

CSCI 468

Spring 2023

Jeremy Heng, Josh Fried

Section 1

The source code for the Catscript Compiler is attached to this project documentation submission in a .zip folder. The technical specifications of the compiler consist of the Catscript grammar, which is as follows:

- `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, '>']

Section 2

Team member 1 was the primary engineer for the Catscript Compiler. They wrote all of the classes, methods, and code needed for the various components of the compiler. In essence, they drove the development of the compiler. Team member 2 served as the testing engineer for the test driven development of the compiler. They provided unit tests to ensure that the compiler had been properly implemented. They also created all of the documentation for the compiler

and the Catscript language. Team member 1 contributed about 80% of the time/effort in the overall project and team member 2 contributed 20% of the time/effort.

Section 3

The **memoization design pattern** was used in the development of the compiler. This pattern is a form of data caching and is used to improve the performance of the compiler. The results of method/function calls are cached so that they can be retrieved again at a later time without having to perform the same, potentially expensive, method/function calls again. For CatscriptType.java in the Catscript Compiler, memoization was used to cache the list type for retrieval in a method called getListType(). This improved the performance and speed of the compiler by reducing the number of method calls needed to retrieve the list type of a given parse element.

Section 4

Introduction

Catscript is a statically typed language, which means that variables are known at compile time. The Catscript language is inspired by Java and therefore has a similar type system and other similar features. Here are some examples of the Catscript language:

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

```
var x = add(1, 2)
```

```
if (x > 5) {  
    print("x is greater than 5")  
}
```

```
    } else if (x < 2) {  
        print("x is less than 2")  
    } else {  
        print("x is less than 5 but greater than 2")  
    }  
}
```

Compiler

The Catscript compiler was written in Java and has four main parts. The first part is the tokenizer that divides the input into different tokens that can be used later by the parser. The second part is the parser that helps the compiler understand the context of each token from the tokenizer to the Catscript language. The parser uses recursive descent to simplify the process. The third part is evaluation which evaluates expressions to derive the correct output. The final part is the bytecode that converts the code into readable bytecode for the Java Virtual Machine.

Type Systems

The Catscript type system takes inspiration from the Java type system. There are many similar variable types between the two, such as int, bool, string, list, null, and void. The int, bool, string, null, and void are all based on Java types. The list variable type is an immutable iterable collection of another variable type. Variables can be declared both implicitly and explicitly. An example of implicit and explicit declaration:

```
var x = 1  
var x : int = 1
```

Features

For loops are an iterable statement that repeatedly executes while true. An example of a for loop:

```
for (x in list) {  
    print(x)
```

```
}
```

If statements are conditional statements that function like those of other contemporary languages. There are also if else and else statements. If else statements act like if statements, but they must come after an if statement and only execute if the above statement is false and if they are true. Else statements must come after an if statement or after an if else statement. Else statements do not take a condition but instead execute if the previous statements are all false. An example of if, else if, and else statements:

```
if (1 < 2) {  
    return(true)  
}  
  
if ( 1 > 2) {  
    return(true)  
} else if (1 > 3) {  
    return(true)  
} else {  
    return(false)  
}
```

Equality expressions can be used in if statements to compare two things. The two equality expressions “!=” and “==” mean not equal and equal, respectively. An example:

```
if (“Hi” != “Hello”) {  
    return(true)  
}  
  
if (“Hi” == “Hi”) {  
    return(true)  
}
```

Comparison expressions can be used in if statements to compare two things. The four comparison expressions ">", ">=", "<", and "<=" mean greater than, greater than or equal, less than, and less than or equal, respectively. An example:

```
if (2 > 1) {  
    return(true)  
}
```

```
if (2 >= 2) {  
    return(true)  
}
```

```
if (1 < 2) {  
    return(true)  
}
```

```
if (1 <= 1) {  
    return(true)  
}
```

A function header contains the keyword function followed by the name of the function, the parameters in parentheses, and the return type. A function body can have anything in the Catscript language inside of it, but must return the correct return type. An example:

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

A function call will have the name of the function and the parameters in parentheses. An example:

```
add(1, 2)
```

The return statement is found inside of function bodies and must return the same type as the declared return type in the function header. An example:

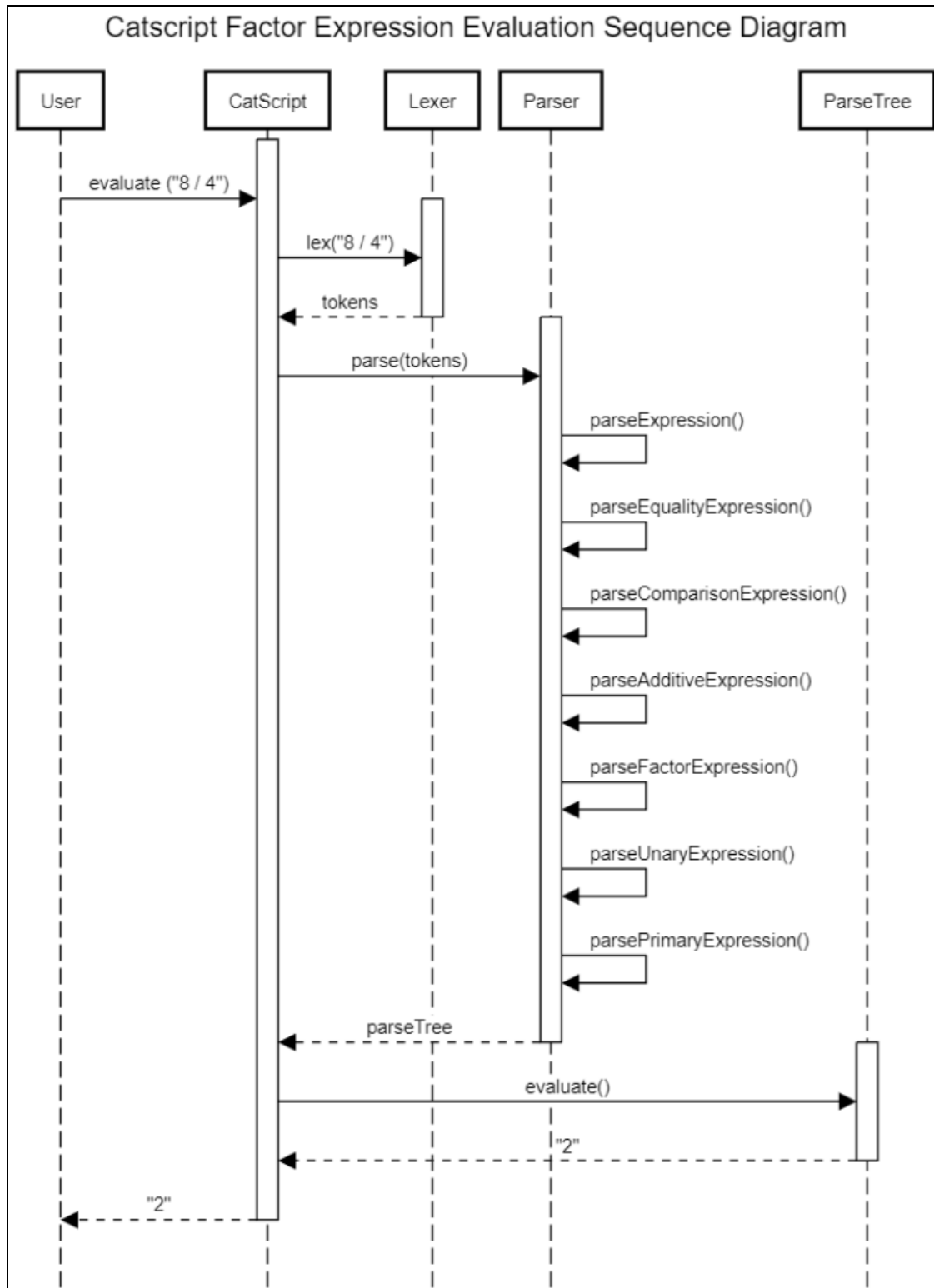
```
function add( a :int, b :int) {  
    c = a + b  
    return(c)  
}
```

The print statement will print an output to a terminal window. An example:

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

Section 5

UML sequence diagrams were used in the development of the compiler to ensure proper implementation of parse element classes and methods. The following sequence diagram was used for development of the evaluation of factor expressions.



Section 6

A design trade-off decision that occurred in the project occurred before implementation. The trade-off was to create the compiler by hand instead of using parser generator tools. The main reason for this decision was that parser generators would have abstracted too much code and implementation details away from the team. This would have resulted in a significant reduction of understanding and learning concerning parsers. Using parser generators would have only resulted in understanding of the specific generators/tools used instead of how a compiler works and how to implement compiler components. Furthermore, the recursive descent technique was used for implementation instead of parser generators because the technique is very intuitive and reinforces understanding of fundamental computer science principles, like recursion. It also led to a greater understanding of how programming languages work and considerations when designing languages. Although abstraction is an important concept in computer science, it can lead to lessened understanding of the concepts or things that are abstracted (which would have been the case for compiler development). There are some inherent issues with code generators that also influenced the design trade-off. Code generators produce a significantly more amount of code compared to writing a compiler by hand. Furthermore, the code is almost unreadable and almost impossible to debug. These issues would have introduced significant challenges when creating the compiler.

Section 7

A test driven development model was used for software development when implementing the compiler. This model helped to drive development and ensure proper implementation of classes and methods. Having tests for each component of the compiler helped define the scope of the work that needed to be accomplished and provided an easy way to view progress being made. Smaller, low-level tests provided an easy way to distinguish if methods were implemented correctly. The larger, high-level tests ensured proper

communication and coordination between the various components and classes. The test driven development really increased production speed as well as quality. Without the development model, testing the compiler would have been a laborious and frustrating process. Test driven development proved to be an effective, easy, and satisfying model.