CSCI 468 Capstone Document

By Thomas Wesley Gaylord, Partner: Brandon Chandler

Section 1: Link to The Source Code

The source code can be found at https://github.com/ThomasGaylord/csci-468-spring2025-private/blob/main/capstone/portfolio/source.zip

Section 2: Teamwork

For the teamwork section we were instructed to write a version of the documentation and three tests for the compiler, then send it to our partner for them to submit our version as we submit their version. In the technical writing section is my partner's version of the documentation while the three tests are below. The three tests test the logical negation of comparison operators, the comparison of different types, and the comparison of Booleans which all pass. The documentation meanwhile lightly explains the workings and purpose of the CatScript compiler enough that it should be relatively easy for a new programmer to learn the CatScript language.

```
@Test
void logicalNegationOfComparisonWorks() {
    assertEquals(false, evaluateExpression("not(3 < 5)"));
}

@Test
void comparingDifferentTypesReturnsFalse() {
    assertEquals(false, evaluateExpression("3 == \"3\""));
}

@Test
void boolComparisonsAreCorrect() {
    assertEquals(true, evaluateExpression("true == true"));
    assertEquals(false, evaluateExpression("true == false"));
}
```

Section 3: Design Pattern

The design pattern we used in this project was memoization. Memoization is the process of storing previously seen values and reusing them later rather than redoing the process of creating or finding that value. A common example of this is when implementing a Fibonacci sequence, a sequence where each number is equivalent to the sum of the two previous numbers. When implemented with memoization for each current number the two previous would already be stored since they had to be calculated already in order to reach the current number, therefor the current number could just reference the previous two numbers rather than recalculate them. The way we implemented memoization for the project was when getting list types. The code, shown below, saves previously seen list types then when the method is run and the type matches it gives the saved list type, otherwise it makes a new list type and saves it in the HashMap. In all, we used the design pattern called memoization to save resources by saving previously seen list types for later use.

```
static Map<CatscriptType, CatscriptType> MEMOIZATION_CACHE = new HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
    CatscriptType existingType = MEMOIZATION_CACHE.get(type);
    if(existingType != null){
        return existingType;
    } else {
        ListType listType = new ListType(type);
        MEMOIZATION_CACHE.put(type, listType);
        return listType;
    }
}
```

Section 4: Technical Writing (Code Documentation)

CatScript Language Overview

CatScript is a simplified educational programming language modeled after JavaScript, developed to help students learn the fundamentals of how compilers and interpreters work. It features a dynamically typed system, basic syntax, and enough structure to demonstrate the process of parsing, type-checking, and execution.

CatScript was designed specifically for compiler instruction, which means the language isn't built for large-scale applications, but rather to illustrate key compiler components like tokenization, parsing, and AST traversal. Programs written in CatScript can be either interpreted or compiled into bytecode, depending on the phase of execution. In this document, we'll walk through the essential parts of the language and what makes it useful as a teaching tool.

Core Language Concepts

Dynamic Typing

In CatScript, variables do not require explicit type declarations. Instead, the language uses type inference. A variable assigned a number will be treated as an integer, while one assigned text will be treated as a string. Booleans, null, and lists are also supported. Although types are inferred, type checking is still enforced later in the verification phase to catch incompatible operations.

Example:

```
var a = 5      # a is inferred as int
var b = "cat"  # b is inferred as string
```

Dynamic typing makes the language easier to write for beginners, but it also means you must be careful about the types you are working with. Type mismatches (like trying to multiply a string) will result in errors during program verification.

Expressions

Expressions are components of the language that resolve to a value. They can be used in assignments, control flow conditions, function parameters, and more.

Additive Expression

Handles both numeric addition and string concatenation. If both operands are numbers, they are added. If either is a string, the result is a concatenated string.

Example:

```
print(2 + 3)            # Outputs: 5
print("Hello" + "World") # Outputs: HelloWorld
print("Version " + 2)     # Outputs: Version 2
```

Comparison Expression

Used for checking the order or magnitude of two numeric expressions. Available comparison operators include >, >=, <, and <=, which return a Boolean result.

Example:

```
print(3 < 5)    # true
print(10 >= 20) # false
print(100 <= 100) # true
```

These are commonly used in loops and conditional logic.

Equality Expression

Used to compare two expressions for equality (==) or inequality (!=). Works across different types, and results in true or false.

Example:

```
print(4 == 4)       # true
print("cat" != 3)   # true
print(null == null) # true
```

Equality checks help drive decisions in control flow logic.

Factor Expression

Supports multiplication (*) and division (/) operations between integers.

Example:

```
print(6 * 2)   # 12

print(8 / 4)   # 2

print(9 / 2)   # 4 (CatScript uses integer division)
```

This expression is evaluated before additive expressions due to operator precedence.

Function Call Expression

Allows calling functions within expressions. The return value of the function is used in place. This is separate from a function call statement which is used only for side effects.

Example:

```
function double(x : int) : int { return x * 2 }

print(double(5)) # Outputs: 10
```

Function call expressions can be chained or nested as part of larger expressions.

Identifier Expression

Used to reference previously declared variables. The interpreter or compiler will look up the value associated with the identifier at runtime.

Example:

```
var name = "Leo"

print(name) # Outputs: Leo
```

If a variable is not declared before usage, an error is raised.

Parenthesized Expression

Changes the default order of operations by wrapping expressions in parentheses. This allows for explicit control over evaluation.

Example:

```
print(2 + 3 * 4)     # Outputs: 14
print((2 + 3) * 4)   # Outputs: 20
```

This expression type ensures more complex calculations can be clearly defined.

Unary Expression

Handles negation of numbers (-x) and logical negation of Booleans (not x). Unary expressions apply to a single operand.

Example:

```
print(-5)        # -5
print(not false) # true
print(not (3 > 5)) # true
```

Unary expressions add expressive power to conditionals and math.

Type Literal Expression

Used in type annotations or generic lists. For example, list<int> indicates a list containing integers.

Example:

```
var nums : list<int> = [1, 2, 3]
var flags : list<bool> = [true, false]
```

These are useful when you want to restrict a list to only certain data types.

Statements

Statements are complete instructions that tell the program to do something. These do not directly return values.

Assignment Statement

Assigns a value to an existing variable or one declared earlier. The value may be an expression.

Example:

```
var x = 10
x = x + 5
```

This updates the variable x to hold a new computed value.

For Statement

Iterates through a list and executes a block of code for each item in the list.

Example:

```
for(item in [1, 2, 3]) {
  print(item)
}
```

This is the main looping construct in CatScript.

Function Call Statement

Executes a function for its side effects rather than its return value.

Example:

```
function greet() { print("Hello!") }
greet()
```

Used when the return value isn't needed.

Function Definition Statement

Defines a new function in the language. A function can take parameters, optionally define their types, and return a result.

Example:

```
function square(x : int) : int {
  return x * x
```

```
}
```

Functions can encapsulate logic and be reused.

If Statement

Performs conditional logic.

Example:

```
if(3 > 2) {
  print("Yes")
} else {
  print("No")
}
```

You can nest if statements or chain them with else if.

Print Statement

Outputs the result of an expression to the console.

Example:

```
print("Testing output")
print(10 + 5)
```

Often used for debugging or user feedback.

Return Statement

Returns a value from a function.

Example:

```
function getTen() : int {
  return 10
}
```

A return without a value returns void.

Variable Statement

Declares a variable in the current scope.

Example:

var word : string

var count = 5

Variables can be declared with or without initialization.

Language Internals

Tokenization

The first phase of the compiler turns raw source code into a sequence of tokens. Tokens are strings with assigned types such as INT_LITERAL, IDENTIFIER, KEYWORD, or OPERATOR. This process is handled by the CatScriptTokenizer. Tokenization breaks text into meaningful components and removes whitespace or comments.

Parsing

The CatScriptParser uses recursive descent parsing to convert the token stream into an Abstract Syntax Tree (AST). Each node in the AST represents a meaningful piece of the program such as an AdditiveExpression or a FunctionCallStatement. Parsing ensures code is syntactically correct and structured.

Verification (Static Analysis)

After parsing, the tree is validated. The verifier walks through the AST and checks for semantic errors, such as using undeclared variables, invalid type operations, or mismatched return types in functions. This is an essential step to catch mistakes before running the program.

Evaluation (Interpreter Mode)

Each AST node has an evaluate() method. In interpreter mode, the AST is executed directly in memory. Values are computed on the fly, and runtime environments manage scopes, function calls, and data types. This method of execution is good for testing and debugging.

Bytecode Compilation (JVM Mode)

For more advanced use, CatScript supports bytecode generation. Each AST node can generate JVM-compatible instructions using the ASM library. These instructions can be written into .class files and executed using a standard Java Virtual Machine.

Example:

var x = 2 + 2

Compiles into bytecode resembling:

ICONST_2

ICONST_2

IADD

ISTORE_1

This compilation process teaches students how real-world compilers like java works.

Error Handling

CatScript includes a basic error reporting system. During tokenization and parsing, syntax errors are caught. During verification, semantic issues are flagged. At runtime, undefined variables, invalid operations, and null dereferencing are all captured and reported with line number information to aid debugging.

Errors are designed to be student-friendly and descriptive.

Summary

CatScript is not just a language; it's a learning platform. Its simplified syntax, combined with a complete compilation and execution pipeline, makes it ideal for educational settings. Students can:

- Learn recursive descent parsing
- Build and walk ASTs

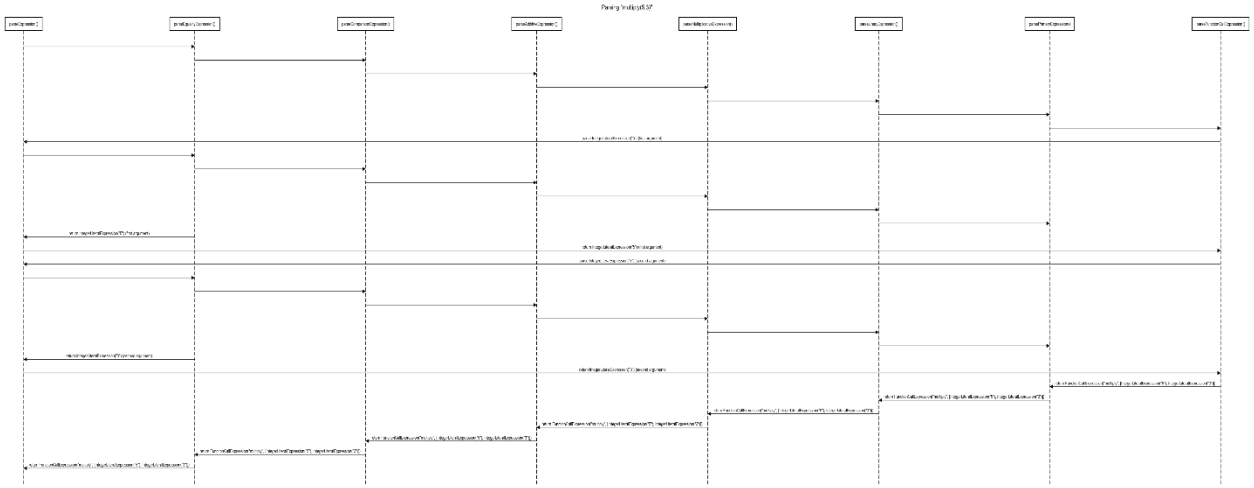
- Implement interpreters and compilers
- Understand scoping and type inference
- Generate real bytecode
- Debug using meaningful error messages

By writing or modifying CatScript, learners gain deep insight into how programming languages function, both in theory and in practice.

Section 5: UML

Here I have a sequence diagram for my functionCallExpression() when it is parsing a function call for some function called multiply taking the integer arguments 5 and 3. The parser first goes down the chain of parse expressions until it recognizes it's a function call expression. The parser then proceeds to parse the individual arguments. It calls parseExpression() to begin parsing the first argument, which is the integer 5. This triggers a series of functions, including parseEqualityExpression(), parseComparisonExpression(), and so on, until it reaches parsePrimaryExpression(), where it identifies 5 as an integer literal. It then creates and returns an IntegerLiteralExpression("5"). Similarly, the parser then moves to parse the second argument, 3, following the same chain of function calls and ultimately creating an IntegerLiteralExpression("3"). Once both arguments have been parsed, the parser constructs a FunctionCallExpression("multiply", [IntegerLiteralExpression("5"), IntegerLiteralExpression("3")]). This expression is then returned back through the chain of functions, resulting in the function call multiply(5, 3) being parsed correctly.

Note: The picture doesn't show up well in the document so I added it to the portfolio folder with my source code.



Section 6: Design Trade-offs

For this course we made a recursive-descent parser, but a parser generator could have done the same job. Parser generators take the rules of a language in a manner similar to a regular expression, then parse the code using those rules. In order to more thoroughly teach us the individual parts of a parser and give us the perspective of a more profession compiler programmer we made a recursive-descent parser by hand instead. While a parser generator is faster and more efficient to use, our personally made recursive-descent parser is more human-understandable, more easily modifiable, and uses fewer total lines of code compared to its equivalent parser generated code. Ultimately we made a recursive-descent parser in order to learn how to make a parser as well as to make the resultant parser much more modifiable compared to what a parser generator would make.

Section 7: Software Development Model

For this course we used a Test Driven Development (TDD) model. A TDD model is a software development model designed such that each section of a project can be implemented sequentially. We did this starting with tests for the parser itself that we had to build code to pass, then we had to pass tests for expressions, then statements, then validation, and finally bytecode. This resulted in more and more functionality coming together as we developed the code more, but it also resulted in bugs coming up in later sections that were due to our code in previous sections, causing confusion when trying to identify the problem. An example of this is not properly implementing ListLiteral functionality in the earlier stages resulting in a test in the later stages not passing. Overall, the TDD model that we used was good for sequentially building up functionality, but it also caused significant roadblocks to arise if we didn't properly implement our code in the early stages of testing.