Compilers Capstone MSU CSCI-468

Megan Steinmasel

Spring 2024

Tester: Brianna Clark

Section One: Program

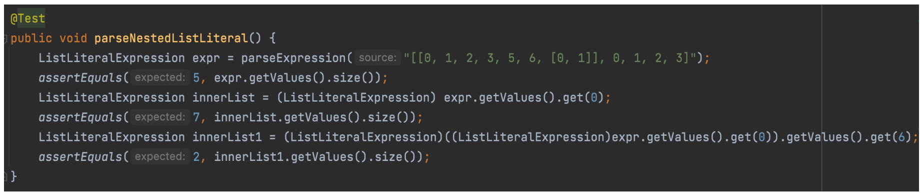
- The complete Compilers Capstone project is located here: https://github.com/megansteinmasel/csci-468-spring2024-private/tree/main
- The zip file, source.zip, of the final /src directory is located here: capstone/portfolio/source.zip

Section Two: Teamwork

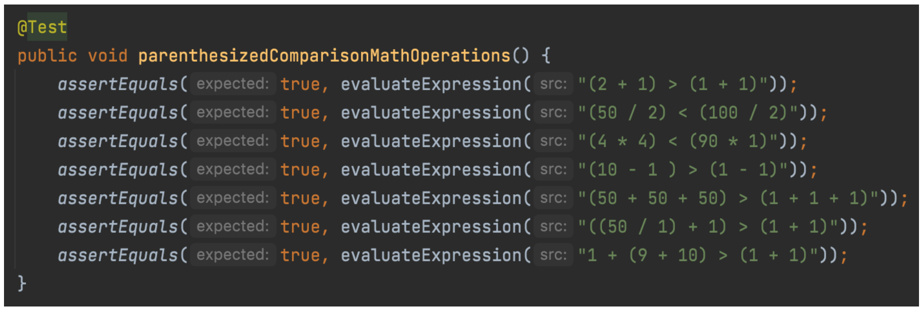
The Catscript Compiler progresses through various stages: tokenization, parsing, evaluation, bytecode generation, partner testing, and documentation creation. The development of tokenization, parsing, evaluation, and bytecode generation was done by the primary developer, Megan Steinmasel. Upon the completion of these four stages, Brianna Clark initiated partner testing and documentation creation. Brianna Clark dedicated approximately 15 hours to the partner testing and documentation creation phases. As the primary developer, I, Megan Steinmasel, contributed a total of 110 hours to this project. This brings the total estimated hours of work to be around 125 hours.

- Primary Developer: Megan Steinmasel
  - Estimated hours: 110 hours
- Tester: Brianna Clark
  - Estimated hours: 15 hours

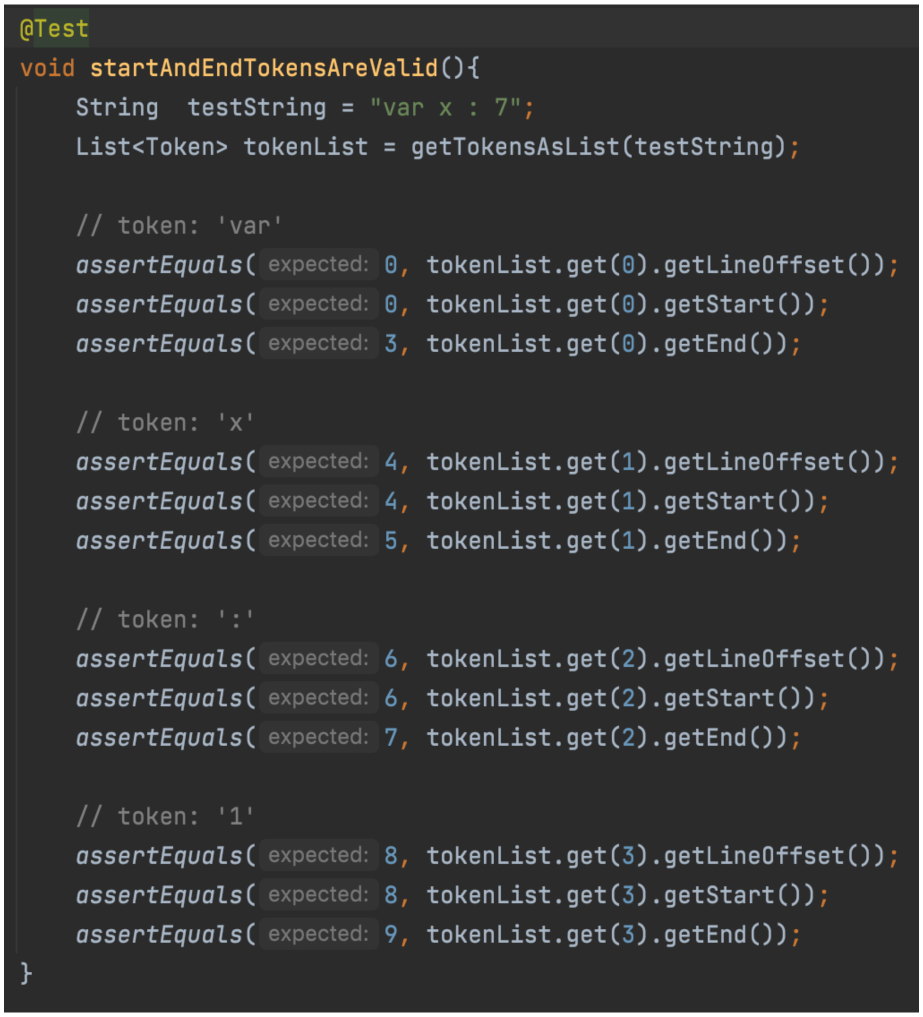
The partner testing can be found in the PartnerTest.java class. The path to this class is src/test/java/edu/montana/csci/csci468/demo/PartnerTest.java. The three successfully passing tests in this class are parseNestedListLiteral(), parenthesizedComparisonMathOperations(), and startAndEndTokensAreValid().



```
@Test
public void parseNestedListLiteral() {
    ListLiteralExpression expr = parseExpression( source: "[[0, 1, 2, 3, 5, 6, [0, 1]], 0, 1, 2, 3]");
    assertEquals( expected: 5, expr.getValues().size());
    ListLiteralExpression innerList = (ListLiteralExpression) expr.getValues().get(0);
    assertEquals( expected: 7, innerList.getValues().size());
    ListLiteralExpression innerList1 = (ListLiteralExpression)((ListLiteralExpression)expr.getValues().get(0)).getValues().get(6);
    assertEquals( expected: 2, innerList1.getValues().size());
}
```



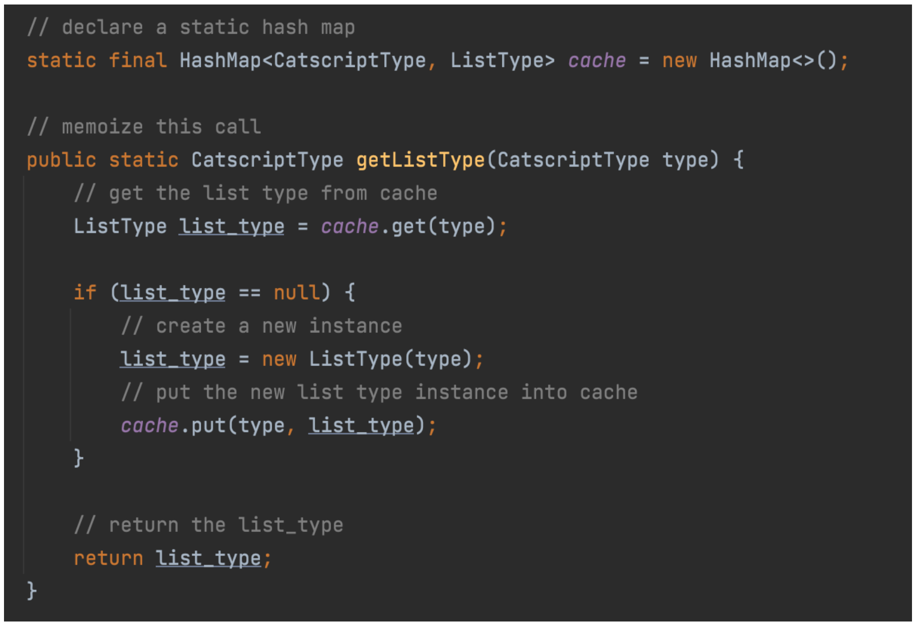
```java
@Test
public void parenthesizedComparisonMathOperations() {
    assertEquals( expected: true, evaluateExpression( src: "(2 + 1) > (1 + 1)"));
    assertEquals( expected: true, evaluateExpression( src: "(50 / 2) < (100 / 2)"));
    assertEquals( expected: true, evaluateExpression( src: "(4 * 4) < (90 * 1)"));
    assertEquals( expected: true, evaluateExpression( src: "(10 - 1 ) > (1 - 1)"));
    assertEquals( expected: true, evaluateExpression( src: "(50 + 50 + 50) > (1 + 1 + 1)"));
    assertEquals( expected: true, evaluateExpression( src: "((50 / 1) + 1) > (1 + 1)"));
    assertEquals( expected: true, evaluateExpression( src: "1 + (9 + 10) > (1 + 1)"));
}
```



```java
@Test
void startAndEndTokensAreValid(){
    String  testString = "var x : 7";
    List<Token> tokenList = getTokensAsList(testString);

    // token: 'var'
    assertEquals( expected: 0, tokenList.get(0).getLineOffset());
    assertEquals( expected: 0, tokenList.get(0).getStart());
    assertEquals( expected: 3, tokenList.get(0).getEnd());

    // token: 'x'
    assertEquals( expected: 4, tokenList.get(1).getLineOffset());
    assertEquals( expected: 4, tokenList.get(1).getStart());
    assertEquals( expected: 5, tokenList.get(1).getEnd());

    // token: ':'
    assertEquals( expected: 6, tokenList.get(2).getLineOffset());
    assertEquals( expected: 6, tokenList.get(2).getStart());
    assertEquals( expected: 7, tokenList.get(2).getEnd());

    // token: '1'
    assertEquals( expected: 8, tokenList.get(3).getLineOffset());
    assertEquals( expected: 8, tokenList.get(3).getStart());
    assertEquals( expected: 9, tokenList.get(3).getEnd());
}
```

Section Three: Design Pattern

A crucial design pattern employed within Catscript is Memoization. Memoization is a technique used to optimize the performance of functions by caching the results of expensive function calls and reusing them when the same inputs occur again.



```java
// declare a static hash map
static final HashMap<CatscriptType, ListType> cache = new HashMap<>();

// memoize this call
public static CatscriptType getListType(CatscriptType type) {
    // get the list type from cache
    ListType list_type = cache.get(type);

    if (list_type == null) {
        // create a new instance
        list_type = new ListType(type);
        // put the new list type instance into cache
        cache.put(type, list_type);
    }

    // return the list_type
    return list_type;
}
```

The location of the memorization design pattern is in the getListType method in the CatscriptType.java class. The path to the CatscriptType.java class is src/main/java/edu/montana/csci/csci468/parser/CatscriptType.java. The memoization design pattern in the getListType method ensures that for each CatscriptType, only one ListType is created. The process unfolds as follows:

1. When the ‘getListType’ function is called with a parameter ‘type’, the function first checks if the result for that particular ‘type’ is already stored in the cache.
2. If the result is found, the function returns the cached result directly, avoiding the expensive computation.
3. If the result is not found in the cache, the function computes the result.
4. The computed result is then stored in the cache for future use.
5. Finally, the function returns the computed result.

Section Four: Technical Writing

Catscript Guide

- This section is used to create a guide for Catscript, to satisfy Capstone Requirement Four.

Introduction

- Catscript is a lightweight and versatile scripting language designed for simplicity and efficiency. Below, we delve into its key features and syntax to help you grasp the fundamentals of Catscript programming. The features that we will be going over in the Catscript guide cover features such as additive expressions, comparative expressions, for-loops, if statements, types, comments, print statements, variable statements, math operations, comparison, equality, unary expressions, and function definitions.

Features

Additive Expressions

- Additive expressions in Catscript facilitate basic arithmetic operations like addition (+) and subtraction (-).

- These expressions serve as the building blocks for mathematical computations and data manipulation tasks.

- Example:

```
var total = 10 + 5; // Assigns the result of 10 + 5 to
the variable ‘total’

var result = 10 - 5; // Assigns the result of 10 - 5 to
the variable ‘result’
```

Comparison Expressions

- Comparison expressions in Catscript enable comparison operations between values, yielding boolean results.

- These boolean results indicate whether the comparison holds true or false.

- Comparison operators encompass less than (<), less than or equal to (<=), greater than (>), and greater than or equal to (>=).

- Example:

```
var isGreaterThan = (10 > 1); // Assigns ‘true’ to
‘isGreaterThan’

var isLessThan = (1 < 10); // Assigns ‘true’ to ‘isLessThan’

var isGreaterThanOrEqualTo = (10 >= 1); // Assigns ‘true’ to
‘isGreaterThanOrEqualTo’

var isLessThanOrEqualTo = (1 <= 10); // Assigns ‘true’ to
‘isLessThanOrEqualTo’
```

For-Loops

- The for-loop in Catscript offers a convenient method for iterating over collections or executing code a specified number of times.

- This loop structure enhances code readability and efficiency, particularly when dealing with repetitive tasks or data processing operations.

- Catscript's for-loop syntax closely resembles that of established languages like Java and Python.

- In the provided example, the for-loop traverses and prints each element within the ‘list’ variable.

- Example:

```
var list = [1, 2, 3, 4];
```

```
for (i in list) {
    print(i);
}

// Prints each element in ‘list’
```

If Statements

- Catscript’s if statement facilitates conditional execution of code blocks based on the evaluation of an expression.

- It begins with the ‘if’ keyword followed by an expression in parentheses, evaluating whether the condition is true. If the condition is met, the code within the following curly braces executes. Optionally, ‘else if’ clauses can be added, each with its own expression and a corresponding block of code to execute if its condition evaluates to true. Finally, an ‘else’ clause can be included to specify a block of code to execute if none of the previous conditions are met.

- The general layout of the if statement is shown below.

```
if (expression) {
    \\ statement
 } else if (expression){
    \\ statement
 } else{
    \\ statement
 }
```

- Example:

```
var num = 20;

if (num < 25) {
    print("Number is less than 25");
} else {
    print("Number is greater than or equal to 25");
}

// Prints ‘Number is less than 25’
```

Types

- Catscript is a statically typed programming language supporting types such as integers, strings, booleans, lists, null, and objects.

- The Catscript type system is shown below.
  - int: a 32-bit integer
  - string: a java-style string
  - bool: a boolean value
  - list: a list of values with the type 'x'
  - null: the null type
  - object: any type of value

- Example:

```
var age: int = 25; // Declares an integer variable 'age'
with value 25

var name: string = "Alice"; // Declares a string variable
'name' with value 'Alice'
```

- Catscript also has one complex type, the list type. You can declare a list of a given type with 'list'.

- Example declarations of the list type are shown below.
  - list: list of integers
  - list <object >: list of objects
  - list <list <int >>: a list of lists of integers

Comments

- Comments in Catscript aid code clarity and documentation without affecting functionality.

- Single-line and multi-line comments are supported.

- Single-line comments, denoted by //, are perfect for adding brief explanations or notes to specific lines of code. Meanwhile, multiple-line comments, enclosed within /* */, provide the flexibility to include more extensive descriptions, and comments spanning multiple lines.

- We can see how both comment types work below.
- Example:

```
// Single-lined comment

/* Multiple-lined comment */
```

Print Statements

- Catscript's print statements facilitate outputting data to the console, useful for debugging and interaction.
- You can also perform concatenation with print statements as seen below.
- Example:

```
print("Hello World"); // Prints ‘Hello World’

var printNum = 1;
print(printNum); // Prints ‘1’

var printStr = "Student ID: ";
print(printStr + 10345); // Prints ‘Student ID: 10345’ with
concatenation
```

Variable Statements

- The ‘var’ statement in Catscript is used for the declaration of variables. We are able to do this with multiple types.
- Variables can be declared, defined, and then redefined throughout the Catscript code.
- Example:

```
var greeting = "Hello"; // Declares variable ‘greeting’ with
value ‘Hello’

var x = 30; // Declares variable ‘x’ with value ‘30’
```

Math Operations

- Addition, subtraction, multiplication, and division are all supported by Catscript.
- For these operations we use the +, -, *, and / symbols, respectively.
- Example:

```
print(1 + 10); // Prints 11

print(10 - 1); // Prints 9

print(20 / 1); // Prints 20

print(20 * 1); // Prints 20
```

Equality

- Catscript provides operators == and != to assess equality and inequality between values.
- These expressions provide a means for assessing the equivalence of data elements, aiding in decision-making and logical operations.
- Example:

```
print(10 == 10); // Prints true

print(10 != 0); // Prints true
```

Unary Expressions

- Catscript supports unary expressions that can be evaluated, as well as operators for negating values.
- These expressions support the unary minus (-) for numeric values and the ‘not’ keyword for boolean values.
- Example:

```
var x = 1;

var y = 10;

print(x + y);  // Prints 11

print(x + -y); // Prints -9
```

Function Definitions

- Developers can define functions in Catscript using the ‘function’ keyword followed by the function name and parentheses containing any parameters.

- Function definitions in Catscript enable developers to encapsulate reusable blocks of code, promoting modularity and code organization.

- In the example below, the ‘greet’ function is defined to take a single parameter name and print a personalized greeting message.

- The function is then invoked with the argument ‘Alice’, resulting in the message ‘Hello, Alice!’ being printed to the console.

- Example:

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

greet("Alice"); // Invokes the ‘greet’ function with the argument
‘Alice’
```

Catscript Grammar

- Below is the Catscript Grammar, a set of rules and syntax guidelines utilized to interpret and comprehend the Catscript language.

```
catscript_program = { program_statement };


program_statement = statement |
                    function_declaration;
```

```
statement = for_statement |
           if_statement |
           print_statement |
           variable_statement |
           assignment_statement |
           function_call_statement;


for_statement = 'for', '(', IDENTIFIER, 'in', expression ')',
               '{', { statement }, '}';


if_statement = 'if', '(', expression, ')', '{',
                   { statement },
              '}' [ 'else', ( if_statement | '{', { statement }, '}' ) ];


print_statement = 'print', '(', expression, ')'


variable_statement = 'var', IDENTIFIER,
    [':', type_expression, ] '=', expression;


function_call_statement = function_call;


assignment_statement = IDENTIFIER, '=', expression;


function_declaration = 'function', IDENTIFIER, '(', parameter_list, ')' +
                      [ ':' + type_expression ], '{',  {
                      function_body_statement },  '}';


function_body_statement = statement |
                         return_statement;


parameter_list = [ parameter, {',' parameter } ];


parameter = IDENTIFIER [ , ':', type_expression ];
```

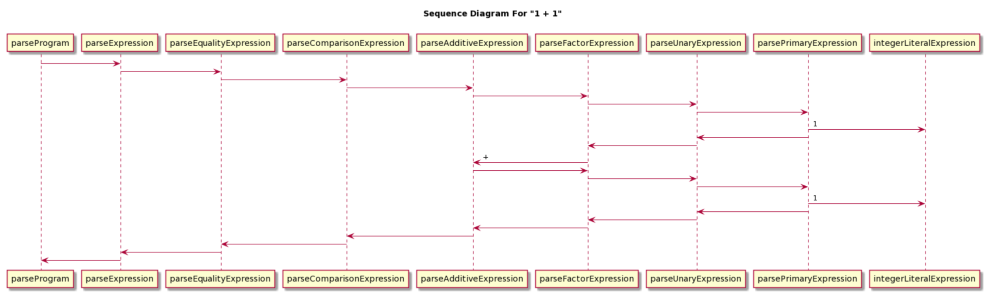
```
return_statement = 'return' [, expression];


expression = equality_expression;


equality_expression = comparison_expression
{ ("!=" | "==") comparison_expression };


comparison_expression = additive_expression
{ (">" | ">=" | "<" | "<=" ) additive_expression };


additive_expression = factor_expression
{ ("+" | "-" )factor_expression };


factor_expression = unary_expression
{ ("/" | "*" )unary_expression };


unary_expression = ( "not" | "-" )
unary_expression | primary_expression;


primary_expression = IDENTIFIER | STRING | INTEGER | "true"
| "false" | "null"| list_literal | function_call | "(",
expression, ")"


list_literal = '[', expression,  { ',', expression } ']';


function_call = IDENTIFIER, '(', argument_list , ')'


argument_list = [ expression , { ',' , expression } ]


type_expression = 'int' | 'string' | 'bool' | 'object' |
'list' [, '<' , type_expression, '>']
```

Conclusion

- Catscript's simplicity and versatility make it a valuable tool for various development tasks. By mastering its features and syntax, you can streamline your development process and create efficient scripts tailored to your needs.

Section Five: UML

There was no need for UML in this project since the overall design had been predetermined by the professor. Given this, I included a sequence diagram that illustrates the step-by-step process of parsing the expression '1 + 1'. The parsing process begins with the high-level task of parseProgram, which initiates the parsing process. Subsequently, the process drills down into specific parsing steps, including parseExpression, parseEqualityExpression, parseComparisonExpression, parseAdditiveExpression, parseFactorExpression, parseUnaryExpression, and parsePrimaryExpression. The diagram then identifies '1' as an integer literal, '+' as an additive expression, and '1' as another integer literal. Through this approach, the parser effectively breaks down the '1 + 1' expression into manageable components.



Section Six: Design Trade-Offs

When comparing recursive descent parsers and parser generators, several significant tradeoffs come into play. Recursive descent parsers, by closely resembling the grammar they parse, provide exceptional ease of use and debugging capabilities. Their structure mirrors the grammar, making them relatively straightforward for developers to understand and troubleshoot. Additionally, recursive descent parsers offer considerable flexibility in handling complex grammars. However, this flexibility may result in inefficiencies when parsing large grammars due to recursion. On the other hand, parser generators excel in optimizing performance, making them ideal for efficiently processing extensive datasets. Yet, this efficiency often comes at the expense of flexibility, as they may struggle to adapt to the intricacies of complex grammars. Furthermore, error handling in parser generators can pose challenges, whereas recursive descent parsers empower developers with direct control over parsing logic, leading to more straightforward error management. Ultimately, the choice between these approaches depends on the specific requirements of the parsing task, balancing considerations such as ease of use, flexibility, performance, and error handling.

Learning recursive descent parsing is straightforward due to its intuitive nature. The direct mapping of grammar rules to parsing functions simplifies the learning process, making it easier for students to grasp parsing concepts and apply them effectively in practice. Therefore, recursive descent is often considered the best approach for beginners in compiler design due to its simplicity and direct applicability.

Section Seven: Software Development Life Cycle Model

Catscript underwent development through a test-driven approach, where progress was determined by passing various sections of unit tests. The development process unfolded in four stages: starting with tokenization, followed by parsing, evaluation, and bytecode generation. Advancement through these stages depended on successfully completing the associated tests in a sequential manner, with each stage building upon the achievements of the previous one.

Test-driven development acts like a roadmap for developers, helping them focus on what needs attention in a project. By setting clear goals with test cases, test-driven development guides developers to tackle tasks step by step, making them more manageable, especially in big projects. This development method keeps teams on track, even when dealing with complex projects, by breaking down the work into smaller and achievable parts. In essence, test-driven development streamlines development efforts, ensuring a structured approach that leads to more successful outcomes.