CSCI 468

aleksandr means Wei You

Fall 2024

Teamwork

In this collaborative project, Team Member 1 and Team Member 2 worked together to develop the CatScript compiler. Team Member 1 took the lead on the coding aspects of the project, implementing the core functionality and ensuring the software met all technical specifications. This role involved not only writing clean and efficient code but also integrating various system components and optimizing the software for performance and scalability.

Team Member 2 focused on designing comprehensive tests and creating detailed documentation. Team Member 2 developed a suite of tests, which helped identify and rectify potential issues early in the development cycle. Additionally, the documentation prepared by Team Member 2 was pivotal in supporting both the end-users and future developers, providing clear guidelines on how to use and maintain the software effectively.

The synergy between Team Member 1's technical expertise in coding and Team Member 2's skills in test design and documentation ensured a well-rounded development process, leading to a software solution that was not only functional and reliable but also well-documented and easy to adopt.

Design Pattern

For this project, I used the memoization design pattern. This design pattern is used to optimize function calls by caching its results. This allows a function to avoid doing expensive calculations when making calls with the same inputs. In the Catscript compiler, I used memoization to improve the getListType method in the CatscriptType class. In particular, I used a HashMap to create a map from a type to the corresponding ListType.

```
static final HashMap<CatscriptType, ListType> CACHE = new HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
    ListType listType = CACHE.get(type);
    if (listType == null) {
        CACHE.put(type, new ListType(type));
    }
    return listType;
}
```

Technical Writing

Catscript Guide

Expressions

AdditiveExpression Additive expressions handle arithmetic addition and subtraction. They operate on numeric values and are used to perform basic arithmetic operations. The result of an additive expression can be used anywhere a number is valid in Catscript.

```
var result = 5 + 3 - 2
```

BooleanLiteralExpression Boolean literal expressions represent boolean values, which are either `true` or `false`. These expressions are often used in conditional statements and loops. Boolean literals are fundamental in controlling flow and logic in Catscript.

```
var isCat = true
```

ComparisonExpression Comparison expressions compare two expressions using relational operators like greater than or less than. They are used to make decisions based on comparing values, such as in conditional statements.

```
var isTaller = height1 > height2
```

EqualityExpression Equality expressions compare two expressions for equality (`==`) or inequality (`!=`). These are crucial for condition checks where exact matches or differences are necessary.

```
var isEqual = (5 * 2) == 10
```

Expression The base type for expressions in Catscript, used to represent any valid combination of operations and literals.

```
var x = 1 * 1 + 1 / 1
```

FactorExpression Factor expressions handle multiplication and division among numeric values. They are used where products or quotients of numbers are needed. Along with factoring with precedence, multiplication and division from left to right to addition and substraction alike.

```
var product = 8 * 3
```

FunctionCallExpression Function call expressions execute a function with the provided arguments. They are essential for reusing code and organizing logic into manageable sections.

```
var greeting = getGreeting("Mittens")
```

IdentifierExpression Identifier expressions reference variables or functions by their names. These are used throughout Catscript to access and manipulate data stored in variables.

```
var age = 10
print(age)
```

IntegerLiteralExpression Integer literal expressions represent numeric integer values. They are the simplest form of expressing numeric data.

```
var count = 42
```

ListLiteralExpression List literal expressions create lists of elements. They are used to group related data together, which can be iterated over or accessed by index.

```
var colors = ["red", "green", "blue"]
```

NullLiteralExpression Null literal expressions represent the null value, indicating the absence of any object. This can be used to denote missing or uninitialized data.

```
var myObject = null
```

ParenthesizedExpression Parenthesized expressions explicitly dictate the precedence of operations. This helps in managing complex arithmetic or logical expressions.

```
var result = (1 + 3) * 2
```

StringLiteralExpression String literal expressions represent sequences of characters. They are used to work with text data in Catscript.

```
var greeting = "Hello, World!"
```

SyntaxErrorExpression Syntax error expressions indicate an unrecognized or misplaced token during parsing. This helps in debugging by signaling where the error occurred.

```
var x = 10 +  // Missing operand
```

TypeLiteral Type literals explicitly declare the type of data. They are used to ensure variables are used consistently within their intended type constraints.

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

UnaryExpression Unary expressions involve a single operand and an operator such as negation. They are used to change the sign of numbers or logically negate boolean values.

```
var negative = not true
```

Statements

AssignmentStatement Assigns a new value to an existing variable. This is fundamental in any programming language for updating the state of the program.

```
x = 20
```

CatScriptProgram Defines the entire Catscript program, which consists of a series of statements. This is the top-level structure that organizes and executes the code.

```
var x = 10
print(x)
```

ForStatement Iterates over elements typically in a collection. For loops are essential for executing a block of code multiple times with different values from a sequence.

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

FunctionCallStatement Executes a function and handles the result. Function calls are used to execute defined functions within the program.

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

FunctionDefinitionStatement Defines a new function with parameters and an optional return type. Function definitions are crucial for encapsulating code into reusable blocks.

```
function greet(name: string) {
    print("Hello, " + name + "!")
}
```

IfStatement Executes blocks of code based on conditional expressions. If statements are the basic form of conditional logic in Catscript.

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

PrintStatement Outputs a string or value to the console. Print statements are used for debugging and user interaction by displaying data.

```
print("Printing to console")
```

ReturnStatement Exits a function and optionally returns a value. Return statements are used within functions to send values back to the caller.

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

Statement Represents the base type for all executable statements in Catscript. Statements form the building blocks of executable code.

```
var isReady = true
```

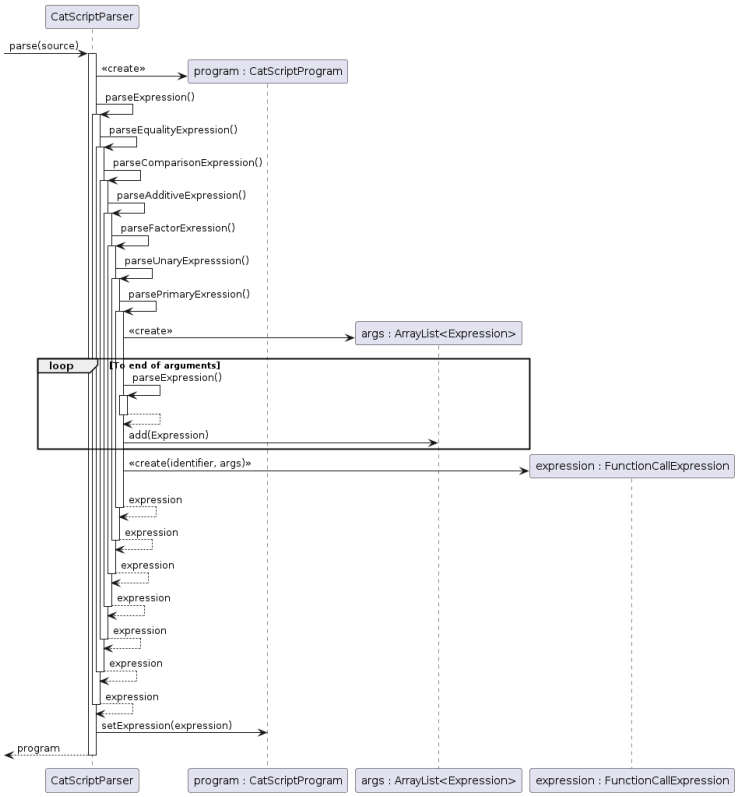
SyntaxErrorStatement Indicates an error in the source code syntax. Syntax errors prevent the program from executing correctly and must be resolved.

```
if (x > 5  // Missing closing parenthesis
```

VariableStatement Declares and initializes a new variable. Variable statements are used to create and set initial values for identifiers that store data.

```
var name: string = "Mittens"
```

UML Diagram



In this UML diagram, I am showing a sequence diagram for the parser when it parses a function call expression. To start, the `CatScriptParser`'s `parse(source)` method is called, where the `source` is a string representation of the code to be parsed. The parser also initializes a `CatScriptProgram` that will be returned when the parser has finished. The diagram then shows how it will call the `parseExpression()` method, which will descend through the parse methods until it identifies the kind expression that needs to be parsed. In our example, once the parser identifies the function call syntax in the parsePrimaryExpression() method, the parser creates the `ArrayList<Expression>` object named `args`. The parser will then loop over the arguments provided in the script and add those expressions to `args` until it reaches the end of the provided arguments. Once the parser reaches the end of the argument list, it creates a new FunctionCallExpression with a name and `args`, and returns that expression through each expression parser method. The CatScriptParser then adds that expression to the program, and returns the program to the `parse(source)` caller.

Design Trade-Off

The compiler was programmed with a Recursive Descent design.

Using a Recursive Descent design has the advantage of being simple to implement, since the structure of the grammar is reflected in the structure of the code. This makes handling complex grammars easier, and helps maintain more readable code. Additionally, this makes it easier to update and maintain the code in the future, especially in contrast to Bottom-Up design methods, which tend to have more complex parser code.

However, the trade-off of using a Recursive Descent parser is that Bottom-Up parsers are significantly more time and space efficient. One reason why Recursive Descent parsers are less efficient is because they have to make guesses about what a statement might be in a grammar, and must backtrack on incorrect guesses. Design considerations also have to be taken when using a Recursive Descent parser because certain grammatical structures can cause problems for Recursive Descent, such as left recursion.

The Catscript grammar was simple enough that the predictive nature of using this style of parser was not going to be a concern, and since the scripts being produced would be simple enough, efficiency was a low priority. Meanwhile, it was important that the compiler be produced quickly, and that it was simple to understand. Therefore, I decided that using a Recursive Descent design was the best choice given these circumstances.

Software Development Model

For this project, I used a Test Driven Development model. Test driven development is a software development strategy that focuses on the creation of feature tests based on outlines before writing the code. You then write enough code that you can pass the test, which encourages writing minimal code that is simple and easy to understand. Once the code is written, you run the test to ensure it passes and refactor the code if it does. This process is repeated until every feature is completed, or every time a new feature is added.

I used test driven development by setting small test goals that covered simple unit cases of the compiler. The sum of these parts would come together to provide a full test coverage of the compiler. When a test would unexpectedly fail, it provided a focused slice of the compiler's functionality that I was able to debug at a granular level. Doing this allowed me to find bugs quickly, and narrow down problems to highly specific use cases, which assisted with deducing where errors were occurring.

This style of development was helpful for me because it kept the goals for the project in a tight scope and allowed me to focus solely on what I needed. Trying to pass the tests also provides a productive dopamine cycle, which helped with keeping my motivation up. It also helped with maintaining functionality in between refactors. Knowing that breaks caused by changes were likely to be caught meant I was able to make changes to my code with confidence.