Capstone Portfolio

CSC468 Compilers - Spring 2024

Jake Ripley

Section 1: Program. The source code for my project can be found in the source.zip file in capstone/portfolio

Section 2: Teamwork. Team member two contributed 3 tests and documentation for the catscript language to my project. I was responsible for implementation.

Section 3: Design pattern. In my capstone project, the Memoization design pattern was employed within the getListType method of the CatscriptType.java class. This method is tasked with creating and returning a ListType object based on a given CatscriptType. The pattern is applied as follows:

```java
private static final Map<CatscriptType, ListType> cache = new HashMap<>();
public static ListType getListType(CatscriptType type) {

   if (cache.containsKey(type)) {
       return cache.get(type);
   }

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

In the development of this project, I employed the memoization pattern to enhance performance. Memoization is particularly effective in scenarios where creating new instances of data structures, such as lists, is computationally expensive. By storing the results of expensive function calls and reusing them when the same inputs occur again, memoization significantly reduces the need for repeated calculations. This approach not only speeds up the application but also ensures more efficient use of resources.

Furthermore, the use of memoization helps in avoiding redundancy, especially in the creation of new list types with each function call. This is beneficial because it prevents the system from executing the same operations multiple times when the existing results can be reused. By doing so, the pattern minimizes the computational load and enhances the overall efficiency of the application. This strategy is particularly advantageous in systems where functions are called frequently with similar inputs, as it ensures that performance improvements are both significant and noticeable.

Section 4: Technical writing.

@Test void localVarStatementsWorkProperly() { assertEquals("2\n4\n6\n", compile("for(x in [1, 2, 3]) {\n" + " var y = x * 2\n" + // * 2 " print(y)\n" + "}\n")); }

@Test void printStatementWorksProperly() { assertEquals("2\n", compile("print(1 + 1)")); // 1 + 1 assertEquals("3\n5\n", compile("print(1 + 2)\n" + // 1 + 2 "print(2 + 3)")); // Add 2 + 3 assertEquals("true\n", compile("print(1 < 2)")); // 1 < 2 assertEquals("[12, 22, 32]\n", compile("print([12, 22, 32])")); //2 }

@Test void forStatementWorksProperly() { assertEquals("2\n4\n6\n", compile("for(x in [1, 2, 3]) { print(x * 2) }")); // x * 2 }

Introduction

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

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

Features

For loops

For Loops in Catscript allow iterating over sequences. They can be used to perform actions on each element of a collection, such as printing each item.

```
for (item in [1, 2, 3, 4]) {
    print(item)


}
```

If Statements

If Statements provide conditional logic, executing different code blocks based on the truth value of expressions.

```
if (x > 5) {
    print("x is greater than 5")


} else {
    print("x is not greater than 5")


}
```

Print Statements

Print Statements output the result of an expression to the console, useful for debugging or user interaction.

```
print("Hello, Catscript!")
```

Variable Declarations and Assignments

Variable Declarations and Assignments allow for the definition and initialization of variables.

```
var x: int = 10
x = 20
```

Function Declarations and Calls

Function Declarations and Calls let you define reusable blocks of code which can be called with parameters.

```
function add(a: int, b: int): int {
    return a + b


}
print(add(5, 3))
```

Return Statements

Return Statements are used to exit a function and optionally return a value to the calling context.

```
function getName(): string {
    return "Catscript User"


}
print(getName())
```

List Literals

List Literals enable the creation of lists which can be used to store multiple values in a single variable.

```
var myList: list<int> = [1, 2, 3, 4]
print(myList)
```

Function Calls

Function Calls involve invoking functions by their name and passing the required arguments.

```
var result = add(10, 20)
print(result)
```

Type Expressions

Type Expressions specify the types of variables, parameters, and function return types, enhancing type safety.

```
var flag: bool = true
if (flag) {


   print("Flag is true")
}
```

Return Statements

Return Statements are used to exit a function and optionally return a value to the calling context.

```
function getName(): string {
    return "Catscript User"


}
print(getName())
```

Expressions in Catscript

Arithmetic Expressions

Perform calculations such as addition, subtraction, multiplication, and division, returning numerical results.

Boolean Expressions

Evaluate conditions and comparisons, returning a boolean value (true or false).

List Access Expressions **Retrieve elements from lists, allowing for dynamic data handling.**

Function Calls

Execute functions, using the return values as part of larger expressions.

Variable Types in Catscript

Variables in Catscript are declared with specific types that define the kind of data they can hold. Understanding these types is crucial for effective programming in Catscript

Int

A 32-bit integer capable of holding a whole number. Used for counts, indexes, and scores.

String

A sequence of characters used to store and manipulate text. Supports operations like concatenation and substring extraction.

Bool

A boolean value, either true or false, used in conditional logic and loops.

List

Represents a list where x is the type of items in the list. Lists are versatile data structures suitable for storing ordered collections.

Null

Represents the absence of a value. Useful for initializing variables and handling optional data.

Object

A generic type capable of holding any type of value, providing flexibility in data handling.

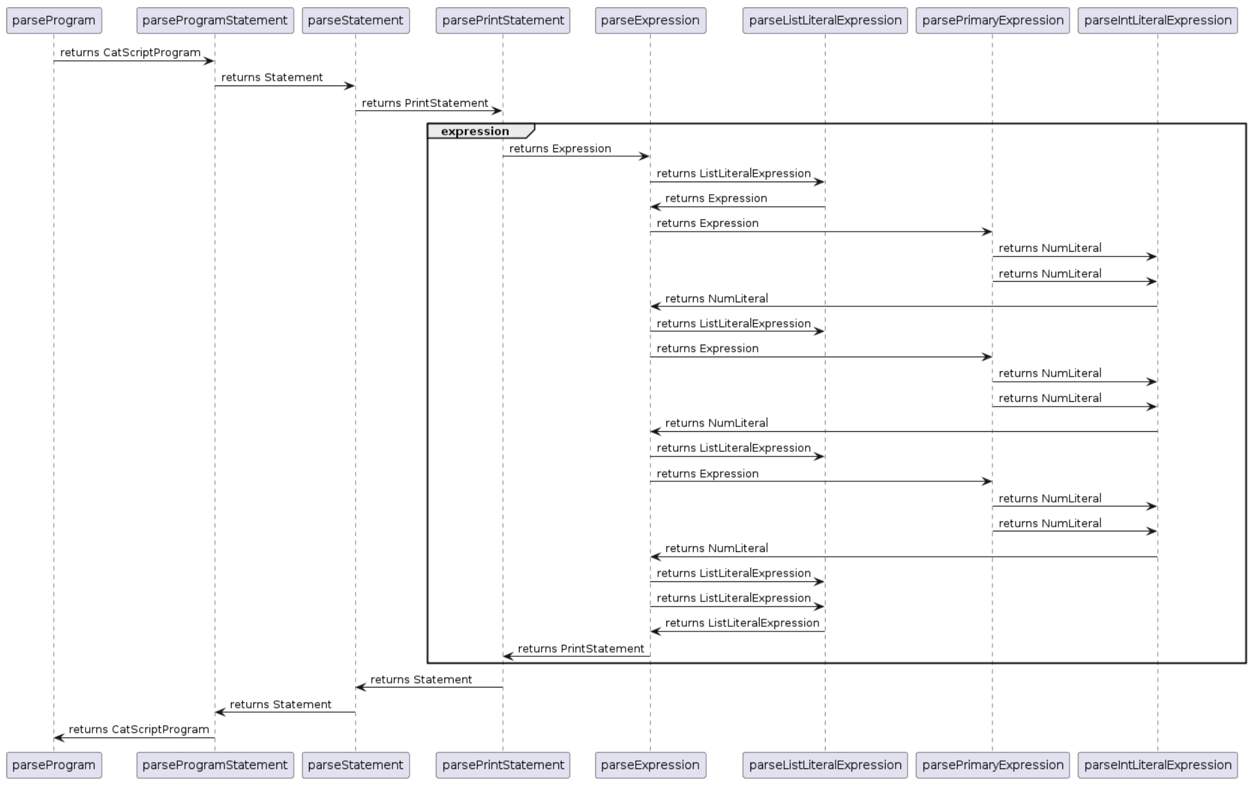
Usage Examples

```
var count: int = 10
var welcomeMessage: string = "Hello, Catscript!"
var isValid: bool = true
var numbers: list<int> = [1, 2, 3, 4]
var optionalName: string = null
var anyValue: object = "This can be anything!"


if (count > 5) {
    print("Count is greater than 5")



}

print(welcomeMessage)
print("Is valid? " + isValid)


for (num in numbers) {
    print(num)



}

print("Optional Name: " + optionalName)
print("Any value: " + anyValue)
```

These examples demonstrate variable declarations, basic operations, and the practical use of different types in Catscript, providing a foundational understanding for programming in this language.

Section 5: UML

' sequence diagram for parsing "print([1, 2, 3])"



The provided sequence diagram represents the process of parsing a print statement that includes a list literal [1, 2, 3] as its argument in the Catscript programming language. This is a detailed breakdown of each step in the sequence diagram, which traces how the parser processes the input from recognizing the program structure down to handling individual elements of the list.

Section 6: Design trade-offs.

In this project, I opted for a recursive descent parser instead of a parser generator. The primary reason for this choice was the enhanced readability offered by recursive descent. This approach not only makes the code easier to follow but also grants more control and flexibility during the development process. Furthermore, the inherent readability of recursive descent greatly simplifies debugging. This method allows developers to identify and resolve issues more efficiently, which is crucial in maintaining the integrity of the parser.

Additionally, the recursive descent method enhances error handling capabilities. It provides the opportunity for immediate and context-specific responses to errors, which in turn improves the clarity and usefulness of error messages. However, it's important to recognize the trade-offs associated with this choice. Although recursive descent parsers are beneficial for their readability and error handling, they fall short in efficiency and handling complex grammars compared to parser generators. Parser generators excel in these areas, particularly during the optimization phase of parser development, making them suitable for more complex parsing tasks.

Section 7: Software development life cycle model.

In this project, I implemented the test-driven development (TDD) methodology, which I found to be highly effective. Utilizing TDD allowed for a deeper understanding of the project requirements, as frequent testing and debugging provided clear insights into the functionality and performance expectations. This approach encourages a more thorough exploration of potential issues and helps in clarifying the objectives of various components within the software.

However, adopting TDD wasn't without challenges. Initially, there was a significant learning curve associated with understanding the application programming interface (API) and mastering the tests. This phase required time and patience to overcome. Additionally, a notable concern with using TDD was the risk of incomplete test coverage. Despite our efforts, there were areas within the project that may not have been fully tested, potentially leaving some functionality unchecked and not performing as intended. This highlighted the importance of continuously refining test cases and ensuring comprehensive coverage to maintain the integrity and reliability