Capstone Documentation

Eoghan Perkins, Kyla Andersen

~ CSCI 468: Compilers ~

Section 1: Program

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

Section 2: Teamwork

This project was completed by two team members working in a partner exchange model.

Eoghan Perkins was responsible for the full implementation of the CatScript compiler, including the tokenizer, the recursive descent parser, all expression and statement AST nodes, semantic analysis and type checking through the `verify()` pass, evaluation in `CatscriptRuntime`, bytecode generation and JVM compilation in `ByteCodeGenerator`, and JavaScript transpilation in `JSTranspiler`. The estimated time contribution was about 70% of the total project effort.

Kyla Andersen, Tester/Documentor Partner, contributed three partner tests in `csci468.partner.PartnerTests`, included inline below, that exercise for-loop iteration, recursive functions, and nested conditional branching. Kyla also contributed the CatScript language documentation described in Section 4. The estimated time contribution was about 30% of the total project effort, including the tests and documentation exchange.

Partner Tests (inline)

```
// the following are partner tests for Eoghan:
@Test
void nestedListIndexingWorks() {
    // verifies that indexing into a list of lists works correctly
    assertEquals("2\n", execute("[[1, 2], [3, 4]][0][1]"));
}

@Test
void listTypeInferenceWorksWithStrings() {
    // ensures list<string> is inferred and printed correctly
    assertEquals("[hello, world]\n", execute("var x = [\"hello\", \"world\"]\nprint(x)"
}

@Test
void functionReturnsListElement() {
    // checks interaction between functions, lists, and indexing
    assertEquals("3\n", executeProgram("""
                    function getSecond(list : list<int>) : int {
                        return list[1]
                    }
                    print(getSecond([2, 3, 4]))
            """));
}
```

Section 3: Design Pattern

We implemented the memoization pattern in `csci468.parser.CatscriptType.getListType()`.

Memoization is a simple caching technique: the first time you call a function with a particular argument, you compute the result and store it. Every subsequent call with the same argument just returns the stored value instead of recomputing. Before memoization, every call to `getListType()` would create a brand-new `ListType` object:

```
// Before, a new object every time
public ListType getListType() {
    return new ListType(this);
}
```

After adding memoization, we cache the result in a static `HashMap` and reuse it:

```
// After, cached with computeIfAbsent
static Map<CatscriptType, ListType> cache = new HashMap<>();

public ListType getListType() {
    return cache.computeIfAbsent(this, ListType::new);
}
```

Why Does It Help?

`getListType()` is called frequently during type checking, for example whenever the parser needs the type `list<int>`. Without caching, each call allocates a new object and runs the `ListType` constructor again. With memoization, the same `ListType` instance is reused for the same component type, which preserves object identity for == comparisons, reduces allocation, and keeps lookups fast with `HashMap`. In a compiler that processes many types, this makes the implementation both cleaner and more efficient.

Section 4: Technical Writing/Catscript Guide

Catscript is a simple, beginner-friendly programming language designed to demonstrate core programming concepts such as variables, expressions, control flow, functions, and lists. It is dynamically typed, meaning variables do not require explicit type declarations, and types are determined at runtime.

Catscript programs are executed line-by-line and support both interpretation and compilation to JVM bytecode.

Basic Syntax

Catscript uses a clean and minimal syntax. Statements are written one per line, and blocks of code are enclosed in curly braces `{}`.

Variables

Variables are declared using the `var` keyword. The type is inferred automatically. Variables in Catscript are dynamically typed, meaning their type is determined by the value assigned to them at runtime. Once a variable is assigned a value, it can later be reassigned to a value of a different type.

For example, a variable that initially stores an integer may later store a string. This flexibility makes Catscript easy to use, but it also means type-related errors may occur during execution rather than at compile time.

Data Types

Catscript supports the core types `int` for integer values, `string` for text values, `boolean` for true or false values, `list` for ordered collections, and `null` for the absence of a value.

Example:

```
var a = 5;
var b = "hello";
var c = true;
var d = [1, 2, 3];
```

Expressions

Expressions compute values. Expressions are evaluated at runtime and must follow type compatibility rules. For example, arithmetic operations require integer operands, while string concatenation uses the + operator with string values.

If incompatible types are used in an expression (such as adding an integer to a list), a runtime error may occur.

Arithmetic:

```
var result = (3 + 4) * 2;
```

String concatenation:

```
var message = "Hello " + "World";
```

Comparisons:

```
var check = 5 > 3;
```

Control Flow

Control flow determines the order in which statements are executed in a Catscript program. By default, statements execute from top to bottom, but control flow structures allow programs to make decisions, repeat actions, and iterate over data.

Catscript supports conditional execution using `if` statements, as well as repetition through `while` loops and `for` loops. These constructs rely on boolean expressions to control program behavior.

If Statements

```
if (x > 5) {
  print("x is large");
} else {
  print("x is small");
}
```

While Loops

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

For Loops (List Iteration)

```
var list = [1, 2, 3];
for (i in list) {
  print(i);
}
```

Lists

Lists store multiple values and are indexed starting at 0. Attempting to access an index outside the bounds of the list will result in a runtime error.

Lists in Catscript can store elements of mixed types, but operations on those elements must still respect type rules when used in expressions.

```
var nums = [10, 20, 30];
print(nums[0]); // prints 10
```

Lists can contain mixed types:

```
var mixed = [1, "hello", true];
```

Functions

Functions allow code reuse and can take parameters and return values. Functions define their parameter types and return type explicitly. When a function is called, the provided arguments must match the expected parameter types.

If a return statement is missing in a function that specifies a return type, or if the returned value does not match the declared type, an error will occur.

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

Errors and Runtime Behavior

Catscript performs type checking and evaluation at runtime. Common errors include using incompatible types in expressions, accessing invalid list indices, calling functions with incorrect argument types, and missing or incorrect return values. These errors will typically result in runtime exceptions during program execution.

Complete Example Program

The following program demonstrates basic list iteration and arithmetic in Catscript.

```
var numbers = [1, 2, 3, 4];
var total = 0;

for (n in numbers) {
  total = total + n;
}

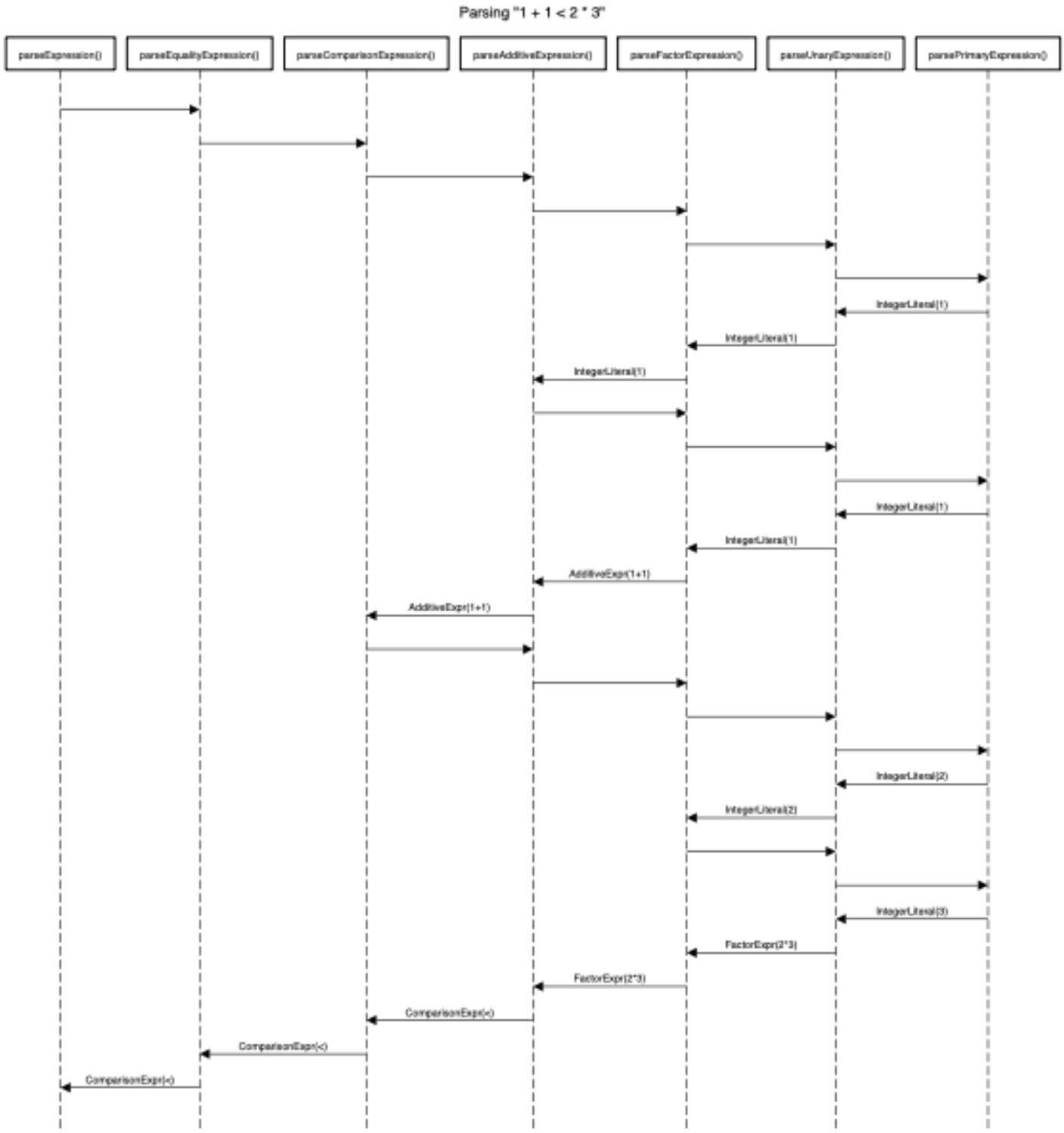
print(total);
```

Program Explanation

This program creates a list of integers and uses a `for` loop to iterate through each element. During each iteration, the value is added to a running total. After the loop completes, the program prints the final sum of all numbers in the list.

Section 5: UML

The sequence diagram below shows how the recursive descent parser handles the expression `1 + 1 < 2 * 3` illustrating how control flows down through the grammar levels and back up as each sub-expression is returned.



And here is the source code from https://sequencediagram.org used to generate the diagram:

```
title Parsing "1 + 1 < 2 * 3"

parseExpression()->parseEqualityExpression():
parseEqualityExpression()->parseComparisonExpression():
parseComparisonExpression()->parseAdditiveExpression():
parseAdditiveExpression()->parseFactorExpression():
parseFactorExpression()->parseUnaryExpression():
parseUnaryExpression()->parsePrimaryExpression():

parseUnaryExpression()<-parsePrimaryExpression(): IntegerLiteral(1)
parseFactorExpression()<-parseUnaryExpression(): IntegerLiteral(1)
parseAdditiveExpression()<-parseFactorExpression(): IntegerLiteral(1)

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

parseUnaryExpression()<-parsePrimaryExpression(): IntegerLiteral(1)
parseFactorExpression()<-parseUnaryExpression(): IntegerLiteral(1)
parseAdditiveExpression()<-parseFactorExpression(): AdditiveExpr(1+1)

parseComparisonExpression()<-parseAdditiveExpression(): AdditiveExpr(1+1)

parseComparisonExpression()->parseAdditiveExpression():
parseAdditiveExpression()->parseFactorExpression():
parseFactorExpression()->parseUnaryExpression():
parseUnaryExpression()->parsePrimaryExpression():

parseUnaryExpression()<-parsePrimaryExpression(): IntegerLiteral(2)
parseFactorExpression()<-parseUnaryExpression(): IntegerLiteral(2)

parseFactorExpression()->parseUnaryExpression():
parseUnaryExpression()->parsePrimaryExpression():

parseUnaryExpression()<-parsePrimaryExpression(): IntegerLiteral(3)
parseFactorExpression()<-parseUnaryExpression(): FactorExpr(2*3)
parseAdditiveExpression()<-parseFactorExpression(): FactorExpr(2*3)

parseComparisonExpression()<-parseAdditiveExpression(): ComparisonExpr(<)
parseEqualityExpression()<-parseComparisonExpression(): ComparisonExpr(<)
parseExpression()<-parseEqualityExpression(): ComparisonExpr(<)
```

Section 6: Design Trade-offs

Hand-Written Recursive Descent vs. Parser Generators

One of the biggest design choices in a compiler is whether to write the parser by hand or use a parser generator. A recursive descent parser written by hand translates grammar rules directly into Java functionality via methods, making the control flow easy to follow and providing precise error messages.

A parser generator reduces manual parser code and may support grammars of higher complexity, but it also adds the need for additional tooling, generated code, and results directly in less direct control over behavior. For CatScript, recursive descent is a better fit due to the fact the grammar is small, the parser needed clear student-friendly error messages, and keeping the parser as plain Java kept the project simpler to build and maintain.

Section 7: Software Development Life Cycle Model

We used Test Driven Development as our life cycle model.

How TDD Worked in Practice

The course structured each assignment as a set of failing JUnit tests that we had to make pass. In practice, the workflow followed the classic red, green, refactor cycle: run the tests and watch them fail, write just enough code to make them pass, then clean up the implementation without breaking behavior. For the partner exchange component, Kyla Andersen wrote three additional tests and sent them over before Eoghan Perkins integrated them.

What Worked Well

The provided test suite was effectively the language specification, which enabled tests to provide a clear thresehhold for correctness across all of the goals of the language and removed ambiguity about expected behavior. Building one feature at a time and mastering individual concepts through assignments really assisted in learning the material at a reasonable pace. A test suite 'going green' allowed refactoring with immediete feedback upon something breaking.

What Was Harder

The provided tests made for good coverage of the overall goal of learning compilation, but edge cases such as unusual type coercions or deeply nested expressions occasionally required debugging. Partner coordination add a slight amount of mental overhead due to the coordination and ensuring our functionality was written similarly. Ultimately it was not a large issue to have to deal with.

Overall, Test-Driven-Development was a good fit for a compiler project where correctness is well-defined and easy to verify programmatically. The test suite serves as living documentation of what CatScript is supposed to do.