor pattern. The main idea with the visitor design pattern is to separate the algorithms and methods from the underlying data structure, so that methods can be added, removed, or changed without having to modify the data structure. In the Catscript compiler, the main data structure that we were using was a Parse Tree. The parse tree is very large and complicated in terms of its code base, and it would be extremely costly to need to make adjustments to it. We used the visitor pattern in the evaluation and compilation steps of the compiler, where a Visitor would walk the parse tree and evaluate, execute, or compile each node. Using this pattern allowed us to be able to adjust the methods and implementations for these steps without having to touch our underlying data structure, the parse tree.

Section 4: Documentation

Here is my documentation for the Catscript language. This link goes to a markdown file where the formatting is a little neater. Section 5 picks up again on page 13.

Catscript Guide

This document serves as an introductory guide to the catscript programming language.

Introduction

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

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

Output:

```
Foo
```

Features

For loops

For loops are used to iterate over a known range of values. This can be particularly helpful to iterate through lists. The following example will iterate through the given list, and print each value.

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

Output:

```
1
2
3
```

You can also use a for loop to iterate over a pre-existing list:

```
var nums : list<int> = [1, 2, 3]
for(x in nums) {
        print(x)
}
```

Output:

```
1
2
3
```

If Statements

If statements are used to perform an action if some given condition is true. In the following example, the program checks the value of a variable named x, and if it is less than 10, it will print the value of x.

```
var x : int = 7
if(x > 10) {
        print(x)
}
```

Output:

```
7
```

Else Statements

We can build on if statements with the else statement:

```
var x : int = 12
if(x < 10) {
        print(x)
} else {
        print(10)
}
```

Else statements will run if the if statement evaluates to false. In this case, if x is 10 or more the program will print 10.

Output: 10

Print Statements

The print statement is used to output values to the console.

print("Hello World")

Output:

Hello World

Types

Catscript supports the following types:

boolean

Booleans can hold the value True or False.

var falseExample : bool = false
var trueExample : bool = true

In catscript, boolean values are typed in all lower case.

integer

Integers in Catscript are represented in 32 bits and can hold any whole number,

i.e 0, 1, 2 3, etc.

var x : int = 10

string

Catscript supports strings, which can consist of any sequence of ASCII characters.

var str : string = "Hello World"

Object

The object type can hold many different types. It is also the default when no type is defined.

var obj : Object = "Object"

Object is the default type when no type is defined.

function x(a, b, c) {}

In the above code snippet, catscript will recognize the type of the parameters a, b, and c as Objects.

list

A list is always assigned to a variable, and follows the standard variable syntax for catscript. To assign a list to a variable, you must specify the variables type as a list, and specify the type of values the list will hold:

var x : list<int> = [1, 2, 3]

In catscript, lists cannot contain values of different types.

null

Catscript supports a null type, which represents the absence of a value.

var x = null

Variables

When declaring a variable, catscript supports both implicit typing as well as explicit typing.

Explicit typing

In the following example, the type is explicitly declared.

var x : int = 10

Implicit typing

In the following example, the type is not explicitly declared, and is instead inferred.

var x = 10

Assignment

Variables can be reassigned after their initial declaration, as long as the type of the new value is the same as the type of the variable's previous value.

```
var x : int = 10
x = 15
```

Scoping

Catscript uses a static scoping system instead of a dynamic scoping system. This means the scope is designated off of the position of the variables in the source code instead of the condition of the runtime stack. Here is an example function that displays the scoping.

```
var x : int = 10
function func() {
        var x : int = 20
   print("Inside func: x = " + x)
}
print("Outside func: x = " + x)
```

Output:

```
Inside func: x = 20
Outside func: x = 10
```

Functions

Functions are a core building block of catscript. They consist of a set of statements, can take 0 or more inputs, and perform a task and/or returns a value.

Function Declaration

Function definition consists of the function keyword, a list of parameters, and the function body. The following code defines a very simple function called HelloWorld, which takes no parameters and prints the string "Hello, World!".

```
function HelloWorld() {
        print("Hello, World!")
}
```

Output:

```
Hello, World!
```

Functions with parameters

to define a function with a set of inputs, or parameters, add the desired parameters between the parentheses:

```
function sum(a, b) {
        print(a + b)
}
```

The above example will print the sum of the parameters a and b when called. If a and b are strings, they will be concatenated.

We can also require parameters to have a particular type:

```
function sum (a : int, b : int) {
        print(a + b)
}
```

This will only allow integers as parameters.

Returning a value

If we wish to return a value from a function, we must provide the return type and add a return statement:

```
function sum:int(a : int, b : int) {
        var total : int = a + b
        return total
}
print(sum(1,2))
```

Output:
3

Now, instead of printing the sum we will return it.

Function Calls

Take the following function:

```
function sum (a : int, b : int):void {
        print(a + b)
}
```

In order to call this function, we will use its name and any necessary parameters as shown in the following example.

```
sum(2, 2)
```

Output:

```
4
```

Math operators

Catscript supports the following four mathematical operators:

Addition

Add two integers together.

```
print(2+2)
```

Output:

```
4
```

Subtraction

Subtract the second integer from the first.

```
print(4-2)
```

Output:

```
2
```

Multiplication

Multiply two integers together.

*print(2*2)*

Output:

4

Division

Divide the first integer by the second integer.

print(4/2)

Output:

2

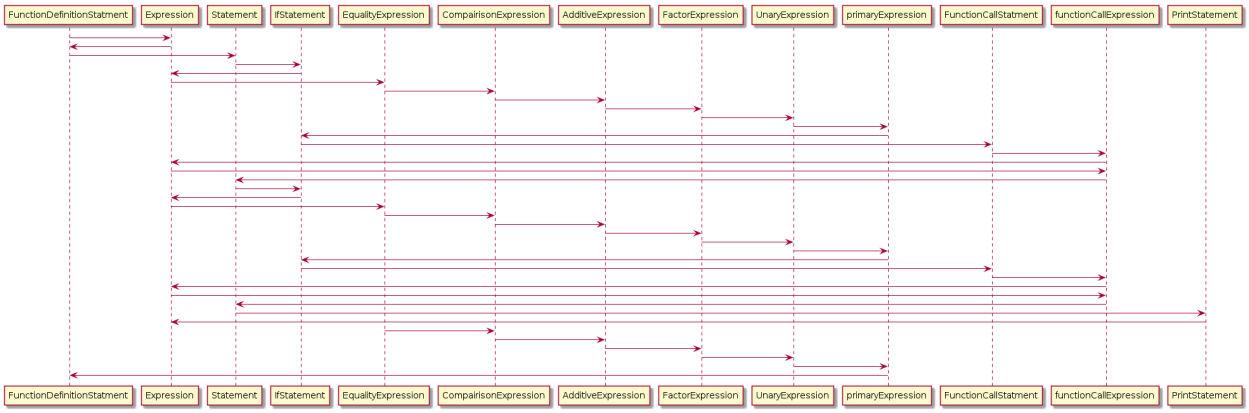
Section 5: UML

Here are the UML diagrams that I made for Catscript Language Compiler.

Here is a UML Sequence Diagram which displays the recursive parsing logic for this example snippet of a Catscript program.

```Catscript
// this function takes an int and calls itself recursively until the input is 0

Function example(input:int):void {
        If (input != 0){
                example(input-1);
        }
        print("Input has been reduced to 0")
}
```



This shows the recursive nature of the parser, as the parser cascades through the grammar and trickles back up from the bottom as primary expressions are evaluated. Eventually, when the exterior recursion from the function is finished, Statement goes down to print statement to output the string "input has been reduced to 0". The parser has to go back up the grammar and parse through to get the string from primary expression, then the parser returns to the top of FunctionDefinitionStatment so that It can continue to parse the code that might follow this function definition.

Section 6: Design Trade-Offs

The primary design trade off we made during this project was to hand write a Recursive Descent Parser instead of using a Parser Generator tool. It is my understanding that typically in academia, tools that automate parser generation and tokenizer generation are widely used to streamline the speed at which a compiler is completed. However, these generator tools can often write code that is very complicated and frankly difficult to understand. They use randomized generic variable names and abstract recursion and looping to achieve a similar effect as a handwritten parser. These tools often end up implementing a recursive descent pattern at the end of the day anyways. So even though it may have taken us more time to handwrite the parser, overall it makes it easier to read and debug, and resulted in significantly less lines of code than a generated parser.

Choosing to handwrite the parser also gave us a tremendous academic advantage. Developing this code by hand gave the students an intimate understanding of the functionality of the parser, and allowed us to control the readability of our code. By writing it ourselves, we were forced to use representative and understandable names for functions and variables, which significantly increased the overall human digestibility and cognition of the compiler. In other academic settings, students may construct a compiler using generative tools, however if they need to make changes or debug this generated code, it would be immensely difficult to understand. While it surely took many many more hours to implement the parser using this method, the students gained a much deeper understanding of the internals of this system on top of producing a more concise, more readable final product.

Section 7: Software Development Life Cycle Model

The software development life cycle model that we used in this project was Test Driven Development. Luckily for the students, most of the tests for this project were prepared by our professor. This is a realistic experience to prepare us for a career as most software companies will have a separate department for testing. The tests were comprehensive to cover all tokenizing, parsing, evaluation, and compilation functionalities that the program would need to successfully compile programs in the Catscript Language. Overall, there was 152 tests included to check the functionality of the tokenizer, the parser, the evaluation of expressions, the execution of statements, and the compilation of functioning java bytecode. As part of the teamwork portion of this class, we wrote supplemental tests to double check the functionality of complex scenarios and edge cases.

This development life cycle model was very helpful to our team. It forced us to make sure that we didn't leave loose ends or dead code in our project. It also allowed for us to be able to monitor our functionality in checkpoints so that we didn't build new software on top of dysfunctional or incomplete code. The tests were organized in a similar fashion to the structure of our parser, and completing the tests in order alleviated the risk of having issues down the road. Consider the example where compilation tests are failing due to an error in the parser implementation instead of an error in the compilation implementation. This problem would be very difficult to debug, and then would likely imply a myriad of changes to be made to all the code that has been written since the bug was introduced. So test driven development helped us ensure complete functionality of top level processes before beginning implementation of lower level processes in the parse tree.