

Computer Science Capstone

CSCI 468: Compilers
Spring 2023

Tyler Koon (Lead Engineer)
Joseph DiPrizio (QA Tester)

Introduction

This report documents the design and development processes for creating a compiler that implements the CatScript programming language, as part of the CSCI 468 Capstone project. The deliverable for this project is a recursive-decent compiler for the CatScript programming language—a simple yet fully featured C-like language that has been designed for instructional purposes. Given the grammar for this language, we were tasked with developing a complete compiler that included a tokenizer, lexical parser, semantic analyzer and evaluator, and a high-level transpiler that targets JVM bytecode. This document covers the broad development of these stages, including discussions on the team dynamic, the chosen design pattern and its implementation, documentation for the CatScript programming language itself, the structure of the project (presented as UML diagrams), consideration of the design trade-offs, as well as an in-depth analysis of the chosen development life-cycle model. We first begin by defining the team dynamic for this project.

1 Program

Please see the attached 'source.zip' file for the project source code and program specification.

2 Teamwork

For this project we were assigned to groups of two members, with each member implementing their own compiler. The dynamic within teams followed a simple developer-QA relationship: I assumed the responsibility as the lead engineer and developer, and my colleague contributed QA testing and documentation services. As the lead engineer, I implemented the tokenizer, parser, evaluation, and transpilation systems, and resolved each of the unit tests. My colleague provided additional unit tests to verify the functionality and completeness of my implementation, and additionally composed the accompanying documentation of the compiler and underlying CatScript language. Given the inherent differences in our roles, I contributed approximately 90 - 95% of the work. Although perhaps an extreme example, this duality represents what modern development teams might look like; lead engineers working closely with QA and DevOps engineers to develop fully qualified products through an efficient production pipeline.

3 Design Pattern Implementation

One design pattern that was implemented in this project was the memoization pattern. This pattern is an optimization technique that wraps deterministic functions and caches their result in a persistent data structure. This allows for future function calls to return cached items corresponding to the same input parameters in place of creating an entirely new object in memory. The primary benefit of implementing this pattern is that of memory savings; especially in scenarios that involve the deterministic invocation of functions many times through the program lifetime, such as when performing lexical or semantic analysis.

In this project, we implement the memoization design pattern during the semantic analysis phase. This phase primarily supports the static type system, which involves the creation of many identical *CatscriptType* objects; a prime opportunity for memoization. A specific example of this implementation occurs in the 'CatscriptType' class on the 'getListType' method, which is responsible for retrieving the list variant of a CatScript type. To memoize this functionality, we first created a hash map to cache these list variant objects:

```
// Map object for memoizing parameterized CatScript types of a list
private static final Map<CatscriptType, CatscriptType> LIST_TYPES = new HashMap<>();
```

Then, when retrieving a list variant of a CatScript type we first check this LIST_TYPES map for the corresponding CatscriptType object. If a copy of the object exists, then we simply return a reference to that object. Otherwise, we instantiate a new object and update the cache:

```

public static CatscriptType getListType(CatscriptType type) {
    // Attempt to fetch the matching ListType object from the cache
    CatscriptType listType = LIST_TYPES.get(type);

    // If the cache misses, update the cache
    if(listType == null) {
        // Create a new ListType for the provided CatScript type
        listType = new ListType(type);

        // Insert a reference to the associated ListType into the cache
        LIST_TYPES.put(type, listType);
    }

    return listType;
}

```

Memoizing this method improves efficiency when typing list objects during the static analysis phase by preventing the creation of redundant 'CatscriptType' objects. In many cases, this also improves the execution time for this function as there is no longer the overhead of performing memory allocation for each invocation. Although a relatively inexpensive operation, this demonstrates how the memoization pattern can improve the memory management and speed of repetitive operations through the implementation of an efficient caching structure.

4 Technical Writing

CatScript is a simple programming language that we have implemented over the course of the Spring 2023 semester. It is a statically typed language, and includes a number of features such as lists, basic operations, and several structures that manipulate control flow, such as loops and function calls. This documentation aims to explain these different features.

Assignment

Assignment in CatScript allows users to construct user defined variables and functions. This is accomplished by using identifiers to act as a label for the function or variable. When using identifiers this way, it is important to use valid identifiers for label names. Invalid identifiers are either reserved keywords violate the rules for identifier names. These cases are described in the following subsections.

Reserved Keywords

Reserved keywords are set aside to perform specific functions for in the CatScript language. An example of a keyword is "for". The keyword "for" is reserved for "for statements", which are discussed later. All of the reserved keywords in the CatScript language are: **else, false, function, for, in, if, not, null, print, return, true, var.**

Identifiers

When the CatScript Tokenizer scans to check if something is an identifier, it first checks if it begins with an alphabetic character, and then continues to scan for alphanumeric characters. Additionally, valid identifiers cannot include any special characters or spaces. The following table provides some examples of what valid and invalid identifiers look like.

Valid	Invalid
string1	1string
variable	var\$\$
counterVar	counter-var

Variables

As previously mentioned, assignment allows a user to utilize variables. A variable declaration assumes the following syntax:

```
var IDENTIFIER = VALUE
```

An important note about CatScript variables is the their value type either be defined explicitly or implicitly. In the following example, 21 is an integer, so the variable "implicitNum" is implicitly typed as an integer. On the other hand, the variable "explicitNum" is explicitly typed as an integer:

```
// This variable is implicitly typed to an integer based on the value
var implicitNum = 21

// This variable is explicitly typed to an integer
var explicitNum: int = 21
```

Importantly, CatScript does not support uninitialized variables, which means that all variables must be initialized at declaration.

```
// This would be invalid because the variable 'myVar' is not being initialized
var myVar

// Instead, you must initialize the variable upon declaration
var myVar: string = "My favorite string"
```

Functions

Function definitions appear as the following in Catscript:

```
function IDENTIFIER (PARAMETER_LIST): OPTIONAL_RETURN_TYPE {
    // Function body
}
```

The identifier is the label for the function, and follows the rules defined in the *Identifiers* section. The parameter list is an optional list of identifiers that can be passed into the function (with optional type information). The return type is the type that the function returns, and also optional. By default, the return type of a function is VOID. An example of a fully qualified function declaration might look like:

```
function myFunction (param: int): bool{
    // Function body
}
```

The body for a function can consist of any valid set of statements or expressions. Additionally, the function body can contain a "return statement", which allows for a value to be returned from the function during or after execution. The "return statement" is further explored in the next section.

Control Flow

Control flow refers to the features in the CatScript programming language that alter the normal flow of execution of a program, which is the execution of program statements in a specified order (statement 1, statement 2, ...). These features typically jump between sections of code, either executing a certain portion of code or skipping over a section. CatScript includes three of these control flow features: functions/return statements, for loops, and if statements.

Function and Return Statements

A function call statement alters control flow by executing the statements in the scope of a function body, and then returning to the parent scope to resume execution after the call. A return statement stops execution of the function body, optionally returning a value to the parent scope.

A function call might look like the following:

```
// Define the function
function isLessThanTen(num): bool{
    if(num < 10){
        // Return a 'true' boolean to the parent scope
        return true;
    }
    print("Number is not less than ten")
    // Return a 'false' boolean to the parent scope
    return false;
}

// Call the function
var test = isLessThanTen(9)
```

In this example, calling the "isLessThanTen" function changes the execution order to the function body. Since 9 is less than 10, the return statement returns true and the print statement is not executed as the execution flow has returned to the parent scope (i.e., the scope where the parent function was called).

For Statements

A "for statement" allows for iteration through elements in an iterable objects. The elements in the iterable object are exposed through a locally scoped identifier provided in the statement definition. The "for statement" assumes the following syntax:

```
for(IDENTIFIER in ITERABLE_EXPRESSION){
    \\ Body statements
}
```

In CatScript, the only valid iterable expressions are lists. When iterating through a list, the elements in the list are individually assigned to the IDENTIFIER variable for each iteration, accessible only by the body of the for statement. Iterating through a list of integers might look like:

```
var myList: list<int> = [1,2,3]

// Iterate through elements in the list
for(num in myList){
    print(num)
}
```

Importantly, the IDENTIFIER infers type information from the ITERABLE_EXPRESSION. So in the above example, the local 'num' variable will be typed as an integer. That said, the type information can also be defined explicitly, as demonstrated in the following example:

```
var myImplicitList = [1,2,3]

// Iterate through elements in the list, explicitly defining the type of the "num"
// identifier
for(num: int in myImplicitList){
    print(num)
}
```

If Statements

An "if statement" allows for logical conditions to be used in the CatScript programming language. If these conditions are met, the "if statement" executes the code in its body, otherwise it skips over it. This behavior is identical to how other high-level programming languages handle such statements. An if statement uses the following syntax:

```
if(EXPRESSION){
    // At least one statement
} else (if | { statement })
```

If the expression evaluates to true, the code is executed, otherwise it is not. The "else" at the end of the "if statement" is optional, and can be paired with either another "if statement" to have multiple logical conditions or can standalone to handle trailing cases (i.e., when all previous conditions have evaluated to false). Some examples of valid if statements include:

```
// If/else statement
if(2 + 2 == 4){
    print("This code will be executed")
} else {
    print("This code will not be executed")
}

var number: int = 7

// Complex conditional if/else if/else statement
if(number == 0){
    print("Will not be executed, since 7 does not equal 0")
}else if(number == 6){
    print("Will not be executed, since 7 does not equal 6")
}else if(number == 7){
    print("Will be executed, since expression 7 == 7 is true")
}else{
    print("Will not execute, since one of the above if statements evaluated to true.")
}
```

Lists

Lists allow the user to store information into a single, iterable object. A CatScript list, and lists in general, can best be thought of as sets containing objects. Lists in CatScript can only contain elements of a single type, so [1, 2, 3] is allowed but [1, "cat", 3] is not. Some examples of lists include the following:

```
// List of strings (component type inferred from value)
var stringList = ["cat", "dog", "goat"]

// List of integers
var intList: list<int> = [1, 2, 3, 4, 5]

// List of lists of integers
var listList: list<list<int>> = [[1,2],[3,4]]
```

In these examples, we can see that the lists support both implicit and explicit type declarations. As for the list of lists, it is important to note that the list elements must also be of the same the same component types.

An important note is that Catscript lists are covariant according to their component type. For example, a list of strings could be assigned to a list of objects, however a list of objects cannot be assigned to a list of strings. The immutability of lists (i.e., the inability for lists to be changed changed after creation) means that this covariance is type safe.

Operators

The CatScript language contains basic operators including addition, subtraction, multiplication, division, and logical negation. Addition and subtraction are handled by additive expressions, multiplication and division are handled by factor expressions, and negations are handled by unary expressions. They are implemented according to the following grammatical rules:

```
additive_expression = factor_expression +|- factor_expression
```

```
factor_expression = unary_expression */ unary_expression
```

```
unary_expression = not|- unary_expression|primary_expression
```

Some examples of implementing these expression are:

```
// Add two integers
1 + 1

// Subtract two integers
(1 + 1) - 2

// Multiply and divide integers
(6 * 4) / 2

// Perform logical negation
!true

// Add two negative integers
-1 + -1
```

Note that CatScript also supports parenthetical expressions. In the examples above, additive and factor expressions were surrounded by parenthesis to overwrite the default order of operations—the contents of the parenthesis being evaluated first within the context of their parent scope.

One special function of the additive expression is that it supports concatenation of strings, which simply combines two component strings into one:

```
// Concatenate two strings to produce "MyString"
var myString: string = "My" + "String"
```

Finally, it is important to note that CatScript does not support non-integer numbers. This impacts factor expressions, which will round up to the nearest whole number. So, something like $3/2$ will evaluate to 1.

Type System

Catscript, as mentioned earlier, CatScript is statically typed. This means that types and type expectations are known at compile time instead of at runtime, unlike dynamically typed languages. The CatScript type system supports the following types:

- int: A 32 bit integer
- string: A java-style string
- bool: A boolean value
- list: A list of values with the type 'x'
- null: The null type
- object: Any type of value
- void: The void type, only used for function returns

All of the different Catscript types follow certain assignability rules, which means some types are assignable to one type but not another. These assignability rules are as follows:

- int: Assignable to object
- string: Assignable to object
- bool: Assignable to object

- list: Component types are assignable to any type (excluding void)
- null: Anything is assignable from null
- object: Assignable to null, but not assignable to any of the primitive types (int, string, bool).
- void: Nothing is assignable from void

Generally speaking, more types of more specificity can be assigned to more general types. For example, the integer, boolean, string, and list types can be assigned to the object type:

```
// Assign an integer to an object
var myNumber: int = 8
var myFirstObject: object = myNumber

// Assign a string to an object
var myString: string = "some string value"
var mySecondObject: object = myString

// Assign a boolean to an object
var myBool: bool = true
var myThirdObject: object = myBool

// Assign a list of integers to an object
var intList: list<int> = [1,2,3]
var objList: list<object> = intList // Demonstrates covariance with list types
```

That said, the object type cannot be assigned back to these child types. This prevents the ambiguity and ensures type safety.

```
var myObject: object = "test"

// This is invalid
var myString: string = myObject
```

Additionally, everything is assignable from the null type:

```
// Assign null to an integer
var myNullNumber: int = null

// Assign null to a string or object
var myNullObject: string = null
```

Finally, the void type is a special type that is not assignable to any other type. It is only used to define functions that do not return a value:

```
// This is invalid
var myInvalidString: void = "string"

// Instead, void is used to define a function that has no return value
function myFunction(): void{
    // Only executes statements in this body, doesn't return any values
    return
}
```


5 UML

To encapsulate the behavior of the entire compilation process, we created a sequence diagram that demonstrates the expected behavior and relationships for each part of the compiler.

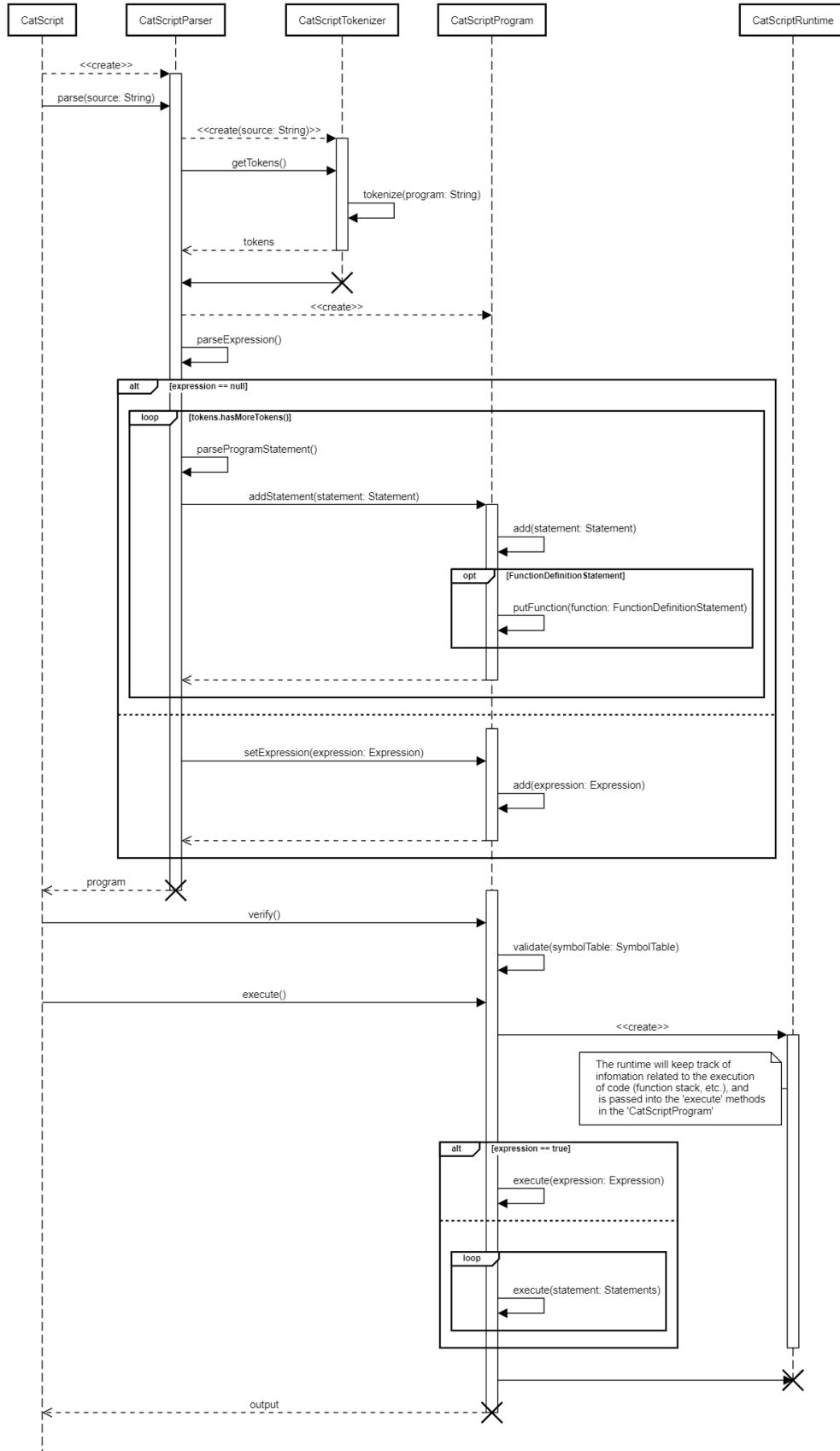


Figure 1: Sequence diagram of CatScript compilation process

6 Design Trade-Offs

When implementing a compiler, there are several critical design decisions that must be made: should the language be compiled or interpreted? Should it compile down to low-level assembly, or transpile to high-level languages? Should it leverage modern parse tools in place of implementing a custom parsing algorithm? Each of these decisions has a significant impact on the capabilities and performance of the resulting language, requiring careful consideration during the design phase. Although we made many choices when designing and implementing our compiler, there were two that had a particularly large impact on our work. Namely, we decided to target low-level Java Bytecode over transpiling to a high-level language (such as JavaScript), and additionally we chose to implement the Recursive Decent Parser instead of using modern parse tools.

Our decision to target low-level Bytecode was largely influenced by the accompanying flexibility. One of the major disadvantages of transpiling to a high-level language is the added constraints imposed by the target language's feature set. This is not the case when compiling to low-level assembly or Bytecode, which allows one to develop their own features from the ground up at the cost of added complexity. This was a trade-off that we were willing to make for the sake of complete control over the implementation of CatScript's features. This was especially beneficial when implementing advanced features such as control structures, scoping, and the static type system which would have otherwise required careful modification to satisfy the scheme of a transpilation. Additionally, targeting low-level Bytecode afforded us the opportunity to develop a more complete understanding of the compilation and code generation processes, all the way down to the system level.

Our second major design decision was to implement the Recursive Descent parsing algorithm instead of using modern parse tools. Although parse tools make it easy to implement a language's grammar, they do so through a complex abstraction that combines many parsing steps making it difficult to understand and modify language features. The Recursive decent parsing algorithm, however, does not impose these constraints. In fact, the algorithm is quite simple to understand and affords complete control over the implementation of the language's grammar and its feature set. This simplicity and control were highly desired for our project, allowing us to develop a deeper understanding the parsing processes while simultaneously reducing code complexity and enforcing a stronger separation of concerns.

7 Software Development Life Cycle Model

For this project, we adopted the Test-Driven Development (TDD) software life cycle model. Spawned from the Agile and Extreme programming methodologies, this model focuses on the rapid development of functional software features by satisfying predefined, atomic unit tests. The strict adherence to these tests promotes the development of robust and functional code that is easy to quantify in the context of project requirements. Generally, this methodology consists of two phases: composing unit tests according to feature specifications and updating the code to resolve those unit tests. For this project, we assumed the responsibility of the second phase; writing code that addresses a suite of provided tests to drive the implementation of each component in the compilation process.

This approach proved extremely effective in the context of developing a compiler, which is an inherently large and complex software project. Specifically, the atomic nature of TDD allowed us to focus on each individual component of the compiler, resulting in improved comprehension and understanding of the behavior and functionality for each stage in the compilation process. Additionally, the structured nature of these tests in combination with the abstraction of boilerplate code allowed for a more rapid development life cycle that served the limited timeline for this project. Although other Agile methods such as Scrum and Feature Driven Development are similarly equipped for rapid development, they fail to offer the same level of structure as TDD does, making them less suitable for classroom applications.

Of course, the Test-Driven Development model is not without its limitations. Perhaps most notably, TDD requires tests to be written before implementing the corresponding code. This makes TDD difficult to implement for real-world products where feature selection and requirements are constantly changing. Additionally, Test Driven Development introduces the overhead of having to update tests as these features change, further contributing to this lack of real-world applicability. Though these were not problems for our purposes, it might have been beneficial to consider other software life cycle models that better prepare us for real-world, dynamic development pipelines.