Capstone Documentation

Theodore Vander Wall, Josh Nelson

~ CSCI 468: Compilers ~

Section 1: Program

Please see the zipped source code & specification in source.zip

Section 2: Teamwork

This project was completed by a team of two members working in separate roles.

Team member 1 (implementor) was responsible for all code implementation across the project checkpoints: tokenizing, expression parsing, statement parsing, semantic analysis, evaluation (tree-walking interpreter), and JVM bytecode compilation. This work comprised the bulk of the project effort, estimated at approximately 85% of total time spent.

Team member 2 (tester/documentor) contributed three tests to validate the implementation, found in `csci468.partner.PartnerTests`, and wrote technical documentation for the CatScript programming language. Their contribution is estimated at approximately 15% of total time spent.

The two team members coordinated via email. Team member 2 sent their tests and documentation as text, which team member 1 integrated into the project. One of the partner tests required a minor fix: a function parameter name conflicted with a global variable name. Which was resolved by renaming the global variable. All three partner tests pass against the final implementation.

Partner tests contributed by team member 2 (Josh):

```
@Test
void StartingTest(){
    assertEquals("10\n", execute("""
                var x = 4 * (4 - 1)
                var y = 30 / 2
                var mod = y % x
                var result = mod + 7
                print(result)
                """));
}

@Test
void SumWorks() {
    assertEquals("100\n", execute("""
                var listOfNumbers = [10, 20, 30, 40]
                function add(list:list<int>):int {
                   var sum = 0
                   for(number in list) {
                       sum = sum + number
                   }
                   return sum
                }
                var result = add(listOfNumbers)
                print(result)
                """));
}

@Test
void FactorialWorks() {
    assertEquals("720\n", execute("""
                var n = 6
                function factorial(start:int):int {
                   var product = 1
                   while(start > 1) {
                       product = product * start
                       start = start - 1
                   }
                   return product
                }
```

```
                var result = factorial(n)
                print(result)
                """));
}
```

Section 3: Design Pattern

We implemented the Memoization Pattern in `csci468.parser.CatscriptType.getListType()`.

Memoization is a design pattern where the result of a function call is cached so that subsequent calls with the same input return the cached value rather than recomputing it. This avoids redundant object creation and ensures reference identity for equivalent types.

Before memoization, every call to `getListType()` created a new `ListType` object:

```
public CatscriptType getListType() {
    return new ListType(this);
}
```

This was problematic because two calls like `CatscriptType.INT.getListType()` would return different `ListType` objects. Since parts of the type system rely on reference equality (`==`) to compare types, creating a new object each time caused type comparisons to fail unexpectedly.

After applying memoization, the result is cached in a field on the `CatscriptType` instance:

```
private ListType listType;

public CatscriptType getListType() {
    if (listType == null) {
        listType = new ListType(this);
    }
    return listType;
}
```

Now, `CatscriptType.INT.getListType()` always returns the same `ListType` instance. This ensures that `==` comparisons work correctly, avoids unnecessary object allocation, and makes the type system behave consistently. The pattern was simple to implement: just a null check and a field. But it solves a real correctness issue in addition to providing a minor performance benefit.

Section 4: Technical Writing/Catscript Guide

CatScript User Guide

CatScript is a small statically typed scripting language where programs execute from top to bottom as a sequence of expressions and statements. A program consists of mainly variable declarations, expressions, control flow constructs, functions, lists, and special values like `null`.

Variable Declaration and Types

Variables are declared using the `var` keyword. They may optionally also include a type annotation such as `int`, `string`, `bool`, or `list<T>`. Variables may also be assigned the special value `null`, which represents the non-existence of a value. A variable can be declared with the type `object` when it needs to store a value whose exact type may vary or is not known at compile time. The object type is a general purpose container that can hold any CatScript value, including integers, strings, booleans, lists, or even null.

```
var x:int = 10
var message:string = "Hello"
var flag:bool = true
var numbers:list<int> = [1, 2, 3]
var y:object = 44
var z:object = "This is of type object"
```

Expressions and Arithmetic

My CatScript has support for arithmetic expressions including addition, subtraction, multiplication, division, and modulus. Expressions are evaluated immediately and can be assigned to variables or used in larger computations.

```
var x:int = 10
x = x + 5
var result = 5 * 4 - x
```

After computation, `result` is `5`

Lists and Indexing

Lists store ordered collections of values. Elements can be accessed using index notation starting at 0.

```
var nums = [10, 20, 30]
var sum = nums[0] + nums[1] + nums[2]
```

Built-in Functions

CatScript provides a single built in function, `print()`, that is always available to the user. `print()`, outputs the evaluated result of an expression to the console and can accept any valid CatScript expression, including integers, strings, booleans, lists, or variables. When executed, the expression is evaluated first, and then the resulting value is displayed.

```
print("Hello, world")
print(10 + 5)
```

```
var x = 44
print(x)
```

After computation, the console displays:

```
Hello, world
15
44
```

Comparisons and Logical Operations

CatScript supports the comparison operators `>`, `<`, `>=`, `<=`, `==`, and `!=`, which produce boolean values. Logical negation using `not` can be used to invert boolean expressions and control program flow.

```
var result = 20
var isGreater:bool = result > 10
var isEqual:bool = result == 20
var isNotGreater:bool = result > 30
print(isGreater)
print(isEqual)
print(isNotGreater)
```

After computation, the console displays:

```
true
true
false
```

Control Flow

Control flow in CatScript includes:

- `if / else` for conditional execution
- `while` loops for repeated execution while a condition is true
- `for` loops for iterating over lists

These constructs determine how/when blocks of code are executed.

```
var value = 5
if (value > 3) {
    print("value is greater than 3")
} else {
    print("value is less than or equal to 3")
}

var i = 0
while (i < 3) {
    print(i)
    i = i + 1
}

var numbers = [1, 2, 3]
var sum = 0
for(n in numbers) {
    sum = sum + n
    print(sum)
}
```

After computation, console displays:

```
value is greater than 3
0
1
2
1
3
6
```

Functions

Functions are declared using the `function` keyword. They may take typed parameters, and optionally specify a return type, and are invoked using standard function call syntax. Functions allow for reuse of logic across a program.

```
function multiply(a:int, b:int):int {
    return a * b
}
var product = multiply(3, 4)
print(product)
product = multiply(5,6)
print(product)
product = multiply(11, 4)
print(product)
```

After computation, the console displays:

```
12
30
44
```

Null Values

The value `null` represents the non existence of data and can be assigned to variables or used in expressions like any other value.

```
var x = null
print(x == null)
print(x)
```

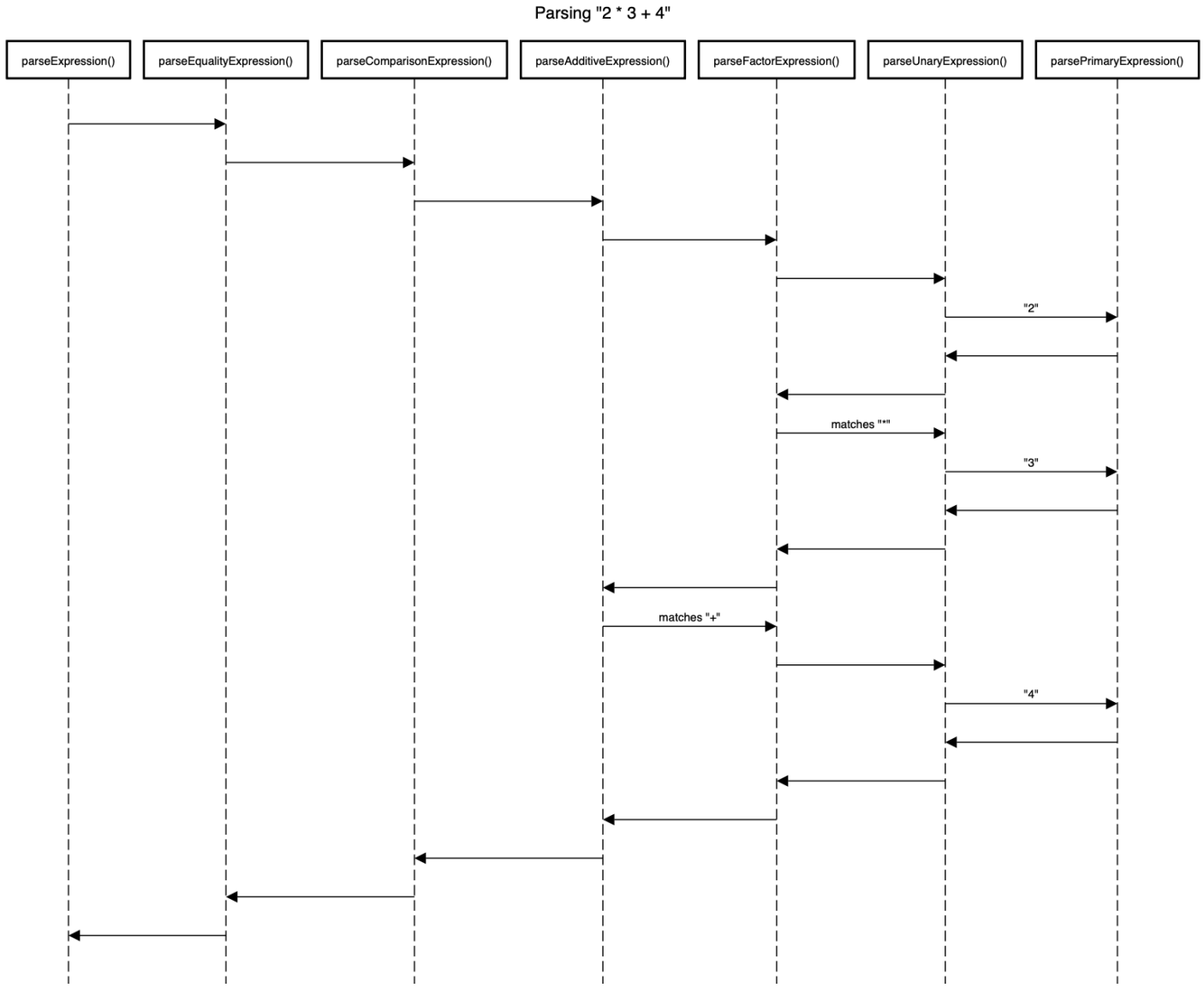
After computation, the console displays:

```
true
null
```

Section 5: UML

Since the CatScript project focused on implementing a recursive descent parser rather than data-oriented design, traditional class diagrams were not applicable. Instead, we use a sequence diagram to illustrate how an expression is parsed through the recursive descent call chain.

The following sequence diagram shows how the expression `2 * 3 + 4` is parsed. The parser descends through the expression hierarchy, first parsing `2` as a primary expression, then matching the `*` operator in `parseFactorExpression`, descending again to parse `3`, returning up to `parseAdditiveExpression` where the `+` is matched, and finally descending once more to parse `4`.



Here is the code from https://sequencediagram.org:

```
title Parsing "2 * 3 + 4"

parseExpression()->parseEqualityExpression():
parseEqualityExpression()->parseComparisonExpression():
parseComparisonExpression()->parseAdditiveExpression():
parseAdditiveExpression()->parseFactorExpression():
```

```
parseFactorExpression()->parseUnaryExpression():
parseUnaryExpression()->parsePrimaryExpression(): "2"

parseUnaryExpression()<-parsePrimaryExpression():
parseFactorExpression()<-parseUnaryExpression():

parseFactorExpression()->parseUnaryExpression(): matches "*"
parseUnaryExpression()->parsePrimaryExpression(): "3"

parseUnaryExpression()<-parsePrimaryExpression():
parseFactorExpression()<-parseUnaryExpression():
parseAdditiveExpression()<-parseFactorExpression():

parseAdditiveExpression()->parseFactorExpression(): matches "+"
parseFactorExpression()->parseUnaryExpression():
parseUnaryExpression()->parsePrimaryExpression(): "4"

parseUnaryExpression()<-parsePrimaryExpression():
parseFactorExpression()<-parseUnaryExpression():
parseAdditiveExpression()<-parseFactorExpression():
parseComparisonExpression()<-parseAdditiveExpression():
parseEqualityExpression()<-parseComparisonExpression():
parseExpression()<-parseEqualityExpression():
```

Section 6: Design Trade-offs

The primary design trade off in this project was the choice between a hand written recursive descent parser and using a parser generator tool like ANTLR.

A **recursive descent parser** is written manually, with one function per grammar rule. Each function examines the current token, decides which production to apply, and calls other parsing functions recursively. This is the approach we used in the CatScript compiler.

A **parser generator** takes a formal grammar specification as input and automatically generates the parsing code. Tools like ANTLR, Yacc, and Bison are parts of this category.

We chose recursive descent for several reasons:

- **Clarity**: Each grammar rule maps directly to a function, making the parser easy to read and understand. The code is the same as the grammar itself, which is good in an educational setting.
- **Debuggability**: When something goes wrong, you can set breakpoints in specific parsing functions and step through the logic. With generated parsers, the generated code is often hard to read and debug.
- **Error reporting**: A hand written parser also gives full control over error messages. We can detect specific error conditions and report meaningful diagnostics, but parser generators often produce generic error messages.
- **No external tooling**: The parser is plain Java code with no build time code generation step, simplifying the build process.

The trade offs of this choice include:

- **More code to write**: Every grammar rule must be implemented manually, which is more work than writing a grammar file for a generator.
- **Risk of bugs**: Manual implementation means manual mistakes. A parser generator can make sure that the generated code correctly implements the grammar.
- **Harder to extend**: Adding new syntax requires modifying multiple parsing functions, where a grammar file change in a parser generator is often simpler.

For a language as small as CatScript, recursive descent was the right choice. The parser is straightforward, the error messages is clear, and the educational value of learning how parsing works is more important than the convenience of a generator. For a larger, more complex language, the balance might more towards using a parser generator to reduce the implementation effort and potential for bugs.

Section 7: Software Development Life Cycle Model

We used Test Driven Development as our life cycle model.

In Test Driven Development (TDD), tests are written before the implementation. We start with a failing test, then write the minimum code to make it pass, and then the refactors if needed. This cycle is often called "red, green, refactor". Because it repeats for each new feature.

In this project, TDD was applied in a specific way: Carson provided a suite of tests for each checkpoint (tokenizing, expression parsing, statement parsing, semantic analysis, evaluation, and bytecode compilation). My job as implementor was to make those tests pass by filling in the implementation. This gave me a clear, measurable target for each part of the project.

The coordination with my partner followed the same model. Josh contributed three additional tests that exercised combinations of CatScript features: arithmetic with modulus, list summation with a function and for loop, and iterative factorial with a while loop. These tests were integrated into the project and all passed against the existing implementation.

What worked well about TDD in this project:

- Having tests up front gave me immediate, good feedback on if the implementation was correct. There was never confusion about what "done" looked like.
- The tests served as a specification. Reading the test cases helped clarified exactly what was expected for edge cases, error conditions, and the type system rules.
- Incremental progress was not hard to track. Going from 50/77 tests passing to 77/77 gave me clear motivation and a sense of progress.

What was challenging about TDD in this project:

- Some tests were difficult to understand by themselves without first understanding the overall architecture. It sometimes meant that I had to read multiple test cases together to understand the right behavior.
- When a lot of tests failed for the same reason, it was not always clear which test to focus on first. Fixing one issue could sometimes reveal that several other failures were the same problem.

In conclusion, TDD was a good fit for this project. The pre written tests gave me a clear roadmap through the project, and the partner contributed tests gave more validation from an different perspective.