

Senior Team Portfolio

**Montana State University
Computer Science Department**

**Andrew Kattine-Grimsley
Samuel Mocabee**

**CSCI 468 - Compilers
Spring 2023**

Section 1: Program

See [source.zip](#) for the project code.

Section 2: Teamwork

Throughout the development of CatScript, my partner and I contributed our skills and expertise to the project. I was responsible for the coding aspects, which took approximately 200 hours of work. This included implementing the language's tokenizer, parsing algorithms, evaluator, and Java bytecode compiler. As the primary developer, I had to ensure that the language provided the necessary features and functionality to meet the project's objectives.

My partner, on the other hand, focused on the crucial aspect of technical documentation and writing test cases. They meticulously prepared comprehensive documentation detailing the language's internal mechanisms, features, and usage. This documentation not only served as an invaluable resource during the development process but also provided clear guidance for future users of the language. Their efforts in creating high-quality documentation were essential in making CatScript accessible and user-friendly.

In addition to their documentation work, my partner also contributed to the project by authoring three critical test cases, which played a significant role in ensuring the functionality and correctness of CatScript. These tests covered various aspects of the language, such as concatenation and expression evaluation, function definition and list search, as well as function definition with if-else, return, and variable handling. Their test cases allowed us to identify and resolve any issues, further improving the language's reliability and performance.

The collaboration between my partner and me was essential in the successful development of CatScript. By dividing the tasks according to our respective skills, we were able to focus on the areas where we could contribute the most. This division of labor helped ensure that the final product was robust, efficient, and easy to use.

In conclusion, our teamwork and dedication were instrumental in creating CatScript. My coding expertise, combined with my partner's attention to detail in documentation and testing, allowed us to develop a language that is both functional and user-friendly, providing a valuable educational tool for the both of us. The process of implementing a compiler from scratch was a rewarding learning experience that deepened our understanding of programming languages, parsing techniques, and the intricacies of compiler construction. This hands-on approach not only enriched our knowledge but also honed our problem-solving and debugging skills. Through our combined efforts, we have crafted a programming language that showcases the value of a strong partnership and the benefits of implementing a compiler from scratch.

The specific tests they wrote were:

concatinationAndExpressionEvaluationTest() -

This test verified the accurate behavior of string concatenation and arithmetic expression evaluation in CatScript.

```
@Test
void concatinationAndExpressionEvaluationTest() {
    String input = "\"Hello\" + \" World\" + \" 2023! \" +
    \"null\"\\n";
    String expectedOutput = "Hello World 2023! null\\n";
    assertEquals(expectedOutput, executeProgram(input));

    input = "(279-(8 * 12))+14/2\\n";
    expectedOutput = "190\\n";
    assertEquals(expectedOutput, executeProgram(input));
}
```

functionDefinitionListSearchTest() -

This test examined the proper implementation of function definitions, list search using for loops, and conditional statements (if-else) in the language.

```
@Test
void functionDefinitionListSearchTest() {
    String input = "function foo(x : int) : int {\\n" +
        "for(i in [1, 2, 3, 7, 4]){\" +
        \"if(i!=x){}\\n \" +
        \"else{print(\"Yes!\")}\" +
        \"return 1}\\n\" +
        \"var k = 7\" +
        \"print(\"Is \" + k + \" contained in the list?\")\" +
        \"foo(k)\";

    String expectedOutput = "Is 7 contained in the list?\\nYes!\\n";
    assertEquals(expectedOutput, executeProgram(input));
}
```

functionDefinitionIfElseReturnAndVar() -

This test ensured that CatScript could handle function definitions with if-else statements, return values, and variable manipulation.

```
@Test
void functionDefinitionIfElseReturnAndVar() {
    String input = "function foo(x : int) : int {\n" +
        "if(x >= 0){print(\"is positive.\")}\n" +
        "else{print(\"is negative.\")}\n" +
        "return 1" +
        "}\n" +
        "var x = -3" +
        "print(x)" +
        "foo(x)";

    String expectedOutput = "-3\nis negative.\n";
    assertEquals(expectedOutput, executeProgram(input));

    input = "function foo(x : int) : int {\n" +
        "return x" +
        "}\n" +
        "var x = 12" +
        "print(foo(x))";

    expectedOutput = "12\n";
    assertEquals(expectedOutput, executeProgram(input));
}
```

Section 3: Design Pattern

Memoization is a design pattern that involves caching the results of expensive function calls and returning the cached result when the same inputs are encountered again. This technique helps improve the performance of a program by reducing the need for redundant computations, especially when dealing with recursive or iterative processes that involve a significant amount of computation.

In CatScript, memoization was used as a design pattern within the `CatscriptType` class to optimize the creation and retrieval of list types. Here is the implementation of a memoized method in the class:

```
private static final Map<CatscriptType, CatscriptType> LIST_TYPES =
    new HashMap<>();

public static CatscriptType getListType(CatscriptType type) {
    CatscriptType listType = LIST_TYPES.get(type);

    if (listType == null) {
        listType = new ListType(type);
        LIST_TYPES.put(type, listType);
    }

    return listType;
}
```

In this example, a static `HashMap` called `LIST_TYPES` is used to store and cache previously computed list types. When the `getListType` method is called with a specific `CatscriptType`, it first checks whether a list type for the given `CatscriptType` already exists in the cache. If the list type is not present in the cache, a new `ListType` is created, added to the cache, and returned. If the list type is already present in the cache, the method returns the cached instance. This approach prevents the unnecessary creation of multiple instances of the same list type, resulting in reduced memory usage and improved performance.

By employing memoization as a design pattern in CatScript, the language benefits from enhanced efficiency, particularly when working with list types. This optimization technique is a critical component keeping Catscript running fast.

Section 4: Technical Writing

Introduction

The technical documentation for the Catscript language will contain three sections, expressions, statements, and functions. Within each section will be inner details that include a generalized version of different statement and function code labeled to how the compiler reads Catscript code. Following the general form, the document will contain simplistic examples of different statements and function declaration/calls to help ease the process of learning the Catscript programming language.

Expressions

There are several types of expressions offered in the CatScript programming language which follow the standard logical rules for each category. These categories include equality, comparison, additive, factor, and unary expressions. Below you can view the different operators that belong to their respective category.

Description: Included expressions in the Catscript programming language

```
Equality expressions: !=, ==
    //Examples:
    //2 != 1, true.
    //1 == 1, true.
    //2 == 1, false.
    //1 != 1, false.

Comparison expressions: >, >=, <, <=
    //Examples:
    //1 > 2, false. 1 >= 1, true.
    //1 < 2, true. 3 <= 2, false.

Additive expressions: +, -
    //Examples:
    // 1 + 2, 3
    // 3 - 2, 1.

Factor Expressions: /, *
    //Examples:
    //14 / 7, Output: 2
    //3 * 2, Output: 6

Unary Expressions: not, -
    //Examples:
    //-1
    //var x : bool = false
    //not x
    //print(x), Output(true)

List Literals: []
    //Examples:
    //[1,2,3]
    //[]
    //[“Hello”, “World”]
```

Statements

For Loop Description:

For loops requires an identifier followed by the “in” keyword and an expression. Following that is the statement that will be executed however many times required by the parameter described previously. Below is a general example followed by specific examples to follow and compare.

```
General Code:
for_statement = 'for', '(', IDENTIFIER, 'in', expression ')',
'{' , { statement } , '}'
```

```
Example(s)
for(i in [1,2,3]) {
    print(i)
}
Output: 1 2 3
```

If Statement Description:

If statement requires a parameter expression that will check if the conditions to that parameter are met. If it is it will execute the statement inside the if statement. This can be a standalone statement or followed by an else-if which allows the user to check another condition or an else statement that will execute if none of the previous if statement conditions are met. Below is a general example followed by specific examples to follow and compare.

```
General Code:
if_statement = 'if', '(', expression, ')', '{',
{ statement },
'}' [ 'else', ( if_statement | '{', { statement } , '}' ) ]
```

```
Example(s)
var x = 1
if(x = 1) {
    print("Yes")
}
Output: Yes

var x = 0
if(x = 1) {
    print("Yes")
} else {
    print("No")
}
Output: No
```


Print Statement Description:

Print statements will allow users to print expressions or print functions that return values. Used by using the keyword “print” followed by the desired expression or function in parentheses. Below is a general example followed by specific examples to follow and compare.

```
General Code:  
print_statement = 'print', '(', expression, ')'
```

```
Example(s)  
print("Hello World")  
Output: Hello World  
var x = 10  
print(x)  
Output: 10
```

Variable Statement Description:

Variable statement requires the “var” keyword followed by the desired name for the variable or a semicolon with type for the variable. Each difference in the variable declaration will then contain the assignment operator and the data that will be contained by that variable name. Below is a general example followed by specific examples to follow and compare.

```
General Code:  
variable_statement = 'var', IDENTIFIER, [':', type_expression, ] '=', expression
```

```
Example(s)  
var x = "foo"  
var x : string = "Hello World"  
var x : object = ""  
var x : list<int> = [1, 2, 3]
```

Assignment Statement Description:

The assignment statement follows closely to the variable statement but contains no specific type of addition. Below is a general example followed by specific examples to follow and compare.

```
General Code:  
assignment_statement = IDENTIFIER, '=', expression
```

```
Example(s)  
x = 1  
x = "Hello World"  
x = false
```

Return Statement Description:

Return statements follow the print statement documentation but do not require a pair of parentheses containing the expression that will be returned. Below is a general example followed by specific examples to follow and compare.

```
General Code:  
return_statement = 'return' [, expression]
```

```
Example(s)  
function foo(x : int) : int {  
    return x  
}  
print(foo(9))  
Output: 9
```

```
function foo(x : int) : int {  
    return x + 1  
}  
print(foo(9))  
Output: 10
```

Function Declarations

Function Declaration Description:

Function declaration starts with the keyword “function” followed by the function name and the parameters that will be sent to the function. If variables are sent to the function you can state generic variables or additionally add the types for each variable. Then you can add the body of the function that will execute when called. Below is a general example followed by specific examples to follow and compare. Also included is the formatting for parameters for function declaration.

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

```
Example(s)  
function foo(){}  
function x(a, b, c) {}  
function foo(a : object, b : int, c : bool) {}  
function x() {return}  
function x() : int {return 1}
```

General Code:

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

Example(s)

```
()
```

```
(a, b, c)
```

```
(a : int, b : bool)
```

Function Call Description:

Function call statements require the name of the function followed by a pair of parentheses that will contain the argument list of variables that the function requires. Below is a general example followed by specific examples to follow and compare. Also included is the formatting for parameters for function calls.

General Code:

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

Example(s)

```
foo()
```

```
foo(x)
```

```
foo(a : int, b : bool, c : string)
```

General Code:

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

Example(s)

```
()
```

```
(a, b, c)
```

```
(a : int, b : bool)
```

Section 5: UML

The Catscript Equality Sequence Diagram illustrates the process of evaluating an equality expression in the Catscript language. When a user inputs an expression such as "3 == 3" to be evaluated, the process begins with the activation of the CatScript module. The Lexer is then activated, and the CatScript module sends the input to the Lexer to tokenize the expression. After tokenizing, the Lexer sends the tokens back to the CatScript module, and the Lexer is deactivated.

Next, the Parser is activated, and the CatScript module sends the tokens to the Parser to generate a parse tree. The Parser employs several internal functions, such as `parseExpression()`, `parseEqualityExpression()`, and `parseIntegerLiteral()`, to construct the parse tree. Once the parse tree is created, the Parser sends it back to the CatScript module, and the Parser is deactivated.

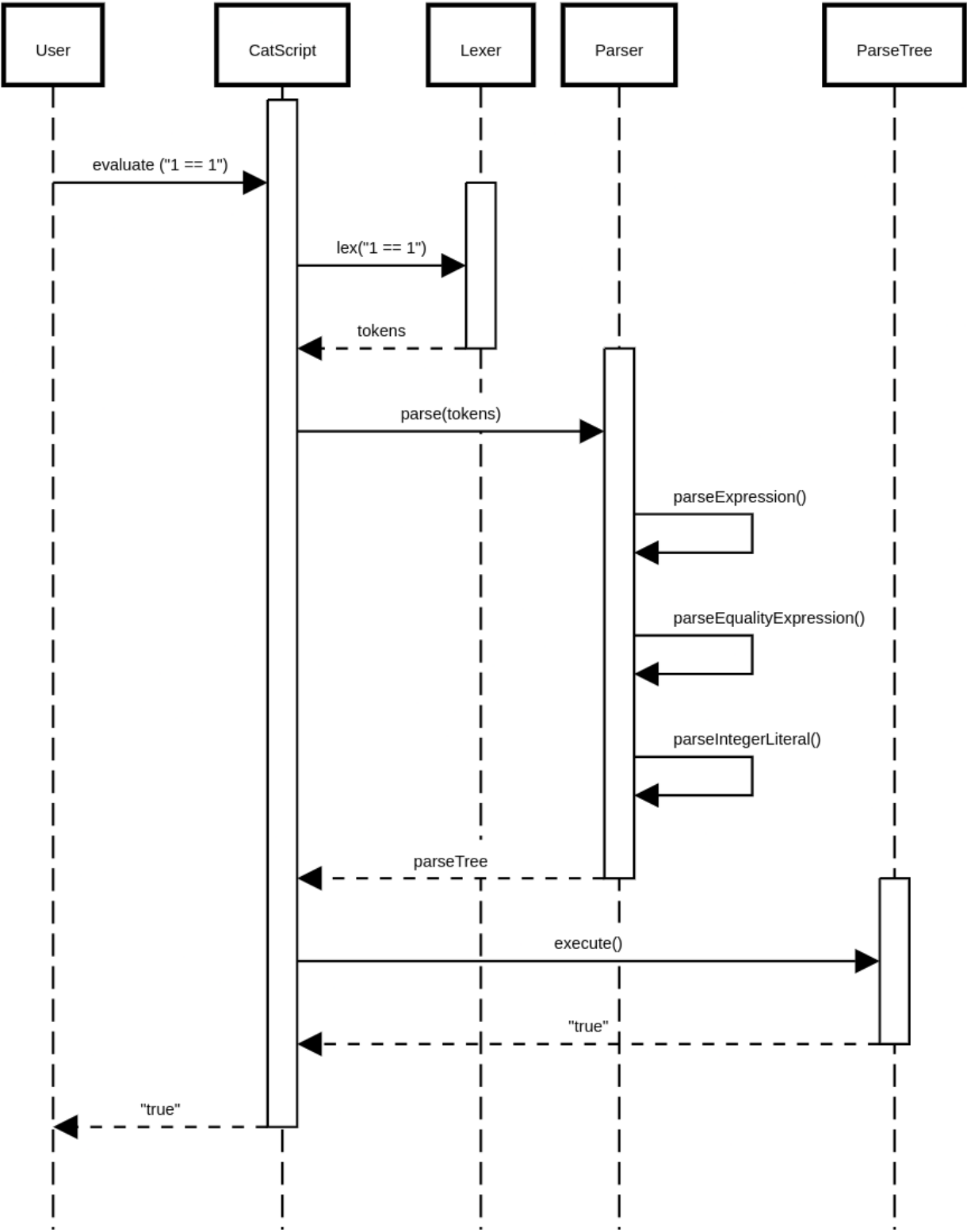
Subsequently, the ParseTree is activated, and the CatScript module sends an `execute()` command to the ParseTree, which evaluates the equality expression and returns the result. The ParseTree is then deactivated, and the CatScript module returns the final result, "true" or "false", to the user before deactivating itself.

The Catscript language parsing process, as described for comparison, addition, and factor expressions, utilizes a recursive descent parsing approach. This technique is a top-down, LL(1) parsing method that begins with the highest-level grammar rule and proceeds to break down the input into its constituent parts, making recursive calls as needed to handle nested expressions and complex structures.

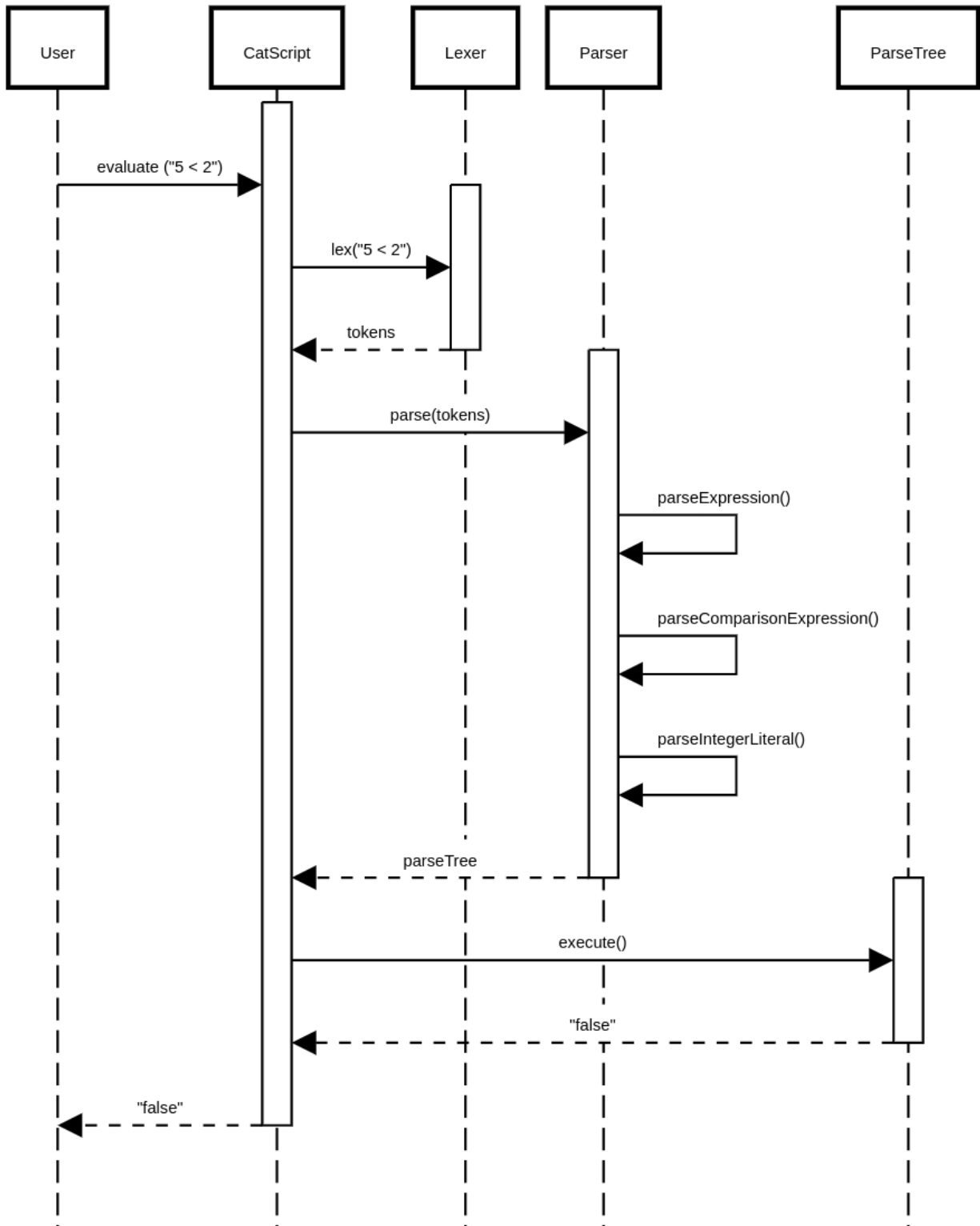
In the case of Catscript, the Parser employs specific internal functions for different types of expressions, such as `parseComparisonExpression()`, `parseAdditiveExpression()`, and `parseFactorExpression()`. These functions represent individual grammar rules and are designed to match and process the input tokens accordingly. When a function encounters a sub-expression or a nested structure, it makes a recursive call to the appropriate function to handle that specific part of the input. This modular, recursive approach enables the Parser to navigate the complexity of the input expression systematically and construct the corresponding parse tree.

The recursive descent parsing approach is well-suited for Catscript, as it allows for the effective handling of various expression types and enables the language to be easily extensible by adding new grammar rules and corresponding parsing functions. This method provides a structured, step-by-step process for parsing and evaluating expressions, ensuring a robust and versatile user experience.

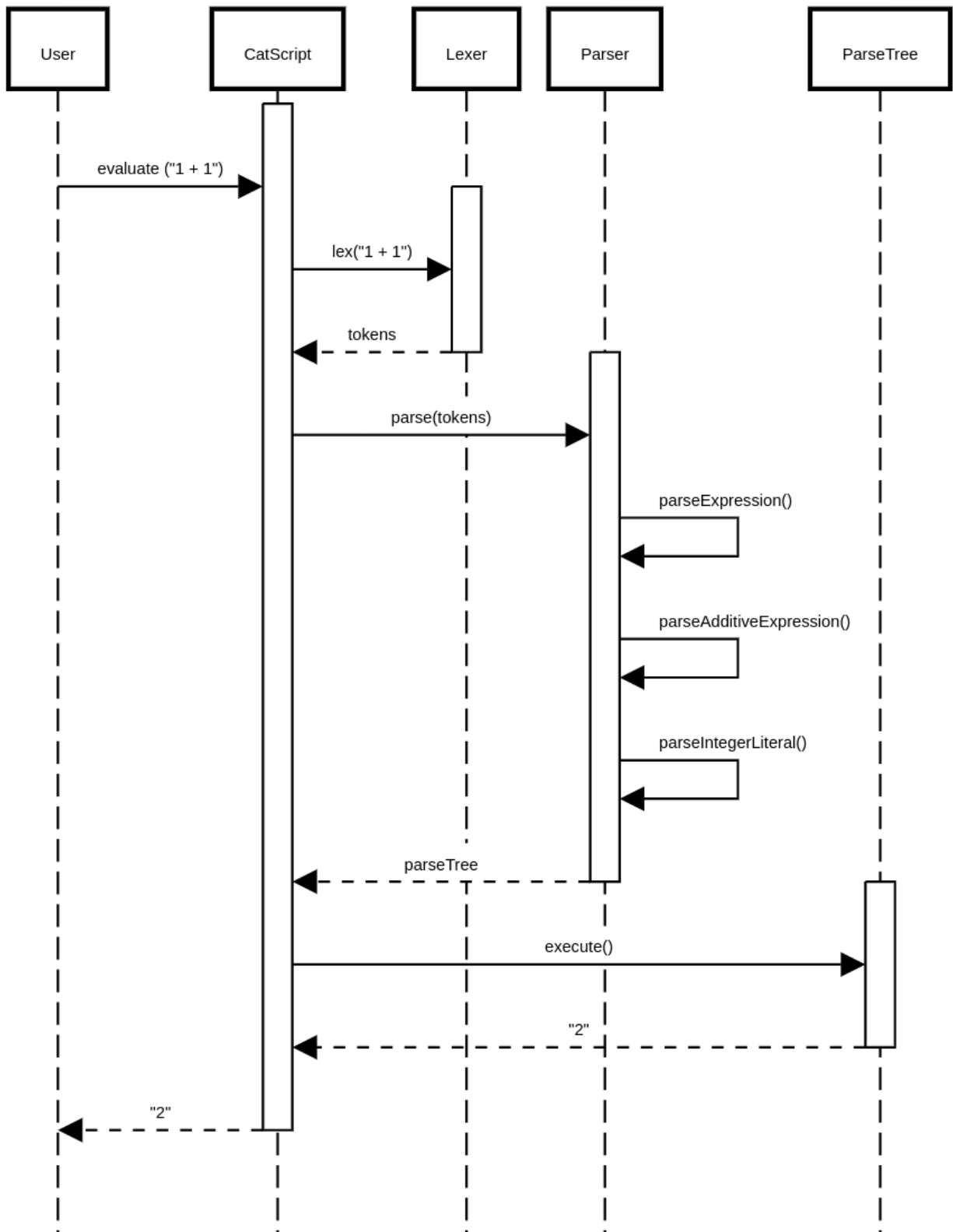
Catscript Equality Sequence Diagram



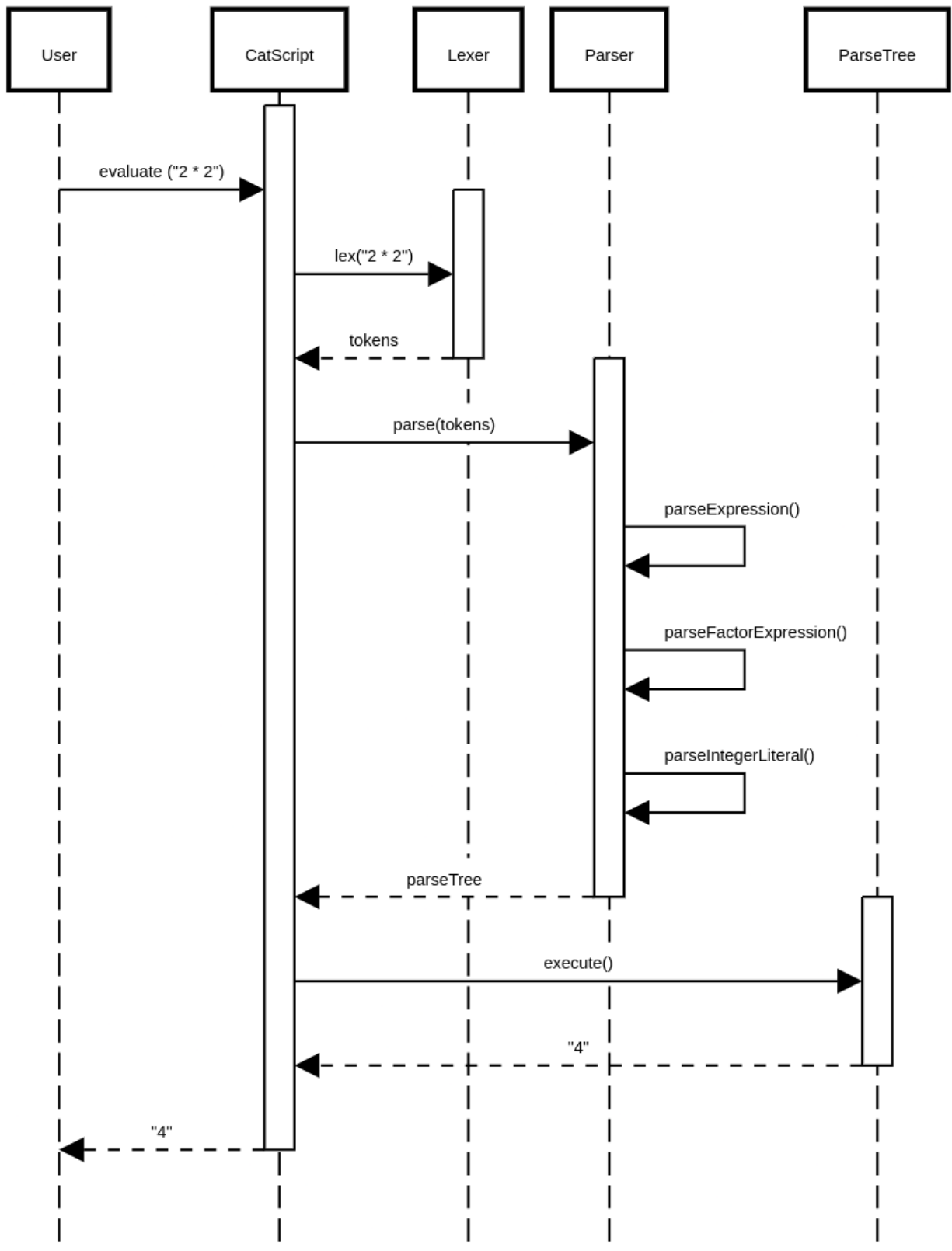
Catscript Comparison Sequence Diagram



Catscript Addition Sequence Diagram



Catscript Factor Sequence Diagram



Section 6: Design Trade-Offs

Introduction

Parsing is a critical component of compiler and interpreter design, responsible for analyzing and transforming source code into a format suitable for further processing. There are various approaches to building parsers, two of which are hand-written recursive descent parsers and parser generators. This essay explores the design trade-offs between these two approaches and provides insights into the factors to consider when making a choice between them.

Hand-Written Recursive Descent Parsers

A hand-written recursive descent parser is a top-down, LL(1) parser that developers create manually by implementing a series of functions corresponding to the grammar rules of the language. The following are the advantages and disadvantages of choosing this approach:

Advantages

Control and flexibility:

One of the most significant benefits of hand-written parsers is the level of control and flexibility they offer. Developers can easily tailor the parsing process to suit their specific needs and implement custom error handling and reporting mechanisms. This can result in a better user experience and more precise error messages.

Readability:

Recursive descent parsers are often more readable than their generated counterparts, as the code closely mirrors the structure of the grammar rules. This makes it easier for developers to understand, maintain, and modify the parser when language changes occur.

No additional tools required:

Since hand-written parsers do not rely on external tools or libraries, they minimize dependencies and potential compatibility issues.

Disadvantages

Time-consuming:

Writing a parser by hand can be labor-intensive, particularly for complex languages with numerous grammar rules. This can significantly increase development time and resource requirements.

Error-prone:

Hand-written parsers are more susceptible to bugs and errors compared to generated parsers, as the developer is responsible for handling all grammar rules and corner cases manually.

Limited support for grammar changes:

Adapting a hand-written parser to accommodate grammar changes in the language can be challenging and may necessitate extensive modifications to the parser code.

Parser Generators

Parser generators, such as ANTLR, Bison, and Yacc, automatically create parsers from formal grammar specifications, eliminating the need for manual coding. These tools offer certain advantages and disadvantages:

Advantages

Automation:

The primary advantage of parser generators is their ability to automate the parser creation process, which saves development time and reduces the potential for human error.

Easier grammar changes:

When language grammar changes, parser generators can automatically update the parser code based on the modified grammar. This simplifies maintenance and ensures that the parser remains in sync with the language specifications.

Optimization:

Parser generators often include optimizations for performance and memory usage. These optimizations result in efficient parsers with minimal manual tuning.

Disadvantages

Less control:

Generated parsers offer less control over the parsing process, and customizing error handling or other aspects of the parser can be more challenging than with a hand-written parser.

Readability:

The code generated by parser generators is often more complex and less intuitive than hand-written code, making it harder to understand and maintain.

Dependency on external tools:

Relying on parser generators introduces dependencies on external tools and libraries. This can create compatibility issues and make the project more challenging to set up and maintain.

Conclusion

The decision to use a hand-written recursive descent parser for Catscript was influenced by multiple factors that aligned with the language's goals and requirements. One such factor is the simplicity of Catscript's grammar, which made it feasible to implement a hand-written parser without excessive development time. The recursive descent approach also allowed for greater control and flexibility, enabling the implementation of customized error handling and reporting to enhance the user experience.

The use of a hand-written parser minimizes dependencies on external tools and libraries, reducing potential compatibility issues and simplifying project setup and maintenance. Catscript's recursive descent parser has a modular design, making it extensible and allowing for the addition of new grammar rules and corresponding parsing functions with relative ease.

Developing a language like Catscript with a recursive descent parser offers valuable learning opportunities that can enhance a programmer's understanding of language design, parsing techniques, and compiler construction. Implementing a hand-written recursive descent parser gives developers hands-on experience in creating a parser from scratch, leading to a deeper understanding of the underlying parsing algorithms and language grammar rules. This knowledge is essential for those aspiring to work with programming languages, compilers, or interpreters in the future.

Constructing a recursive descent parser helps develop problem-solving skills and fosters a strong grasp of recursion, an essential concept in computer science. Working with recursive descent parsers also sharpens debugging skills, as developers must handle corner cases and potential errors while implementing and maintaining the parser.

Learning to create a hand-written recursive descent parser encourages a better appreciation for the structure and organization of programming languages. Developers can explore the intricacies of language design and understand the trade-offs associated with various parsing techniques. This knowledge is invaluable for designing and maintaining custom languages or working with existing ones.

In summary, Catscript's recursive descent parser offers numerous educational benefits and enhances design flexibility. The parser contributes to a deep understanding of parsing algorithms, language grammar rules, recursion, problem-solving, debugging, and language design. The use of recursive descent parsing enables greater control over the parsing process, allowing developers to easily adapt and customize the parser to accommodate changes in the language's grammar or requirements. These skills are highly transferable and advantageous for developers in various computer science disciplines and software development roles, providing an excellent foundation for working with diverse programming languages and paradigms.

Section 7: Software Development Life Cycle Model

Test-Driven Development (TDD) is a software development methodology that emphasizes writing tests before implementing the actual code. This approach prioritizes testing and validation from the very beginning of the development process, ensuring that each component of the software is functioning as intended. As the sole developer working on the CatScript project, I found that adopting TDD provided me with numerous advantages throughout the development process.

One of the most significant benefits of TDD was the improvement in code quality and reliability. By writing tests for every functionality of the parser before implementation, I ensured that the codebase was robust and less prone to errors. This approach not only minimized the occurrence of bugs but also increased the overall reliability of the parser, making it more dependable for its users.

Incorporating TDD into the development process also facilitated a faster feedback loop. By identifying and addressing issues early in the development process, I was able to efficiently iterate on the design and implementation of the parser. This early detection of errors not only reduced the time and effort required for debugging but also helped maintain a steady development pace, enabling me to deliver a functional and reliable parser in a timely manner.

Having tests before implementing the actual code encouraged me to create modular code, which resulted in a more extensible parser. This modularity made it easier for me to understand the code, ensuring that I could quickly add new features or make modifications to the parser as needed. The modular design also allowed for more straightforward future enhancements, such as the addition of new grammar rules or the incorporation of optimizations for better performance.

As the main person working on the project, using tests as a form of documentation proved particularly beneficial. The tests helped me keep track of the expected behavior of each functionality and allowed me to maintain a clear understanding of the project's scope and requirements. This documentation also served as a valuable reference when revisiting various aspects of the code, ensuring that I could quickly identify and address potential issues or areas for improvement.

In conclusion, adopting a Test-Driven Development model for the CatScript project proved advantageous in multiple aspects, including code quality, reliability, and my understanding of the project. This software development life cycle approach ensured

the creation of a robust, efficient, and easily extensible parser. By prioritizing testing and validation throughout the development process, I successfully positioned CatScript as a reliable and versatile language for its users.