

Section 1: Program

The purpose of this project was to create a recursive descent parser for a new language called Catscript. This parser would parse the language into its respective expression and statement types, these would then be compiled into JVM byte code to be executed.

Source Code: <https://github.com/kieran-ringel/csci-468-spring2022-private/blob/master/capstone/portfolio/source.zip>

Section 2: Teamwork

Team members 1 and 2 contributed equally to this project. They both individually wrote a recursive descent parser that converted Catscript to bytecode. Each team member also provided the other team member with three tests to include in their test driven development. Each partner also wrote up documentation of Catscript and how it worked, this documentation was exchanged between partners to ensure that each team member had a sound understanding of the language.

Section 3: Design pattern

We utilized memoization in the CatscriptType for a function where we are returning the type for a certain list. This is a feature built into the structure of the language.

```
private static Map<CatscriptType, ListType> cache = new
HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
    ListType listType = cache.get(type);
    if(listType != null) {
        return listType;
    } else {

    }
    ListType listType1 = new ListType(type);
    cache.put(type, listType1);
    return listType1;
}
```

Without memoization, this function is called with a given type to return a new list type. With memoization a list type for a given type is stored in a hashmap so the list type does not need to be created every time. If the type does not exist in the hashmap then it is created and stored in the hashmap before returning the corresponding list type.

Section 4: Technical writing.

See documentation on pages 3 and 4.

Section 5: UML.

See class diagram on page 5.

Section 6: Design trade-offs

The largest design decision was choosing to use a recursive descent parser. This was chosen over a parse generator because it is simpler and provided us with a greater understanding of the recursive nature of languages. Its tradeoffs were that the reductive descent parser required more code and is not a standard as a parse generator. We also chose not to use the visitor pattern from *Crafting Interpreters* for the expression problem. Instead evaluation and compilation took place at each parse tree node. The tradeoff was that this did not separate concerns.

Section 7: Software development life cycle model

This project was done using Test Driven Development (TDD). These tests helped the flow through the project as they guided the focus of development. They were valuable in guaranteeing that the project satisfied all of the project requirements. The only downfall of TDD was dependency on tests. If a test passed, it was assumed that part of the project was implemented correctly. When developing down the line, there were times that a test would fail because a previous section was developed incorrectly and would have to be fixed.

Catscript Guide

Introduction

Catscript is a statically typed scripting language comprised of a small type system. This document provides an overview of key features, as well as providing explicit code examples.

Features

Data Types

The following are the data types that are supported by Catsript: - 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

Lists

Lists are a useful data structure for storing multiple elements within a single variable. Lists are specified with `[]` in Catscript.

```
l = [1, 2, 3]
```

For Loops

A `for` loop is used when iteration needs to occur, and can be applied to iterable objects like lists. The `for` loops in Catscript work similarly to `for` loops in python and are identified by the `for` keyword.

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

Unary Expressions

Unary expressions are used to negate values using either `-` or the `not` keyword.

```
-1  
not true
```

The first example results in the numerical value of -1, whereas the second example results in `false`.

Additive and Factor Expressions

Additive and factor expressions are used to support basic mathematical operations. The supported operations are addition, subtraction, multiplication, and division.

```
1 + 1  
3 - 1  
3 * 4  
6 / 2
```

Comparison and Equality Expressions

Comparison and equality expressions are used to support the common logic conditions from mathematics. Catscript supports the following statements:

- Equals: `x == z`
- Not equals: `x != z`
- Less than: `x < z`
- Less than or equal to: `x <= z`
- Greater than: `x > z`
- Greater than or equal to: `x >= z`

If (and Else) Statements

Catscript supports mathematical conditional statements through `if` statements, which are identified by the `if` keyword. `else` statements are used to catch anything leftover from the preceeding `if` statement.

```
if(a == b){
  print(1)
} else {
  print(2)
}
```

Print Statments

`Print` statements are used to display output to the user and are idetified by the `print` keyword.

```
print("Hello World")
```

Assignment Statements

Assignment statements are as simple as assigning a value to a variable.

```
n = 7
```

Variable Statements

Variable statements are used to declare variables and are identified by the `var` keyword. The variable type can also be declared.

```
var x = 21
var y : int = 3
```

**** Both are valid statements ****

Functions

Functions can be created within Catsript to run specific blocks of code at specified times and are identified by the `function` keyword. Parameters can be passed into functions by being specified in the parentheses after the function name with multiple parameters being separated by commas. Optional data types can also be specified for parameters. Data can be returned with the return type being specified in the function definition. Recursive functions are also supported by Catscript.

```
function capstone(done : bool) : string {
  if(done){
    return "graduate"
  } else {
    return "try again"
  }
}
```

The body of a function can contain both statements and return statements.

Return Statements

`Return` statements are used to return values from a function and are identified by the `return` keyword.

```
function foo(x : int) : int
  return x + 1
```

Inspiration for documentation & formatting came from: W3Schools <https://www.w3schools.com/>

