Catscript Compiler Project

Section1: Program

Please see zip file in the capstone/portfolio directory.

Section2: Teamwork

Team member one developed documentation explaining how to use Cat Script in a simple yet intuitive manner and designed unit tests to ensure that the functionality of the Cat Script parser is sound. these Tests can be found in /src/test/java/edu/montana/csci/csci468/partnerTesting/Tests.java within the src zip file and the documentation is presented in **Section 4** Below. Team member one contributed ~20% of the total time it took to complete this project. Team member two primarily wrote the code for the Cat Script parser. This included building the tokenizer, parser, and methods to evaluate the code in both JVM bytecode and Java. Team member one also helped debug challenging aspects of the code and asked critical questions that helped team member two build a more robust compiler. For example, team member one would often suggest theoretical implementations of a certain feature of the compiler and would have a constructive debate about different micro designs pros and cons. Team member two was also responsible for periodically pushing code updates to GitHub and ensuring that the local repository was in synch with the class repository. Team member one and two would typically meet weekly to ensure a holistic understanding of the project on everyone's part and to partition the projects various tasks. Overall team member two contributed ~80% of the total time it took to complete this project.

Section3: Design Pattern

One design pattern that was found to be particularly useful was the memoize pattern. The Cat Script parser uses a class called CatscriptType to store meta information about variables which is critical to validating the semantic correctness of the language. The getListType method within this class takes a CatscriptType like an Integer or Boolean and returns a ListType Object

representing its list form. The memoize design pattern was used to fetch a ListType object given its CatscriptType from a class level cache if it exists. If the corresponding ListType doesn't exist in the cache a constructor is called to generate the ListType object, and it is stored into the cache for future use and is then returned by the method.

It is located at: src/main/java/parser/CatscriptType

-The logic within the getListType method: line 41

-The class level cache: line 18

Section4: Technical Writing

Documentation detailing the features and usage of Catscript is listed below.

Catscript Language Guide

Introduction

Catscript is a modern scripting language designed for readability and simplicity. Its features include immutability, error management, and flexible data types, making programming tasks straightforward.

Core Concepts

No Variable Shadowing: Variables cannot be re-declared within nested scopes. **Immutable Lists**: Lists are read-only after initialization.
Whole Numbers Only: Supports only integer data types.

Basic Syntax

Variable Statement

Variables are initialized with the `var` keyword. The type can be inferred or explicitly declared.

```
var departmentName = "Engineering"

var departmentID: int = 101
```

Data Types

Primitive Types: `int`, `string`, `boolean`
Object Types: Complex objects or lists that can hold mixed data **Collections**: Immutable lists with mixed types

Primitive and Object Type Example:

```
type Project {
    name: string

budget: int }

var projects = [
    Project { name = "Apollo", budget = 5000 },
    Project { name = "Gemini", budget = 3000 }
]
var mixedList: object = ["Research", 42, true, Project { name =
"Orion",
budget = 2000 }]
```

Print Statement

Display output using `print`: `print("Welcome to Catscript!")`

Control Structures

Conditional Statements

Handle conditions using `if` and `else`:

```
if (departmentID > 100) {
    print("Large department")
} else {
    print("Small department")

}
```

Loops

`for` loops iterate through collections of different data types:

```
for (item in mixedList) {
    print(item)
}
for (project in projects) {
    print(project.name + " has a budget of $" + project.budget)
}
```

Advanced Features

Functions and Function Calls

Define functions with `function` and use `return` for output. Call a function using its name and parameters.

```
function allocateBudget(projectName: string): string {
    return "Allocating budget to " + projectName
}
var budgetMessage = allocateBudget("Apollo")
print(budgetMessage)
```

Assignments and Errors

Reassign values to existing variables while respecting their types:

```
var budget = 10000
budget = 8000  // Valid reassignment
var employeeCount: int = 50
// employeeCount = "Full"  // Error due to type mismatch
```

Expressions

Factor: Evaluate the product of numbers `catscript var result = 5 * 8 // 40 print(result)`

Additive: Combine multiple values `catscript var sum = 10 + 5 - 3 // 12 print(sum)`

Comparison: Compare values using relational operators `catscript var isHigher = 15 > 10 // true print(isHigher)`

Equality: Check for value equality `catscript var isEqual = "Research" == "Research" // true print(isEqual)`

List Covariance

Lists can hold items of different types:

```
var employeeData: object = ["Alice", 35, true, Project { name =
"Apollo",
budget = 5000 }]
print(employeeData[0] + " is the team lead for " +
employeeData[3].name)
```

Scoping

Variables within inner blocks can't shadow outer variables:

```
var teamLead = "Alice"
function assignProject() {
    var teamLead = "Bob"  // New scope
    print(teamLead)  // Prints "Bob"
}
assignProject()
print(teamLead)  // Prints "Alice"
```

Error Handling

Manage errors with `try` and `catch`:

```
try {
    var num = parseInt("InvalidInput")
} catch (e: Exception) {
    print("Error: " + e.message)

}
```

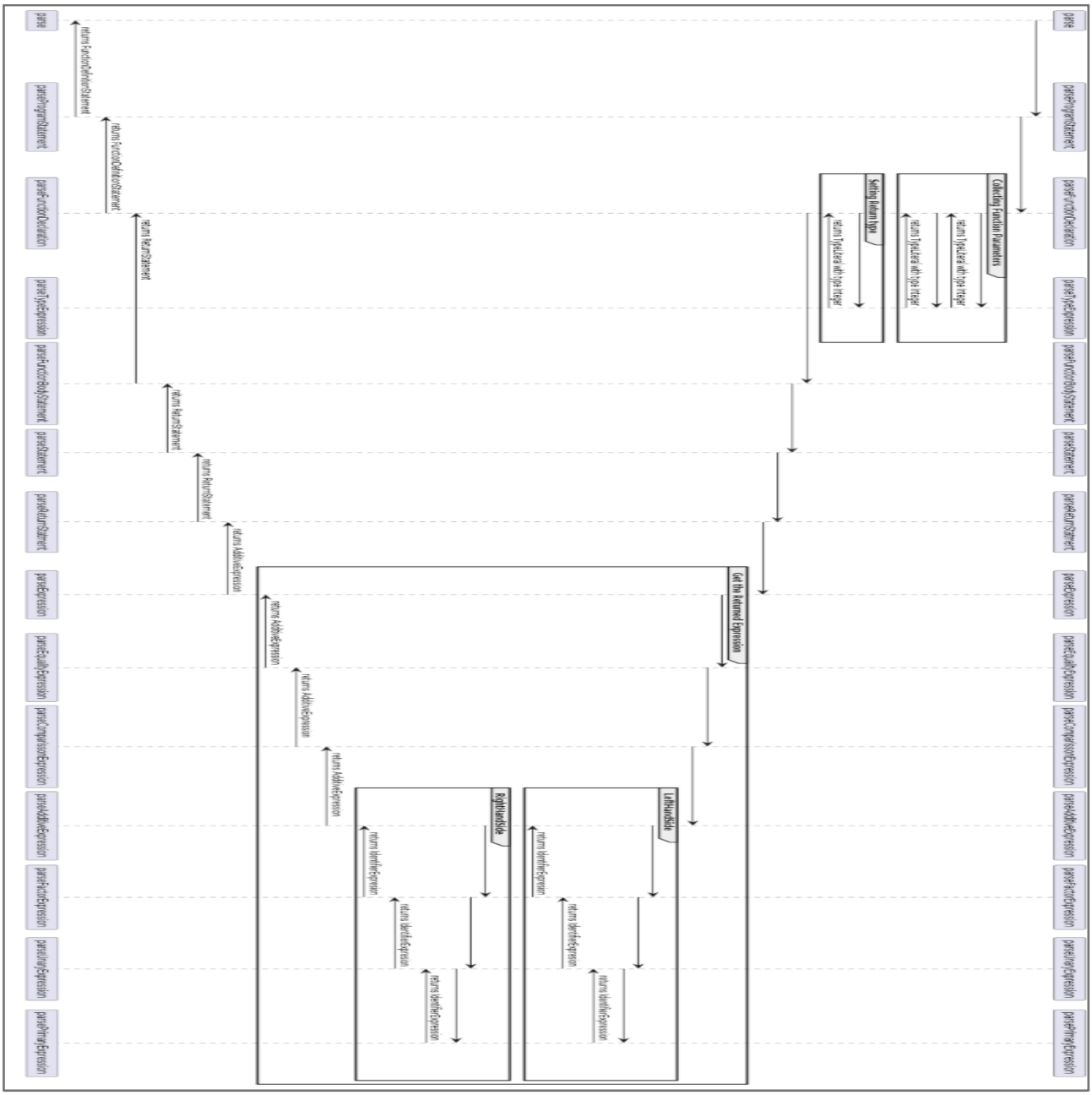
Section5: UML

The diagram below is a sequence diagram for the following snippet of Catscript code.

“function calculator:int (int x, int y){

  return x+y

}”



Section6: Design Trade Offs

The two main ways a programming language can be created is using a parser generator or a hand-built parser. While parser generators offer a way to quickly implement a language that compiles in a reasonable amount of time it comes at the cost of opaqueness and limited flexibility. For example, it is often difficult to embed special features into a language or provide customizable error messages if a parser generator is used. Another drawback is that the code generated by a parser generator is often difficult to interpret and is non-intuitive making it difficult to truly understand how a parser works. Since the primary goal of this class is to walk away with an intuitive understanding of how programming languages work, a parser generator is not an ideal choice. Out of the various methodologies that could be used to build a parser from scratch, recursive decent parsing closely mimics the recursive nature of language in general making it intuitive and easy to understand. Although recursive decent isn't well known for its speed it is still faster than other techniques such as LR parsing. Another benefit of coding a parser from scratch is that it enables much more detailed error messages and highly customizable features or syntax. Being able to code a language at a much more granular level allows for a superior understanding of the nuances and micro-design choices required to bring the language to life. A potential drawback to building a parser from scratch is that it would likely take far longer to build than if a parser generator was used. However, since the goal of the class is to understand how a parser works more so than anything else a recursive decent parser is the one of the best ways to implement the Catscript Programming Language.

Section7: Software Development Life Cycle Model

The Catscript Parser was built using test driven development (TDD) which is where the tests are created before writing the actual code. Test driven development helped to ensure that the parser was relatively bug free and conformed to the catscript grammar. One drawback to TDD is that setting up the tests before hand requires a large upfront cost and could potentially lead to an excessive time spent on testing. Although automation features built into our testing framework Jupiter and the IntelliJ IDEA significantly reduced our testing time these are advanced features that can be difficult to implement correctly. TDD also inspires a mentality within developers where once they set up the tests development becomes centered on mearly passing tests and is

less concerned about creativity. Another drawback to TDD is that if a design choice is reconsidered midway through a project it could be costly to re-write a portion of the test-base to conform to the changes. However, given the clear nature of writing a recursive decent parser our project was unlikely to undergo massive changes midway. If the requirements were less explicit or abstract TDD may not have been a viable approach for building our parser. Overall, TDD created a measurable goal and inspired discipline to adhere to certain quality standards throughout our project. TDD worked well for building our Catscript Parser.