

Catscript Scripting Language

Thesis Portfolio

Collin Wright

CSCI 468 - Compilers, Capstone Project

9 May 2022

Table of Contents

Table of Contents	1
Program	2
Teamwork	2
Design Pattern	3
Technical Writing	3
UML	4
Design Trade-Offs	5
Software Development Life Cycle Model	6

Program

This document outlines my CSCI 468 (Compilers) Capstone Project: CatScript. Catscript is a simplified scripting language used to demonstrate the fundamentals of writing a language from scratch and the inner workings of compilers. This iteration of the CatScript language was completed by yours truly, Collin Wright. CatScript's starting code and sample tests were provided and outlined by CSCI 468 professor, Carson Gross. The source code for the CatScript language is included with the submission of this document in the file "source.zip."

Teamwork

The bulk of this project was done individually by myself, Collin Wright. At the tail end of the semester however, to abide by the Capstone's teamwork requirement, a portion of this project was completed as a team with fellow student, Nathaniel Priestley. Together we collaborated between our two renditions of the CatScript language, helping each other with the final developments of the language. Furthermore, we each acted as an official software tester for the other's CatScript language. Each member contributed specified tests to verify the soundness of our programs under unique, rare, or otherwise complicated input circumstances, to make sure our code was functioning at complete success. Our tests were shared between programs to see if any test cases revealed shortcomings in our code. In cases where tests were failed, the code was revisited and fixed to account for these oversights. Finally, Nathaniel and I also collaborated to generate proper documentation for each other's capstone projects. As per assignment requirements, I completed a fully fledged documentation for Nathaniel's CatScript language, while Nathaniel provided the documentation for mine (see **Technical Writing** below). Thorough communication, scheduling, and design planning were utilized throughout the team-based portions of this project to ensure an organized and successful collaborative process.

Design Pattern

The only explicit use of a software design pattern in CatScript is the use of the Flyweight pattern (also known as Memoization) in CatScript's type storage process. Memoization is most often used to reduce the amount of similar objects that need to be created, by mapping identical created objects to a single immutable object. In this pattern, a hash map is used to store instances of desired generated objects. If the same object is requested multiple times, then rather than creating a duplicate, the map is accessed to share the stored immutable copy. In CatScript, this technique is used to reduce the amount of list types that are created and stored. Before implementing this pattern, when a new list was created, a new list type (corresponding to the type of the list's contents) was created and returned as that object's type. If another list of the same type was created, the process would simply happen again for that list, resulting in two identical type objects being created and stored separately for each list. By deploying the Flyweight pattern, each new type of list was reduced to only a single instance to create, store, and provide to relevant objects. A concurrent hash map was created alongside the "getListType" method (in the "CatscriptType.java" file). Whenever a list type was requested, the hash map was referenced to see if that type of list had been previously implemented. If that type already existed, the immutable list type stored in the map was returned. Otherwise, the first instance of that type was created and added to the map as a reference for the next time that type was needed.

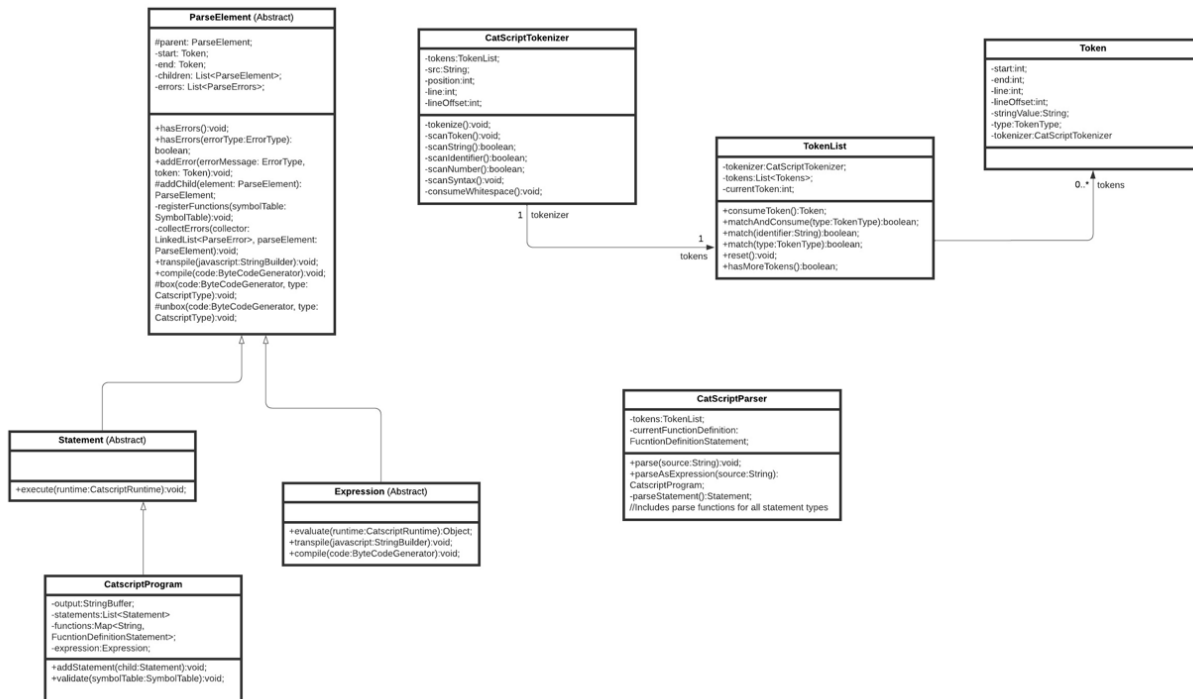
Figure 1: Flyweight Pattern used to eliminate duplicate list types in "CatscriptType.java"

```
//Uses Memoization to handle list types without duplicate storage
static ConcurrentHashMap<CatscriptType, ListType> cache = new ConcurrentHashMap<>();
public static CatscriptType getListType(CatscriptType type) {
    ListType returnType = cache.computeIfAbsent(type, catscriptType -> new ListType(type));
    return returnType;
}
```

Technical Writing

A full technical documentation of the capabilities and usage of CatScript was written for each rendition of the language. As described above (see **Teamwork**), the CatScript project required each student to write and provide the technical documentation for their partner. For this portion, I wrote a full technical overview of Nathaniel Priestley's CatScript grammar. Furthermore, the full documentation of my CatScript language, written by Nathaniel, is included with the submission of this document in the file "Catscript Documentation_Wright Priestley_2022.pdf"

Figure 2: Simplified UML Class Diagram of CatScript Code Structure



Above is a simplified version of the full UML Class Diagram for CatScript. A full version of this diagram (including concrete Statement and Expression types) was included with the submission of this document in the file “CatScript UML Class Diagram.jpeg.” The basic structure of CatScript revolves around the “Parse Element” class. This abstract class accounts for all units of instructions that can be instructed in, read, and executed as CatScript, including “Statements” and “Expressions” (these abstract classes are then extended by an array of more specific statement/expression types, see documentation and full class diagram for more information). One type of Statement, the “CatScriptProgram” encapsulates a full list of Statement and Expression Catscript instructions that constitute a full executable program of Catscript code. All Parse Elements include the basic inherited functions `transpile()`, `compile()`, and either `execute()` or `evaluate()`. These functions provide the framework for each concrete Parse Element to be transpiled, compiled, or processed to an executable output. With this Parse Element structure, each parse element is able to resolve itself within the evaluation and compilation phases of the compiling process. From there, the only missing component is to generate those Parse Elements. When input is given to be read as CatScript, it can first be sent to the “CatScriptTokenizer” class. This class reads in a raw string and divides the input into a list of tokens (individual terms of the language). This Token List can then be sent to “CatScriptParser” which validates that the tokens are a valid CatScript syntax, and produces the Parse Elements necessary for evaluation and compilation.

Design Trade-Offs

Writing a programming language comes with a near endless supply of design decisions. Due to the assignment-based nature of this project however, many of the design decisions for CatScript were inherent to the course guidelines. CatScript's design was, for the purposes of the class, predetermined by Professor Gross. The only major freedoms in design choices were in code structure, which was kept to simple method executions and minimal elaboration to focus on passing software tests and to avoid code-threatening interconnectivity. That being said, major decisions made by the course outline are noteworthy and worth defending.

The most significant design decision was to write the CatScript parser using recursive descent rather than a parser generator. Parser generators are typically the standard for writing language parsers (particularly in academia). Yet in this project, this method was rejected in favor of recursive descent. While generators are more code-efficient than CatScript's Recursive Descent parser due to their automated method call parsing systems, these same systems shield the true inner workings of grammar parsing. Despite having to write more code, writing a recursive descent parser gives a much better interior view of how parsing works, given that grammar has to be handled by hand. Additionally, recursive descent is much simpler, more practical (in lieu of being more theory-heavy), and thus is logically easier to interpret and generate.

Another significant design decision made by the assignment guidelines was to exclude the use of the Visitor Pattern. Typically, the Visitor Pattern is used to separate pieces of software wherever possible, to avoid excessive and dangerous interconnectivity, allowing each function to be removable and individually changeable without jeopardizing the rest of the program. This pattern was intentionally ignored for simplicity's sake. While the Visitor Pattern is arguably more sound, instead including key functionalities as methods within their respective relevant nodes allows for a tighter, more navigable and readable experience. By including functions within each parse element class, all relevant behaviors for each type of parse element are centralized in the same place. This makes debugging, problem solving, and understanding the functionality of each parse element immensely easier to process at the expense of more interdependent code.

Software Development Life Cycle Model

The development of CatScript was done entirely under a Test-Driven Development model. A sample of starter code was provided as a skeleton of the CatScript language, alongside a set of functional tests provided by Professor Gross and my partner, Nathaniel Priestley. These tests acted as landmarks of success for each portion of the language. Running the tests on my program would either result in a success or a fail. Success was an indicator that the code is working properly and giving the correct outputs. A failure or error-catching result indicated that something in the code was not correct, and required some tweaking to fix the issue. The development cycle followed four stages of development (covering the four major components of the language: tokenizing, parsing, evaluating, and compiling) in which tests were run, and code was written and debugged for each failing test until all tests were passing.

Overall, this development cycle was efficient and advantageous. The condensed and modular organization of the tests was pivotal to the organization of my work. Organizing worktimes by reaching goals of a certain number of succeeding tests was a great way to structure the (otherwise overwhelming) workload. The tests always provided a next step in the work, as well as an overarching end goal. Debugging each test through the language generation process was helpful in teaching me exactly what I was doing with each line of code, enhancing my understanding of language compilation. I felt that specifically practicing my debugging skills stood out as a valuable experience I gained from this project that will stick with me throughout my career.