Computer Science Senior Portfolio

Capstone Project

of

Anjali Riedel

Michael Downs

Montana State University

Compilers: CSCI 468

Spring 2025

1 Program

The CatScript program code can be found in: /capstone/portfolio/source.zip

2 Teamwork

Collaborating with someone else on this project was a unique experience and differed from any previous group projects. One of the team members worked on the compiler while attending school in a different country. This presented the difficulties and benefits of working with partners entirely online. It was a useful experience that can be carried on in future business endeavors.

All interactions were via Discord messages with an eight-hour time difference. This proved to come with some difficulties. Most often, one team member would send work over and wait for the other to wake up before receiving a response. This elongated the process of exchanging documents, and despite the obvious time crunch, there was a sign of procrastination in sharing revisions. The issue was brought to attention and fixed quickly, but it was an unexpected and enlightening experience.

Despite the time issue, both team members worked well together and communicated clearly. Team member 1 provided edits to team member 2, who wrote the CatScript document included in the Technical Writing section. The document includes many examples that provide users with a clear frame of reference. Team member 2 added information about the scope of the program that was not apparent to team member 1. It was interesting to see what a different mind understood about the same compiler language. Team member 2 also included three additional tests that checked for cases that were missed in the original test development process.

Overall, dividing the work so that one team member wrote the code and the other wrote the document was a productive and enjoyable experience. Often, coding with a partner can be difficult because each person can really have their own approach. By separating tasks in this way, neither team member was overwhelmed by the differences in pace and allowed team member 1 to truly learn the code.

3 Design Pattern

One design pattern present in the creation of CatScript is memoization. Memoization is an optimization technique that stores information in data structures to avoid redundancies. CatScript implements this pattern in the CatScriptType class. The method getListType is a good example of code that could be improved by utilizing memoization. The method

is expensive, as it is called multiple times in a single program execution. Instead of constructing a new ListType object each time the method is called, we search a HashMap to see if the type exists. If it does, the method returns the cached value, which avoids unnecessary computation.

4 Technical Writing

CatScript is a statically-typed programming language inspired by JavaScript. It was designed to be a tool for students to learn to create a compiler, so it is fairly simple. However, it still has modern and powerful features. This documentation covers the language's core concepts and syntax.

4.1 Lexical Elements

Element	Example	Notes
Identifier	letter, someFunction, a1	case-sensitive must start with a char. can include a-z, A-Z, 0-9
Integer literal	10, 8, -5	32-bit signed
String literal	"dog"	double quotes support for newline, tab, and quotes
Boolean literal	true, false	
Null literal	null	universal null value
List literal	[1,2,3]	homogeneous, compile-time, covariant, immutable
Comments	//single line	no block comments

Table 1: This table lists the available lexical elements along with tips for navigating them

4.2 Types

CatScript is statically typed with optional type inference.

4.2.1 Basic Types

- int: 32-bit signed integer
- bool: Boolean value
- string: Java-style string (immutable)
- object: Supertype meaning it is assignable **from** all types

- null: Null literal - assignable **to** all types
- void: No value (functions only)

Examples:

```
var a: int = 10
var b = "cat"    // inferred as a string
var c: string = null
```

4.2.2 Lists

- list: Ordered collection of values; uses type inference of elements to determine a list type
- list< T >: Explicitly typed. Ordered collection of values where T is the type; either int, bool, string, or object

Examples:

```
var nums = [1,2,3]                    // list<int>
var empties: list<string> = []
var strings: list = ["the", "it",]    // list<string>
```

4.3 Variables

```
var identifier [: Type] = expression
```

": Type" is an optional addition (the type can be inferred from the initializer)
Variables are mutable within their declaring scope
Example:

```
var counter = 0
counter = counter + 1
```

4.4 Expressions & Operators

Arithmetic: + - * / (returns an integer)
Comparison: == != > >= < <= (returns a boolean)
Unary: not(true/false) (returns a boolean); -x (integer negation)

Order of Operations: Arithmetic expresssions follow the rules of PMDAS (exponents are not supported)

Concatenation: Syntax is...string + object where the object is converted to a string. In the special case where object is of type null, null is converted to the string "null". All concatenations return a string value.

Examples:

```
var add = 1 + 1 * 2                    // evaluates to 3
var parenthesized = (3+2) * 4          // evaluates to 20
var addStrings = "the" + "it"          // evaluates to "theit"
var addStringAndOBject = "the" + 1     // evaluates to "the1"
var addStringAndNull = "the + null     // evaluates to "thenull"
```

4.5 Control Flow

4.5.1 if/else

```
if (condition) {
    //statements
} else if (other condition) {
    //statements
} else {
    //statements
}
```

requirements:

- condition must be boolean
- braces are required

4.5.2 for loop

iteration over *lists*

```
for (item in list) {
    // statements
    // e.g. print(item) will print every item in list
}
```

requirements:

- item is read-only within the for loops body
- requires keyword "in"

4.6 Functions

```
function name(paramList) [: ReturnType] {
    //statements
}
```

4.6.1 Parameters

```
paramList    -> [param {, param } ]
param        -> identifier [:Type]
```

parameters without a defined type default to null

4.6.2 Return Type

If function type definition is omitted, the default type is void. Non-void functions **must** return a value on every path; return-coverage analysis is enforced (e.g. every if branch must return)

4.6.3 Examples

```
function add(x: int, y: int) : int {
    return x + y
}

function describe(value) { //value type = null; return type = void
    print("You passed " + value)
}
```

4.7 Built-in Functions

CatScript has one built-in function: print. The print function outputs the parameter and automatically moves to the next line.

Function	Signature	Purpose
print	print(value: object)	console output with trailing space

Table 2: A table defining the print function

4.8 Scope & Lifetime

- Global Scope: top-level variables and functions
- Function Scope: parameters and local var declarations
- Block Scope: anything inside "..."

There is no support for variable shadowing meaning that duplicates within the same scope result in a compile-time error.

4.9 Error Checking

Compile-time errors include

- undefined variables or functions
- re-declaration of an identifier in the same scope
- type mismatch in assignments, return, or arguments
- missing return in non-void functions
- top-level return statements
- incompatible operations (e.g. true + 2)
- calling a non-function value

Runtime errors include

- division by zero

4.10 Inner Workings & Target

The compiler...

- tokenizes inputs
- parses to an AST
- validates semantics
- generates JVM bytecode (target: Java 17)

4.11 Best Practices

- Type Declaration: use explicit type declaration in most cases; this improves readability and results in simpler error checking
- Variable naming: use descriptive variable names and follow consistent naming conventions

5 UML

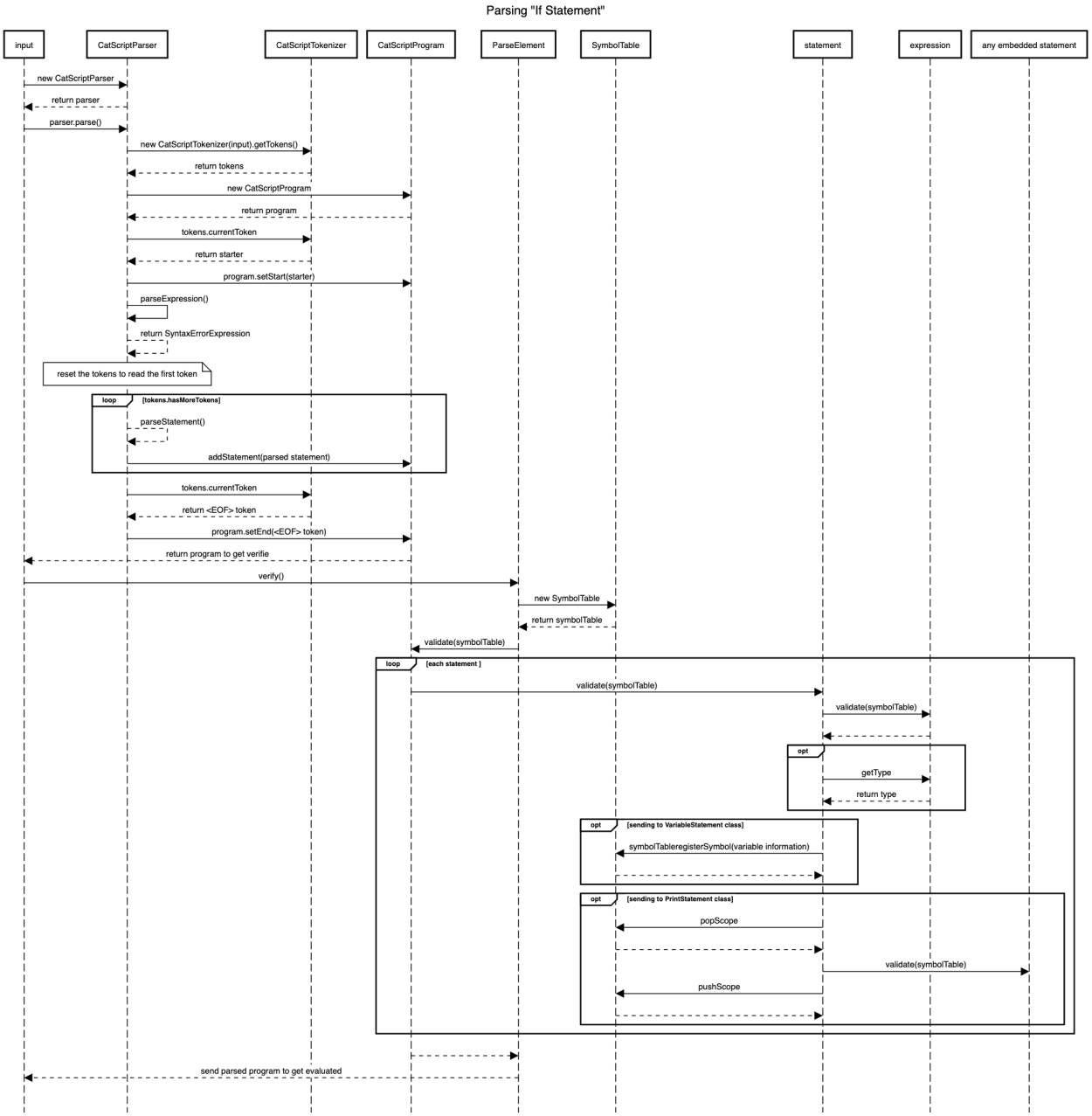

Figure 1: Sequence diagram representing the class movement of parsing an If Statement

The UML diagram in Figure 1 is a sequence diagram that traverses the classes to parse an if statement. The following program is the sample code for the diagram.

```
var x = 8
if(x < 12) {
   print(x)
}
```

This fairly simple program illustrates the complexities of the parser. For simplicity, the diagram disregards the loop that collects tokens from the program. Instead, the first loop

box represents parsing each statement in the program. For this particular program, we cycle through the loop twice; once for the variable statement and once for the if statement. Note that all parsing is done in the CatScriptParser class and is called recursively. After parsing all statements, the program is checked for syntactic errors by validating a symbol table.

The verify() method creates a new symbol table that will hold the defined variables. We then enter our second loop, which traverses through each statement and expression class. The "opt" boxes in this loop represent the different statements being called, as each statement invokes different methods depending on its meaning. Once the loop breaks, any errors found are collected and displayed. If there are no errors, the parser is complete and the program can be evaluated.

6 Design Trade-offs

The CatScript parser was written by hand in recursive-descent format instead of with a parser generator. This allowed a deeper understanding of what happens in the background of creating a parser.

Using a parser generator would have significantly reduced the amount of written code. A generator creates a layout and writes the hefty code, allowing for more creative freedom in the language itself. The user does not have to focus on the architecture of the code, only the grammar. In that way, I can see the benefits of a parser generator. The problem lies in the fact that the generated code is difficult to read.

Parser generators make all the design decisions, including what methods are named and how they are written. This leads to complex code that is hard to debug. It was referenced earlier, in the Teamwork section, how working on code with another person can be strenuous. Two people can have wildly different approaches to coding, leaving one person in the dust. Employing a parser generator gives the same effect. You are granting full freedom to an optimized generator that will likely have access to coding methods incomprehensible to the common coder.

This leaves some background on why CatScript was written by hand. It was an especially important design decision, since the purpose of the language was to learn how to create it. Writing the compiler by hand was a valuable skill that trained the algorithmic side of coding rather than simply learning how to use a product.

7 Software Development Lifecycle Model

This project was developed using Test-Driven Development. In test-driven development, multiple tests are written which assess different uses of the code. This method was an

efficient process that ensured complete implementation. By directing the development of CatScript with a test-driven approach, each use of the language was integrated into the compiler.

One benefit of test-driven code is readability. The tests dissect the code into smaller sections and create a step-by-step approach to the design of the language. Each step creates a clear path and disregards portions of code that may not yet be written. In turn, this keeps the focus in one place, neatly separating one application of the language from another.

Another benefit of TDD is the ease of debugging. When making changes to the code, there was always the possibility of affecting other applications of the language. When tests are broken down enough, errors can be easy to identify. Not only does this ensure full coverage, but it also makes fixing those mistakes quick and precise. The process is simple: find the test that no longer passes and step through the code, fixing any issue that pops up.

An issue I ran into while using tests was receiving incomprehensible error codes; a common occurrence in coding. However, this does not seem like a strong enough reason to avoid test-driven development. Most of the confusing errors came from generating bytecode, which elongated the coding process. Even so, the tests still provided valuable guidance.