Compilers(CSCI 468)

By: Sam Bierens

Partner: Edward Aldeen

Gianforte School of Computing, Montana State University

Professor: Mr. Carson Gross

Spring 2024

Section 1: Program

A zip of the source file is included in the Portfolio.

Section 2: Teamwork

For our project, There were two team members. Team member 1 was the primary Developer responsible for writing the source code for the following checkpoints: Tokenizer, Parser, Evaluator, and Bytecode Generator. The code was based on the initial shell/framework given to us by the instructor, Carson Gross. The initial implementation was for the Tokenizer to read and separate the string tokens from the CatScript source code. Following this, the Parser was implemented, converting the tokens into the respective expressions and statements based on the EBNF CatScript Grammar, we were given. In support of the Parser, the Evaluation logic was implemented to validate its functionality. Finally, the Bytecode Generator was implemented for bytecode usage.

On the other hand, Team Member 2 was the quality tester, integrating a few tests for each checkpoint section besides Bytecode Generator. Additionally, the technical documentation for Catscript was created, as seen in section 4. Team Member 1 made up 90% of the project, spending roughly 80 hours on the code creation, while Team Member 2 made up the other 10%, spending roughly 15 hours creating the documentation and tests for the Compiler.

Section 3: Design pattern

Our program used the Design Pattern technique of Memoization, an efficient program that optimizes recurring functions. We store the output of costly functions. Thus, when the program calls the function again, we load the previous output, saving us the need to re-compute the function repeatedly, saving memory and computing power.

```
static Map<CatscriptType, CatscriptType> CACHE = new HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
    ListType LT = new ListType(type);
    CatscriptType PM = CACHE.get(type);
    if (PM != null) {
        return PM;
    } else {
        CACHE.put(type, LT);
        return LT;
    }
}
```

In our Memoization function, as shown in the example above, we create a Cache and treat each function call passed through and into the Cache as if it does not exist. For our case, we are Caching ListType such as integer, string, and object list types where when the program calls the function, it checks if the ListType was already generated/computed, and if so, it returns the stored result. In contrast, If the ListType has yet to generate, it stores the result in the Cache for later use and returns the results.

Section 4: Technical writing. Include the technical document that accompanied your capstone project.

Catscript Documentation

Introduction

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

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

Features

Program Statement

The Program Statement marks the start of each program, initiating the execution of either Statements or Function Definition Statement.

Statement

For Loops

Catscript supports the for loop for iterating over elements in a list. The syntax for a for loop is as follows:

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

The **for** keyword signals the beginning of a for loop. The opening parenthesis **(** marks the start of the loop's initialization section, and where the arguments will be. An **IDENTIFIER** is used to represent each element of the sequence during iteration. Next an **in** keyword indicates that the loop will iterate over the elements of a sequence. The **expression** determines the sequence or collection over which the loop iterates, which can be an array, list, or any other iterable object. Afterwards a closing parenthesis **)** marks the end of the loop's initialization section. The following opening brace **{** marks the beginning of the loop's body, where the statements to be executed during each iteration are enclosed. The for loop body allows for multiple statements, and even nested for loops. Finally the closing brace **}** marks the end of the loop's body.

If Statement

Catscript includes conditional logic through the use of if statements. The syntax for a if elif else statement is:

```
if( x > 10 ){
  print(x)
} else if (x < 10) {
  print(9)
} else {
  print(10)
}
```

The **if** keyword signals the beginning of an if statement. The opening parenthesis **(** that follows it marks the start of the conditional expression. The **expression** inside the parentheses evaluates to a boolean value. If that **expression** evaluates to true, the statements inside the if statement's body will be executed. Should that **expression** evaluate to false, the program will move to the next else if, or if none remain, the else statement. Next the closing parenthesis **)** marks the end of the conditional expression. Afterwards the opening brace **{** marks the beginning of the block of statements to be executed if the condition evaluates to true. Within the if block, one or more statements can be executed if the condition is true. Finally the closing brace **}** marks the end of the if block. Optionally an **else** clause can be included after an if/else if block. Additional if statements can be nested within the respective body sections to create conditional branches. If there are no statements to be executed in the else block, an empty brace **{}** can be used.

Print Statement

To output values to the console, Catscript provides the print statement. The syntax is as follows:

```
print(x)
```

First the **print** keyword signals the beginning of the print statement. The opening parenthesis **(** marks the start of the expression to be printed. Next the **expression** inside the parentheses specifies the data to be printed, which can be a **IDENTIFIER**, **STRING**, complex **expression**, etc. Finally the closing parenthesis **)** marks the end of the expression.

Variable Declaration and Assignment

You can declare variables in Catscript using the var keyword. Variables can also explicitly specify their type:

```
var x = 10
var x : bool = true
```

The **VAR** keyword is used to declare a new variable. Following the **VAR** keyword, an **IDENTIFIER** is specified to name the variable. After the identifier, an optional type annotation : can be provided to specify the data type

of the variable. It isn't necessary due to implicit types, but it helps in enforcing type safety in Catscript. Then the assignment operator **=** is used to assign a value to the variable. Finally the expression on the right-hand side of the assignment operator represents the initial value assigned to the variable. It can be a literal value, a variable, or a complex expression.

the **Assignment Statement** is very similar to **Variable Declaration**. The syntax is as follows:

```
x = 10
y = "message"
```

Similar to a variable declaration, the **IDENTIFIER** represents the variable to which the value will be assigned. It must match the name of an existing variable declared in the program. Next the assignment operator **=** is used to assign a value to the variable specified by the **IDENTIFIER**. The **expression** on the right-hand side of the assignment operator represents the value to be assigned to the variable. Similar to variable declaration, it can be assigned the same values. A key difference is the lack of the **VAR** keyword and the inability to explicitly declare type.

Function Call Statement

The function call statement allows the program to invoke already defined functions. The syntax is as follows:

```
func (1, 2, 3)
```

the function call statement calls a function call expression, which in turn calls the actual function. The function name, represented by the **IDENTIFIER**, identifies the function to be called. It must match the name of a function already declared within the program. Next the opening parenthesis **(** marks the beginning of the **argument list**. The **argument list** method contains expressions separated by commas, representing the arguments passed to the function. Finally the closing parenthesis **)** marks the end of the argument list, and the end of the function call.

Function Declaration Statement

Catscript allows you to define custom functions using the function keyword. The syntax for a function declaration is as follows:

```
function x(a : object, b : int, c : bool) : int {
  return 10
}
```

First the **function** keyword signifies the beginning of a function declaration statement. It is followed by the name of the function, represented by the **IDENTIFIER**. Inside the parentheses **()**, the **parameter_list** defines the parameters that the function accepts. Each parameter consists of an **IDENTIFIER** followed by an optional type annotation (**: type_expression**) and are seperated by commas. Finally the **function body** is enclosed

within curly braces **{}**. It contains statements and optionally a **return statement**. The statements inside the function body define the behavior of the function when called. Additionally, a function declaration may include a return type annotation (**: type_expression**). This specifies the type of value that the function returns.

Expression

Expression is the generic expression function, and when called, will recursively descend through all possible expressions until something is returned. The syntax for an expression is as follows:

```
x
true
null
1 + 1
2 * 3
[1, 2, 3]
-1
x < 10
true != false
```

Expression first calls **equality_expression**, and continues from there.

Equality Expression

An equality expression consists of one or more comparison expressions connected by equality operators (**==** for equality, **!=** for inequality). This allows for the comparison of values to determine equality or inequality. The syntax is as follows:

```
1 == 1
true != false
```

Equality expression recursively calls **comparison_expression**

Comparison Expression

A **comparison expression** consists of one or more **additive expressions** connected by comparison operators (**>**, **>=**, **<**, **<=**). These operators are used to compare values and produce a boolean result. The syntax is as follows:

```
5 > 0
3 <= 3
```

Comparison expression recursively calls **additive_expression**

Additive Expression

An **additive expression** consists of one or more **factor expressions** connected by additive operators (**+** for addition, **-** for subtraction). It allows for arithmetic operations such as addition and subtraction. The syntax is as follows:

```
1 + 1
4 - 2
```

Addtive expression recursively calls **factor_expression**

Factor Expression

A **factor expression** consists of one or more **unary expressions** connected by factor operators (* for multiplication, **/** for division). It allows multiplication and division operations. The syntax is as follows:

```
1 * 1
1 / 1
```

Factor expression recursively calls **unary_expression**

Unary Expression

A **unary expression** consists of a unary operator (**not** for logical negation, **-** for arithmetic negation) followed by another **unary expression** or a **primary expression**. It allows for logical or arithmetic negation. The syntax is as follows:

```
- 1
not not true
```

Unary expression recursively calls **primary_expression**

Primary Expression

A primary expression is the most basic element of an expression. It can be an **IDENTIFIER**, a **STRING** literal, an **INTEGER** literal, a **BOOLEAN** literals such as **true** or **false**, **null**, a **list literal**, a **function call**, or a nested **expression** enclosed in parentheses. The syntax is as follows:

```
x
Message
1
true
false
```

```
[1, 2, 3]
foo(fighters)
(1 == 1)
```

List Literal is its own seperate method, and its syntax is as follows:

```
[]
[1, 2, 3]
```

Type Expression

A type expression in Catscript specifies the data type of a variable, parameter, or function return value. The syntax is as follows:

```
x : object
var y : int = 1
function z() : bool {
  return true
}
```

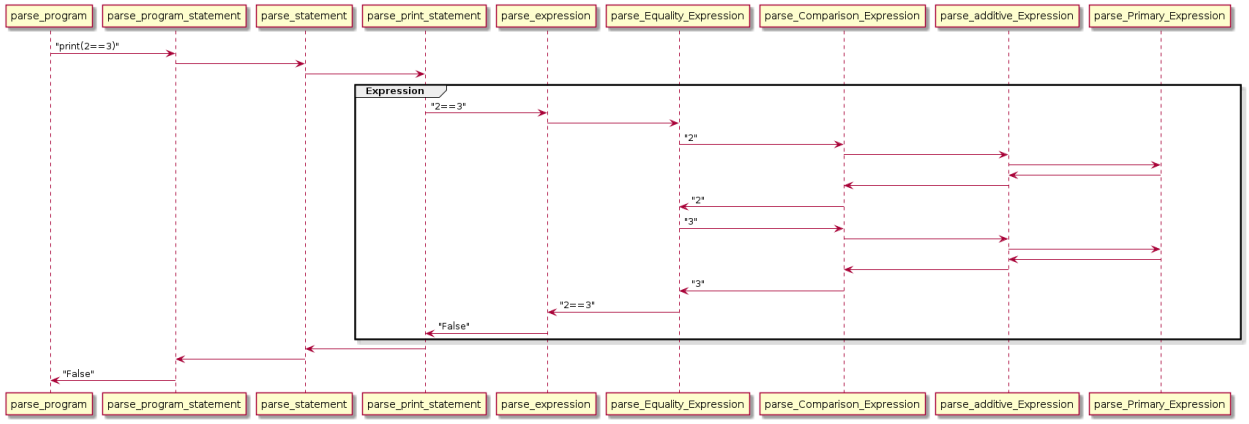
The base types in Catscript are listed below. The list-type can be further specialized with a generic type enclosed in angle brackets (**<>**). This allows for the specification of the type of elements contained within the list.

CatScript Types

CatScript is statically typed, with a small type system as follows

- **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

Section 5: UML.



The sequence diagram above is a UML representation that serves as a valuable tool, illustrating the flow of control and interactions throughout the execution of a print statement within the CatScript Compiler. The sequence initiates when encountering a print statement within the CatScript Compiler. The program evaluates the expression component within the print statement, recursively descending through the grammar to resolve any nested expressions. Then, when the expression is evaluated, the program executes the print statement, which outputs the result. Furthermore, the execution process may initiate more recursive calls within the evaluation process as there may be nested expressions, such as our example being an equality expression, having to recursively call a comparative expression, additive expression, and finally to a primary expression in order to return the integer Literal two through the recursion to the equality expression and is then repeated for the integer three. The result of the executed equality expression is given a false as two does not equal three, returning a false boolean literal to the print statement executing and recursively returning to the program's start. The UML Diagram plays a crucial role in better depicting the essential interactions when executing a print statement in a simple to-observe form. This ensures the sequence diagram remains concise and understandable, emphasizing the critical steps of evaluating the expression and executing the print statement.

Section 6: Design trade-offs

For our Compiler project, we focused on a Recursive descent design rather than a parser generator. Recursive descent adequately fulfills our requirements and more clearly reflects grammar's inherent recursive nature.

Let us look at what each does to understand the reason for the decision to use Recursive descent over Parser Generator. A parser generator consists of lexical grammar using REGEX (regular expressions) and detailed grammar in EBNF (Extended Backus-Naur Form). It takes rules as input to create a parser, similar to building an Abstract Syntax Tree. While this method requires less code and infrastructure, it can be harder to understand and thus takes longer to implement.

While Recursive descent is a top-down approach to parsing, there is an existing method for each output in grammar. Assigning each production a corresponding method is significant because it enhances comprehension of the recursive nature of parsers. Additionally, this technique is commonly employed in the industry for tokenizing source code.

Despite requiring more handwritten code, opting for a handwritten recursive descent parser demanded less effort for beginners in the field, making it quicker to develop and offering a chance to enhance understanding.

Section 7: Software development life cycle model

For our Compiler project, we used the Test Driven Development model, where before programming, tests are created for each step in the process to be run as the program is created and takes shape. In the beginning, we were given a suite of tests built by our professor Carson Gross, separated into individual sections such as Tokenizing, parsing, Evaluation, Execution, and even Bytecode as we progressed through the semester. When looking at Test Driven Development, there are many great and helpful ways the model broke down the development process into simple, easily understood steps. Instead of trudging through understanding and creating the concepts needed to write the needed code, the model helped us understand and write better, faster, and more reliable code, making it easier to spot issues that may arise. The TDD model gives us a better understanding of what our code should be accomplishing and how to accomplish it while allowing us to keep track of our progress by viewing how many of the needed tests pass, giving confidence in our work. The only issue with the Test Driven Development model is if additions in the future are needed as the test format is very rigid, and the Test Driven development model needs more creativity when moving forward. Overall, I enjoyed the program and development of our Compiler project and how the Test Driven Development model put us under a significant number of constraints and taught us the critical importance of meeting all specifications.