# Capstone Documentation


## Owen Cool, Connor Munson


~ CSCI 468: Compilers ~

# Section 1: Program

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

# Section 2: Teamwork

Team Member 1 (Owen Cool) was the primary implementor, responsible for building the CatScript compiler pipeline from end to end: the recursive-descent parser, the type system and semantic analysis pass, the tree-walking evaluator, and the JVM bytecode generation backend. Roughly 50 hours were spent on implementation.

Team Member 2 (Connor Munson) served as the tester and documentor, writing the partner test suite below and authoring the CatScript language guide found in Section 4. Roughly 10 hours were spent on documentation and testing.

Coordination was done in person, with files shared asynchronously via email. Team Member 1 implemented new compiler functionality in the repository and Team Member 2 sent the tests via email, allowing Team Member 1 to see any failures.

### Partner Tests (`csci468.partner.PartnerTests`)

```
// Classifies each number in a list as 1, 0, or -1 by sign.
// Tests: else-if chains, unary minus in print
@Test
void signClassifierTest() {
    assertEquals("1\n-1\n0\n-1\n1\n", execute("""
        var nums : list<int> = [3, -1, 0, -5, 7]
        for (n in nums) {
            if (n > 0) {
                print(1)
            } else {
                if (n < 0) {
                    print(-1)
                } else {
                    print(0)
                }
            }
        }
        """));
}

// Prints a 3x3 multiplication table as a flat list.
// Tests: nested for loops, multiplication operator
@Test
void multiplicationTableTest() {
    assertEquals("1\n2\n3\n2\n4\n6\n3\n6\n9\n", execute("""
        var rows : list<int> = [1, 2, 3]
        var cols : list<int> = [1, 2, 3]
        for (r in rows) {
            for (c in cols) {
                print(r * c)
            }
        }
        """));
}

// Tests: for-in loop, accumulator pattern, function call and return
@Test
void sumListTest() {
    assertEquals("25\n", execute("""
        function sumList(nums : list<int>) : int {
            var total : int = 0
            for (n in nums) {
```

```
                total = total + n
            }
            return total
        }
        var nums : list<int> = [3, 7, 2, 8, 5]
        print(sumList(nums))
        """));
}

// Tests: while loop, modulo operator, if/else, printing mixed int and string
@Test
void fizzBuzzLiteTest() {
    assertEquals("1\n2\nFizz\n4\n5\nFizz\n7\n8\nFizz\n10\n", execute("""
        var i : int = 1
        while (i <= 10) {
            if (i % 3 == 0) {
                print("Fizz")
            } else {
                print(i)
            }
            i = i + 1
        }
        """));
}
```

# Section 3: Design Pattern

The Memoization Pattern is implemented in `csci468.parser.CatscriptType.getListType()`. A static `HashMap<CatscriptType, ListType>` named `cache` stores previously created `ListType` instances keyed by their component type. When `getListType(type)` is called, `computeIfAbsent` checks the cache first; only if no entry exists is a new `ListType` constructed and stored.

```
static Map<CatscriptType, ListType> cache = new HashMap<>();

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

**Why this is better than doing nothing:**

Without memoization every call to `getListType(INT)` would allocate a brand-new `ListType` object. Because the type system uses reference equality in several places (e.g. comparing a variable's declared type against the inferred type of a list literal), two separate `ListType(INT)` instances would not be `==` even though they represent the same type. Memoization guarantees that `getListType(INT) == getListType(INT)` — the same object is returned every time — so identity-based comparisons and hash-map lookups work correctly without needing to override `equals`/`hashCode` on every subclass. It also avoids repeated heap allocation during parsing, which can happen many times as the semantic-analysis pass walks the AST.

# Section 4: Technical Writing/Catscript Guide

## Introduction

CatScript is a small, statically-typed programming language. Programs are compiled and run on the JVM. Its syntax is intentionally simple: a handful of statement types, a clean expression grammar, and a five-type system make it easy to pick up while still covering the fundamental ideas of structured programming.

---

## Types

- `int`: A 32-bit integer
- `string`: A Java-style string
- `bool`: A Boolean (`true` or `false`)
- `list<T>`: An Ordered list whose elements all have type `T`
- `object`: Any value (this is the top type of our language)

`null` is a valid value for any non-primitive type.

---

## Variables

Declare a variable with `var`. The type annotation after `:` is optional when the compiler can infer it, but required for `list` types:

```
var x : int = 10
var name : string = "CatScript"
var flag : bool = false
var scores : list<int> = [95, 87, 72]
```

After declaration a variable can be reassigned with a plain `=`:

```
x = x + 1
```

---

## Expressions

### Arithmetic

```
var a : int = 6 + 4    // 10
var b : int = 6 - 4    // 2
var c : int = 6 * 4    // 24
var d : int = 6 / 4    // 1  (integer division)
var e : int = 6 % 4    // 2  (remainder)
```

### Comparison and equality

```
var gt  : bool = 5 > 3    // true
var lte : bool = 5 <= 3   // false
var eq  : bool = 5 == 5   // true
var ne  : bool = 5 != 3   // true
```

### Unary operators

```
var neg : int  = -5
var inv : bool = not true   // false
```

### List indexing

```
var items : list<int> = [10, 20, 30]
var first : int = items[0]   // 10
```

---

# Print

`print` outputs any value followed by a newline:

```
print("hello")   // hello
print(42)        // 42
print(true)      // true
```

---

# If / Else

```
if (x > 0) {
    print("positive")
} else {
    if (x < 0) {
        print("negative")
    } else {
        print("zero")
    }
}
```

---

# For Loop

Iterate over every element of a list:

```
var nums : list<int> = [1, 2, 3, 4, 5]
for (n in nums) {
    print(n)
}
```

---

# While Loop

Repeat while a condition is true:

```
var i : int = 0
while (i < 5) {
    print(i)
    i = i + 1
}
```

---

# Functions

Declare with `function`. Parameters and the return type are annotated with `:`. A `void` function omits the return-type annotation:

```
function greet(name : string) {
    print("Hello, " + name)
}

function square(n : int) : int {
    return n * n
}

greet("world")
print(square(7))   // 49
```

---

# Complete Example Program

The program below computes the factorial of each number in a list and prints a small multiplication table, demonstrating variables, functions, `for`, `while`, and `if/else`:

```
function factorial(n : int) : int {
    var result : int = 1
    var i : int = 1
    while (i <= n) {
        result = result * i
        i = i + 1
    }
    return result
}

function isEven(n : int) : bool {
    return n % 2 == 0
}

var inputs : list<int> = [1, 2, 3, 4, 5, 6]

for (n in inputs) {
    var f : int = factorial(n)
    if (isEven(n)) {
        print(f)
    } else {
        print(-f)
    }
}
```

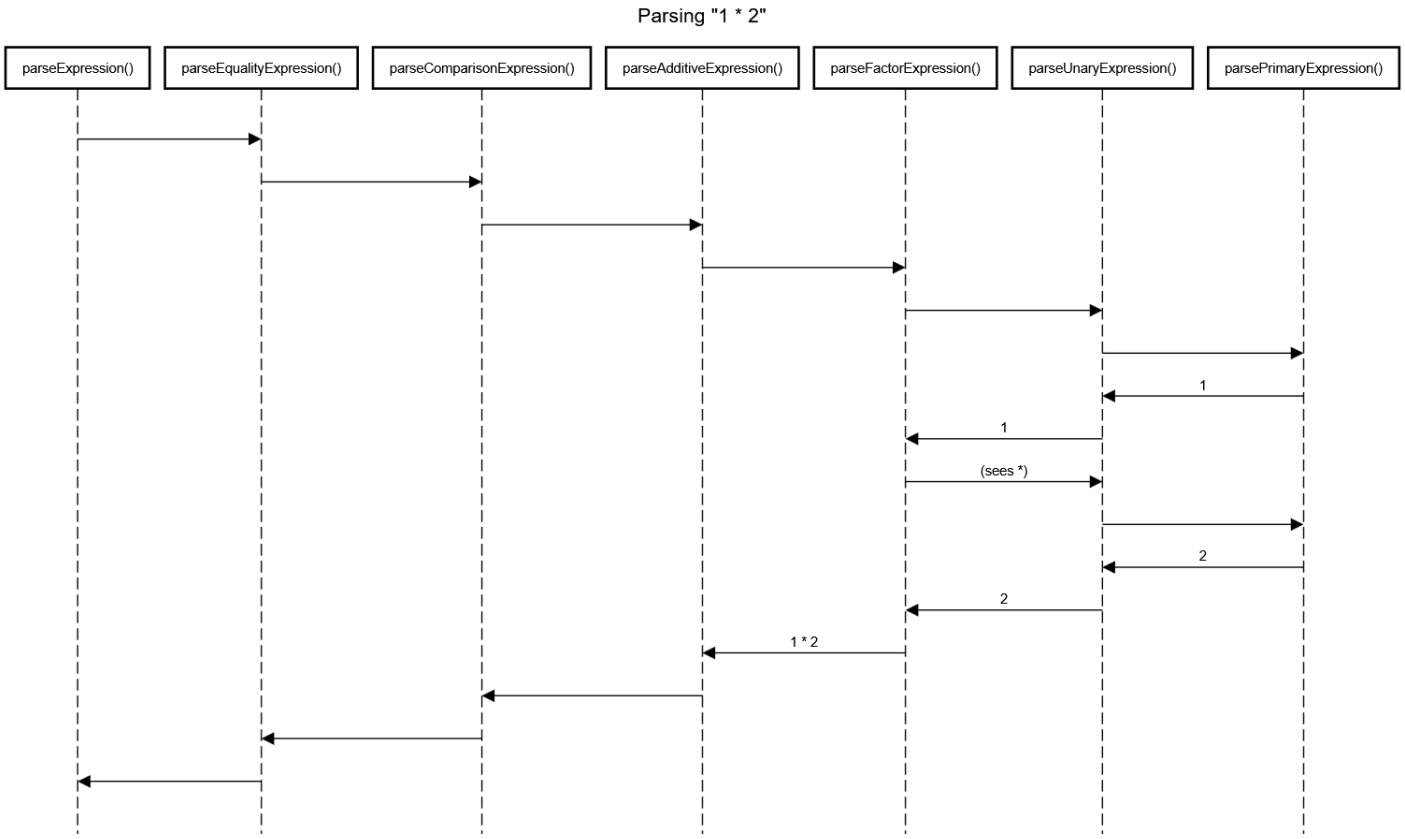
Expected output:

```
-1
2
-6
24
-120
720
```

# Section 5: UML

Include a Sequence Diagram generated via https://sequencediagram.org/

Here is an example diagram:




And here is the code from https://sequencediagram.org for the expression `1 * 2`. `*` binds at the *factor* level, so the additive and higher layers delegate straight through while the factor layer handles both operands.

```
title Parsing "1 * 2"

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

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

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

parseUnaryExpression()<-parsePrimaryExpression(): 2
parseFactorExpression()<-parseUnaryExpression(): 2
parseAdditiveExpression()<-parseFactorExpression(): 1 * 2
parseComparisonExpression()<-parseAdditiveExpression():
parseEqualityExpression()<-parseComparisonExpression():
parseExpression()<-parseEqualityExpression():
```

# Section 6: Design Trade-offs

CatScript uses a hand-written recursive-descent parser. An alternative would be to use a parser generator such as ANTLR or Yacc/Bison, where the grammar is written declaratively and a tool generates the parser code automatically. Each approach has meaningful trade-offs.

**Hand-written recursive descent (what we used)**

The grammar maps directly onto a call hierarchy: each non-terminal becomes a method (`parseExpression`, `parseAdditiveExpression`, etc.), and operator precedence is encoded by the depth of those calls rather than by explicit precedence declarations. This makes the parser easy to read, step through in a debugger, and extend with custom logic — for example, producing informative error messages at exactly the right point in the input, or handling context-sensitive constructs that are awkward to express in a pure grammar rule. The downside is verbosity: every grammar rule requires hand-written Java, so adding or changing a production requires finding and editing the corresponding method. There is also no tooling to warn you when the grammar becomes ambiguous; ambiguities only surface as unexpected parse results.

**Parser generator**

A parser generator accepts a compact, declarative grammar file and emits the parser automatically. Operator precedence and associativity are declared in one place rather than baked into the call structure, making the grammar easier to audit at a glance. The tool can also detect certain classes of ambiguity at grammar-compile time. The trade-offs run the other way: generated code is typically unreadable and nearly impossible to debug directly, error messages from the generated parser tend to be generic, and the project gains a build-time dependency on the generator tool and its runtime library.

**Which is better here?**

For a language the size of CatScript, hand-written recursive descent is the pragmatic choice. The grammar is small enough that the verbosity cost is manageable, and the direct control over parsing makes it straightforward to emit precise syntax-error nodes and integrate tightly with the semantic-analysis pass. A parser generator would pay off on a larger or more frequently evolving grammar where maintaining every production by hand would become a real burden.

# Section 7: Software Development Life Cycle Model

We used Test Driven Development as our life cycle model.

The project shipped in clearly defined checkpoints (parsing, semantics, evaluation, bytecode). Before starting each checkpoint Team Member 1 reviewed the provided test suite to understand the expected behavior, then wrote just enough implementation to make those tests pass. After completing all checkpoints, Team Member 2 wrote the partner tests (`csci468.partner.PartnerTests`) to cover integration scenarios — programs that combined multiple language features.

**What worked well:**

- The existing checkpoint tests acted as a precise specification. There was never any ambiguity about what "done" meant for a given feature.
- Failing tests caught regressions immediately; because every new piece of functionality was accompanied by tests, broken behavior surfaced within minutes rather than days.
- The division of labor (implementor vs. tester) meant the partner tests were written by someone who had not written the production code, which produced more independent coverage and caught assumptions the implementor had baked in without realizing it.

**What was challenging:**

- Team members worked at different schedules, and it was difficult to ensure tests were ready to be run when they were delivered, or that tests were available to be run when code was complete
- Team members were available at different times and synchronous communication was sometimes difficult to coordinate.