2024

CSCI 468 - Compilers

SPRING 2024

CODY FINGERSON & YABETS EBREN

SECTION 1: PROGRAM

The source code for this project can be found inside the included source.zip folder.

To view the code, create a Maven Java project. Next replace the src folder with the src folder produced by the zip file.

SECTION 2: TEAMWORK

This project was organized with each team member assigned specific tasks and sections to focus on. The tasks include managing the code base, executing tests, and constructing documentation.

Team member one managed the GitHub repository and was the main contributor to the code base. Their programming efforts were primarily directed at completing the tokenizer, parser, evaluation, and bytecode components of the compiler, ensuring that each checkpoint was submitted by their respective deadlines.

Team member two was responsible for drafting the technical documentation, including this report, and for developing program tests to support the test-driven development environment. Communication between the two team members was conducted though Discord, as in-person meetings were not necessary.

ESTIMATED CONTRIBUTIONS

Total estimated hours: 140

Team Member One -

Contributions: Code base management, implementation, and debugging.

Estimated Hours: 100 hours

Team Member Two -

Contributions: Documentation and tests.

Estimated Hours: 40 hours.

SECTION 3: DESIGN PATTERN

The design pattern used in this project is commonly known as memoization. This design pattern was chosen due to its ability to optimize expensive, recurring function calls. Essentially, this approach will store the result of a specified function call, enabling it to bypass repeat computation. This design pattern is essentially a method of implementing a cache.

To use the implementation, the function needs to store its input values and output into a cache whenever there are new arguments when it's called. The cache is usually implemented using a HashMap. Whenever the function is called, it will then check its parameters against the existing parameters in the cache. If the parameters are equivalent, the function will return the previous computed values from the cache associated with those arguments.

Using the memoization technique can save a significant amount of time for extremely time-consuming computations by avoiding redundant, and unnecessary computations. Additionally, it has the potential to conserve other system resources such as RAM and CPU times. However, there are still drawbacks that should be considered. Firstly, the technique required time to check whether a function call has already been cached or not. Secondly, it is most effective for functions that are frequently called, have a limited range of inputs, and do not rely on random numbers. Moreover, this is not a thread-safe approach. Despite these limitations, it is suitable for this project, however, may not be suitable for a production environment.

The implementation of this design pattern can be found at:

source.zip/src/parser/CatscriptType.java

Catscript Documentation

Overview

CatScript is a transpiled scripting language that has basic functionality.

Key Features

- **Data Types**
  - CatScript offers five primary data types and one complex type:
    - Integer: a 32-bit integer
    - String: a string similar to Java's string
    - Boolean: a true/false value
    - List: a collection of values of type 'x'
    - Null: represents the absence of a value
    - Object: a versatile type capable of representing any value
  - Variables can be declared with or without specifying types.
  - Lists are special types that can store values of any of the other five types.

```
var age = 25
var name : string = "Fluffy"
var isHungry : bool = true
var myList : list<int> = [1, 2, 3, 4, 5]
var nothing = null
```

- **Basic Operations**
  - CatScript supports basic arithmetic and logical operations:
    - Addition (+)
    - Subtraction (-)
    - Multiplication (*)
    - Division ()
    - Logical negation (not)
    - Negation (-)
    - Boolean values (true, false)
    - Null
  - Examples:

```
var result = -1 + 1 * 4  // Result: 3
if(not isHungry)  // Check if not hungry
if(age != null)  // Check if age is not null
else if(false)  // This block won't execute
```

- **Conditional Statements**
  - CatScript allows conditional statements in various forms:
    - If statements without else
    - With an else clause
    - Or with an else if
    - Mixing and matching is allowed as long as an if statement precedes an else.

```
if(age > 18){
    var message = "You are an adult"
}

if(isHungry){
    var message = "Time to eat"
}
else {
    var message = "Not hungry"
}

if(name == "Fluffy"){
    var message = "Hello Fluffy!"
}
else if(name == "Whiskers"){
    var message = "Hello Whiskers!"
}
else {
    var message = "Hello unknown cat!"
}
```

- **Equality & Comparison**
  - CatScript supports basic equality and comparison expressions:
    - Less than (<)
    - Less than or equal to (<=)
    - Greater than (>)
    - Greater than or equal to (>=)
    - Not equal (!=)
    - Equals (==)
  - They can be used in if and else statements.

```
if(age < 30)
if(name == "Fluffy")
if(age <= 25)
if(age >= 18)
if(name != "Whiskers")
if(age == 25)
```

- **Printing**
  - Any expression can be printed using the `print` statement.

```
print(age)
print(name)
```

- **Loops**
  - For loops can iterate through lists.

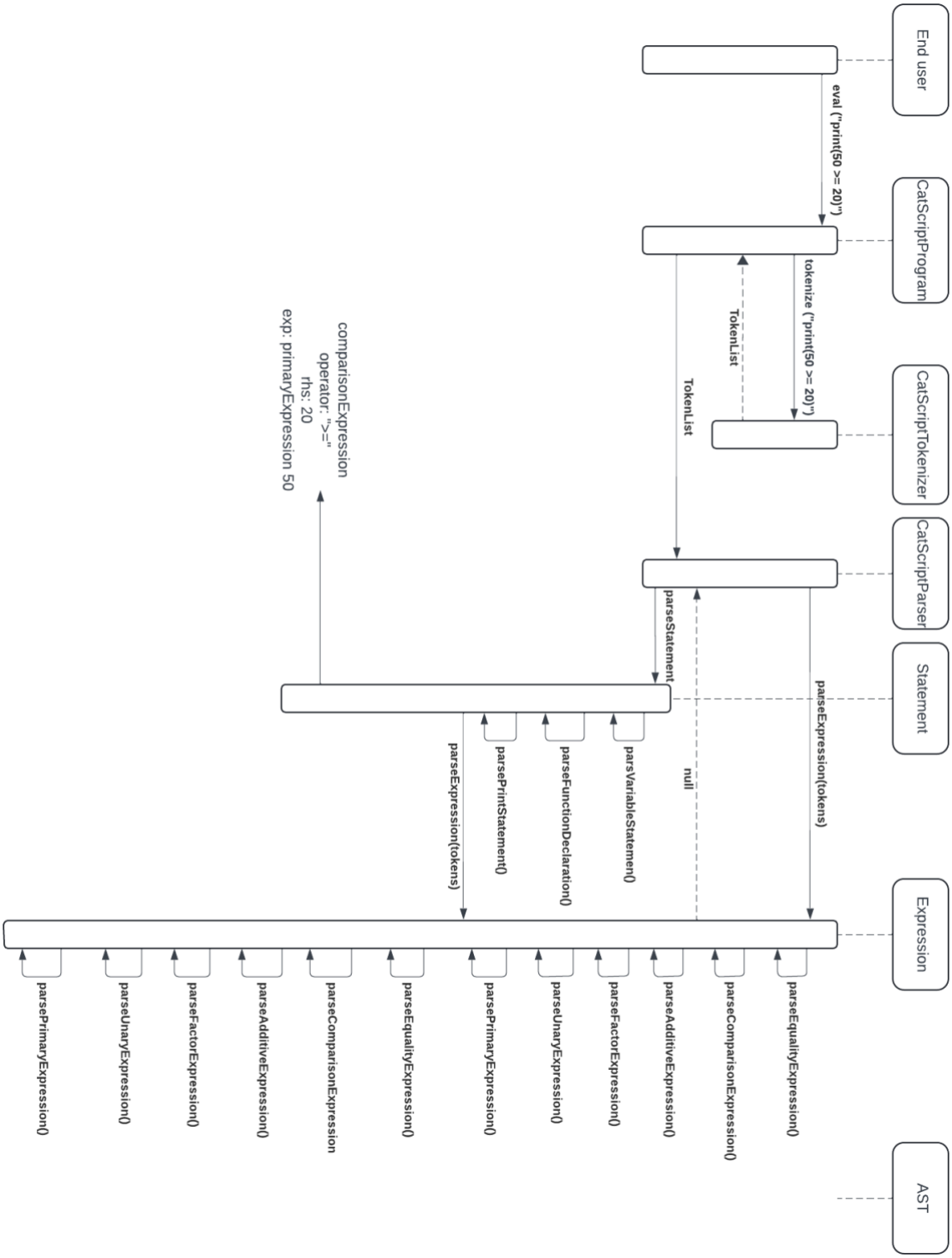
```
var myNumbers = [1, 2, 3, 4, 5]

for(number in myNumbers) {
    print(number)
}
```

- **Functions**
  - Functions are declared with a body.
  - They can be called with function call statements.
  - Functions can have any number of parameters.
  - Optionally, functions can return a value using the `return` statement.

```
function greet() {
    print("Hello, world!")
}

greet()

function add(x : int, y : int) {
    return x + y
}

var sum = add(5, 3)
print(sum)  // Output: 8
```

This concludes the CatScript manual. Enjoy scripting with your feline companions!

SECTION 5: UML



The above diagram represents a sequence diagram of how the compiler will handle a simple comparison of 50 >= 20. Starting with the string given and ending with the elements of the abstract syntax tree at the end. The top value of the abstract syntax tree will represent the left-hand side and the bottom value represents the right hand side of the expression.

SECTION 6: DESIGN TRADE-OFFS

Throughout the development of CatScript, we made several tradeoffs, one of which involved using a recursive descent parser instead of a parser generator. With this approach, we manually defined and parsed tokens, providing us with greater control and a more debug-friendly tokenizer. While parser generators are easier to write and handle many technical details automatically, our manual approach offered better readability and easier debugging, outweighing the tradeoff for writability. Additionally, using a recursive descent parser eliminated the need for the visitor pattern, as we had complete control over the evaluation lifecycle.

Recursive Descent:

This top-down parsing algorithm is widely used in industry for parsing tokenized source code. It offers simplicity and a clearer understanding of grammars and parsing, as each production in the grammar corresponds to a method named after it. This method calls other methods for the right-hand side of the production, following a recursive nature that is easy to understand.

Parser Generators

Parser generators consist of a lexical grammar as a regex and a long grammar as an EBNF. The lexer is generated by mixing code generation with the grammar, while the parser produces an Abstract Syntax Tree. While theoretically requiring less code and infrastructure, parser generators are more complex to learn and use, making them less popular in industry compared to recursive descent parsers.

After considering the complexity, implementing a recursive descent parser made more sense. It offers simplicity, ease of understanding, wider industry usage, and better comprehension of parsers and compilers.

SECTION 7: SOFTWARE DEVELOPMENT LIFECYCLE

The model that we utilized during the creation of this capstone project was the Test Driven Development model. Test-Driven Development is a programming approach that emphasizes simultaneous coding, testing, and design. Developers create the minimum code necessary to pass tests, resulting in cleaner, more concise code that is less error-prone. Test-Driven Development allows programmers to focus solely on essential aspects, temporarily setting aside advanced or unrelated features. This approach not only ensures the functionality of the code but also provides a clear roadmap, guiding towards project completion.