

# CSCI Capstone

Jacob Hofer

## Section 1: Program

The source code for the program can be found at

<https://github.com/CodeAX2/csci-468-spring2023-private/blob/main/capstone/portfolio/source.zip>

and is also included in the submission with this document labeled source.zip.

## Section 2: Teamwork

In order to properly ensure the quality of my code, I worked along with a partner who was responsible for testing my code as well as writing documentation. Testing my code was done primarily through automated tests, written both by Prof Gross and my partner.

The tests written by my partner are located in the CatscriptPartnerTest.java file. These tests covered sections such as Java Bytecode compilation, for loops, functions returning long strings/printing those strings, and unary operators on large integers. These tests helped with the development process as well as identifying bugs and other issues within the code.

My partner also provided documentation on the CatScript programming language I developed for this course. The documentation covered how to program in CatScript by explaining the various expressions and statements. It also provided insight into the grammar of the language and related the execution to other known programming languages.

My own work on the project consisted of implementing the tokenizer, parser, validator, evaluator, and bytecode compiler for the CatScript language. The parser is used to create an in-memory parse tree from the human readable source code. This tree is then validated and checked for errors, such as type mismatches, missing arguments, etc. The evaluator uses a Tree-Walk algorithm to dynamically run the processed code, whereas the bytecode compiler turns the script into static JVM-compatible bytecode.

## Section 3: Design Pattern

One major design pattern that was used when writing the CatScript compiler was the memoization pattern. This pattern was used to cache results of calls to the CatscriptType.getListType method. This method is responsible for generating a new CatscriptType representing a list whose contents are of some specified type.

For example, calling `CatscriptType.getListType(CatscriptType.INT)` would create a new type representing a list of integers.

However, this method may be called quite often through the process of compiling. Whenever this method is called, we save the newly created list type to a map. Then if the method is called again with the same input parameters, rather than create a new list type, we simply return the existing list type found in the map.

This pattern was used as it reduces heap memory consumption, allows the method to run faster, and removes potential bugs, as any `CatscriptType` representing a specific list type will always be the same reference. Directly returning a new list type each function call would not provide these benefits.

The code for the design pattern is included below (extra class code is not included):

```
private static HashMap<CatscriptType, CatscriptType>
    cachedListTypes = new HashMap<>();

public static CatscriptType getListType(CatscriptType type){
    if (cachedListTypes.containsKey(type)) {
        return cachedListTypes.get(type);
    }
    CatscriptType listType = new ListType(type);
    cachedListTypes.put(type, listType);
    return listType;
}
```

## Section 4: Technical Writing

The technical writing document contains the CatScript documentation written by my partner. It has been converted from markdown format and is included on the following pages.

# Catscript Guide

---

Catscript is a fun little project that me and the boys from CSCI 468 at MSU, taught by the legendary Carson Gross. This document is basically my guide designed to help you understand how to use the version of Catscript my amigo Jacob whipped up.

## Introduction

---

Catscript is not hard to figure out. Especially compared to other programming languages (Looking at you C) Below you can see a simple example of how it do what it do.

```
var x = "foo"
print(x)
```

## Features

---

### Expressions for Noobs

Basically expressions are bits of code that turn into a nice value. Think like your ints, strings etc... It's really just another way to say this is a thing. So as long as it gets it turns into a value at the end. We've got a few neat types of expressions that I'll go ahead and explain next.

#### Parenthesis

Parenthesized expressions are an expression inside parentheses () it just means you process everything inside it then treat that result as the value. Good for some math and so on

#### Adding

Additive expressions are expressions with two values and a + or - in between them. If they're both ints then it just adds them like you'd expect. If either is a string it just concatenates them, because that's common sense. (LOOKING AT YOU C)

#### Factoring

Factor expressions are aight. They basically are the same as additive expressions but instead of + and - they have \* or / and they don't do anything if you give them strings.

#### Equality

Equality expressions spit out a boolean value, and take the form != or ==. == checks if they're the same and != checks if they're different

#### Unary

Unary expressions are basically the coding equivalent of the uno reverse card. We got a couple variations - and not. - makes a number negative. Obviously. Not inverts a boolean value. Pretty basic stuff

#### Identifiers

Used to refer to variable or function names

#### Literals

Literal expressions are just literally (I know I know comedian of the year) expressions. It's like just a number or a word or so on. We got the standard rogues gallery for this, Ints, Strings, Booleans, null (It's just null) and lists, which are built like this [value, value]

### Statements

The cooler older brother to expressions, statements do some funky stuff. Taking (Actions) and demonstrating (Initiative) These bad boys are the language elements she tells you not to worry about. Absolute chads.

#### Variables

See I told you it was cooler. We call these variable statements, but they're basically the definition of the variables. Meaning this is the more important part as "Identifiers" Are dependent on them. Simply built better. It's just var name = value. You can pick names and values.

#### Assignment

Assignment statements are the badass order giving drill sergeants of catscript. This is this, that's not a question it's an order. Love to see it. basic as this: thing to be ordered = thing it is from now on.

#### If statements

Baby's first program, these bad boys are where things start looking like coding, with a hearty injection of both logic and complexity, You get to say

```
if (condition) {  
    stuff to do  
}
```

with an optional

```
else if (condition two) {  
    other stuff  
}
```

and an end with

```
else {  
    just do it  
}
```

those conditions take boolean values kids, and these run like a dream

## For loops

You already know how it is,

```
for (counter in some stuff) {  
    do some other stuff  
}
```

counter is a local variable up to you, and some stuff is a list of your choice

## Print statements

```
print(Your stuff)
```

Need I say more?

## Functions

Hey look ma, I made it! I'm a real programming language now! If you don't live under a rock and aren't a masochist, you love these guys as much as I do. just about essential these days, they come in the format:

```
function put_the_name_here(variable_name : variable_type, you_can_just_keep_adding_these) : return_type {  
    all that content baby  
}
```

This is just a function declaration but it gets the job done. Check out this next bit

## Return

Function's best friends, these only work inside them but with a simple `return your_stuff` you can ensure any function call results in a nice value of your choice, a gentleman's pick if you will

## Types

We got the types baby, check it out:

```
void  
null  
int  
bool  
string  
list<type>  
objects
```

## Parsing

This bad boy uses recursive descent, the best option according to our professor, and one even I can understand. It compiles into java bytecode because it's just that

cool.

Here's the grammar

```
catscript_program = { program_statement };

program_statement = statement |
                    function_declaration;

statement = for_statement |
            if_statement |
            print_statement |
            variable_statement |
            assignment_statement |
            function_call_statement;

for_statement = 'for', '(', IDENTIFIER, 'in', expression ')',
               '{', { statement }, '}'

if_statement = 'if', '(', expression, ')', '{',
              { statement },
              '}' [ 'else', ( if_statement | '{', { statement }, '}' ) ];

print_statement = 'print', '(', expression, ')'

variable_statement = 'var', IDENTIFIER,
                    [ ':', type_expression, ] '=', expression;

function_call_statement = function_call;

assignment_statement = IDENTIFIER, '=', expression;

function_declaration = 'function', IDENTIFIER, '(', parameter_list, ')' +
                      [ ':' + type_expression ], '{', { function_body_statement }, '}'

function_body_statement = statement |
                         return_statement;

parameter_list = [ parameter, { ',' parameter } ];

parameter = IDENTIFIER [ , ':', type_expression ];

return_statement = 'return' [, expression];

expression = equality_expression;

equality_expression = comparison_expression { ("!=" | "==") comparison_expression };

comparison_expression = additive_expression { ">" | ">=" | "<" | "<=" } additive_expression;

additive_expression = factor_expression { ("+" | "-") factor_expression };

factor_expression = unary_expression { ("/" | "*") unary_expression };

unary_expression = ( "not" | "-" ) unary_expression | primary_expression;

primary_expression = IDENTIFIER | STRING | INTEGER | "true" | "false" | "null" |
                    list_literal | function_call | "(" , expression, ")"

list_literal = '[', expression, { ',', expression } ']';

function_call = IDENTIFIER, '(', argument_list , ')'

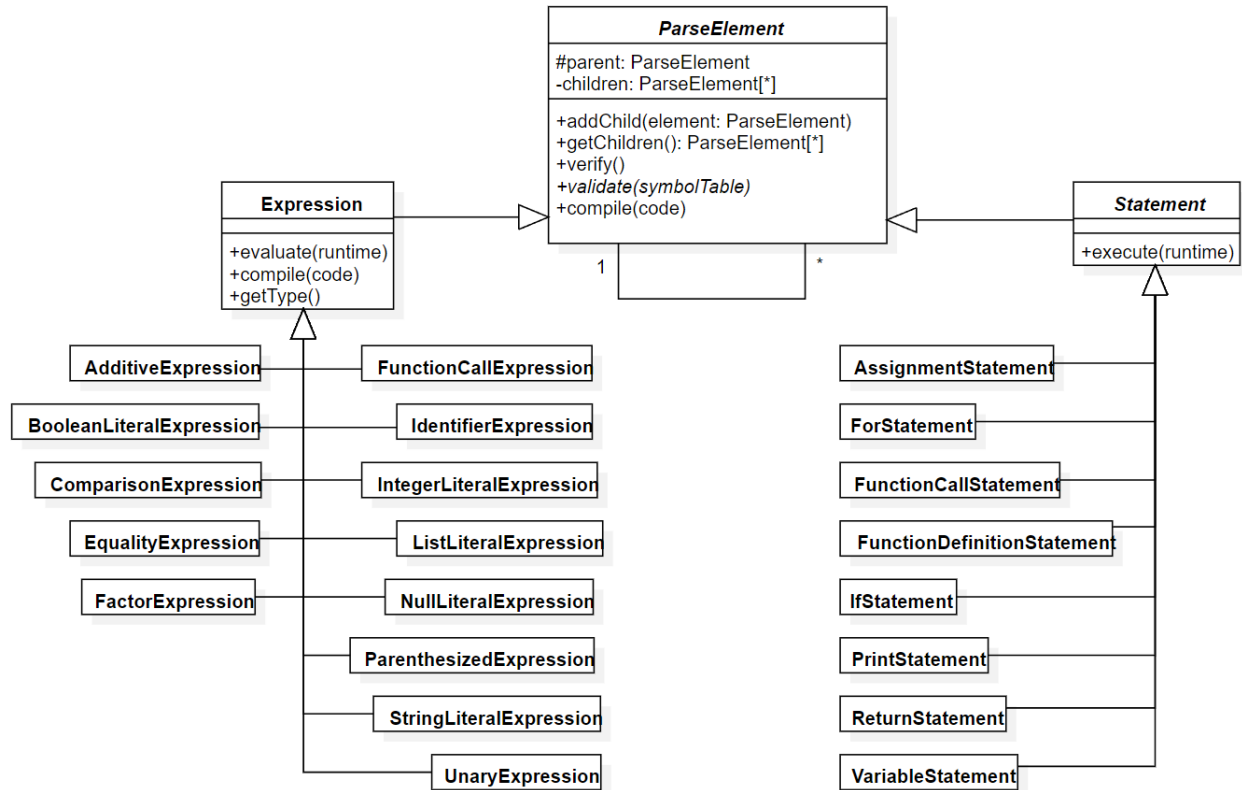
argument_list = [ expression , { ',', expression } ]

type_expression = 'int' | 'string' | 'bool' | 'object' | 'list' [, '<' , type_expression, '>']
```

You can evaluate it at runtime using a tree walk interpreter too

## Section 5: UML

The UML diagram below shows the relationship between the various ParseElements found in the CatScript Compiler:



## Section 6: Design Trade-Offs

One of the major design trade-offs was the decision to use a recursive descent-based parser rather than using a parser generator.

A parser generator takes the direct EBNF grammar and creates Java classes capable of directly parsing the input grammar. Initially, this may seem like the obvious choice for creating a parser, as creating an EBNF grammar is relatively trivial, and then code that would otherwise take weeks to create is done in a matter of seconds. However, two main problems parser generators face are grammar limitations and integration.

In CatScript, our grammar is technically not Context-Free. Function definition statements hold a collection of their body statements, that are equivalent in form to statements outside of functions. However, function bodies can allow for return statements to appear anywhere, including within nested statements (such as loops or if clauses). This provides difficulty when creating an EBNF grammar, as it forces all statements within functions to be technically different from their counterparts outside of functions, despite having identical structure. This will also

translate into generated code, as now the grammar requires twice as many statements, one definition for inside functions, and another for outside.

Parser generators are also notoriously difficult to integrate with other code. Since our compiler supports both Tree-Walk interpretation and bytecode compilation, it would have been quite difficult to integrate automatically generated parser code into the overall architecture.

Therefore, we used a recursive descent algorithm to perform our parsing. This allowed us to use non-Context-Free grammars as well as specifically design the structuring of our parse element classes and how they would interact with the rest of the code. While this did require much more handwritten code, recursive descent is a straightforward algorithm that made the process easier.

## Section 7: Software Development Lifecycle Model

Our lifecycle model was based on Test Driven Development. Unit tests were provided by Prof Gross that covered in-depth all major areas of the codebase, including tokenization, parsing, evaluation, and bytecode compilation.

These tests provided a quick way to determine not only if the code was working or not, but also problematic areas within the code. Since each test covered a specific area, such as if statements or additive expressions, finding the source of the issue was typically a quick process.

Tests also provided a streamlined process for developing the code as a whole. Since tests were separated into independent sections, it was simple to determine which section of the codebase to develop next. Once all of the tokenization tests passed, the next section was all about parsing, then evaluation, and so on. Within each overarching section were subsections that further streamlined the process in a similar fashion.

Once all the tests were passing, it was clear to see that the code worked for the large majority of cases. Some extra work did still need to be done to handle edge cases not covered by the unit tests, which were typically discovered through manual testing and writing custom code in CatScript. In addition, some areas of code were simply marked with a TODO comment, rather than given an explicit test. This did provide some difficulty as some crucial features, such as list component type inference, would be missing despite all of the tests passing.