CSCI 468 Compilers Capstone Portfolio

By: Fletcher Phillips
Montana State University
Gianforte School of Computing

04/30/2024

Spring 2024

Partner: Keegan Gaffney

Section 1: Program

The source code has been zipped in source.zip and added to the capstone folder inside the GitHub repository.

Section 2: Teamwork

For teamwork in this project, my partner worked as a tester for the project and contributed the following items to my project: Three tests found in PartnerTest.java(found at src/test/java/edu/montana/csci/csci468/demo/PartnerTest.java) and the documentation for the Catscript programming language. I was responsible for all of the implementation of all the code that was required for the compiler and to pass all tests required to pass. I also did the same for my partner. I did my part of the teamwork by creating three tests and documentation for the Catscript programming language and giving them to him.

The documentation my partner wrote for my project is shown in Section 4: Technical Writing of this document. As we created tests for eachothers code we made sure that we would handle as many edge-cases as possible. These tests were also made to be very detailed with an adaquete amount of assertions. By having consistent teamwork with my partner, I was able to make sure that the documentation was adaquete for the Catscript langauge so that a developer could look at it and write code with it. The tests were able to certify how robust the code was. The time spent between us was 95 percent me writing the parser and my partner was 5 percent for doing the tests and the documentation.

Section 3: Design Pattern

In the Catscript language compiler project, the design pattern known as Memoization was used. Memoization is the pattern used to optimize the performance of certain functions by caching the results of the computation and retrieving and reusing the cached result if the same input is called for a function more than once. In Catscript, Memoization is implemented by using a data structure called a hashmap storing key:value pairs of the CatscriptType and ListType found in the method getListType located on line 36 in the file that can be found by navigating to src/main/java/edu/montana/csci/csci468/parser/CatscriptType.java. Here is the code followed by the explanation of it.

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

- `cache` is the HashMap that stores the previously computed results from the method.
- `getListType` is the method that takes the `CatscriptType` as its input.
- When `getListType` is called with `CatscriptType`, it will check if the result for that type is already stored in the cache HashMap.
- If the result is found in `cache` `(listType != null)`, it returns the cached result.
- If the result is not found `(listType == null)`, it computes the result, stores the result in the cache, and then returns the result.

The idea for using Memoization in the getListType method here is so that redundant and possibly expensive computations are avoided when the same type of CatScriptType is given as an argument to getListType. By using Memoization, I ensure that only one single instance of ListType will be created for each type, therefore decreasing the amount of space used. Comparatively, while I could have coded the caching into each part of the code where getListType was called, it would have been redundant and decreased the readability and reusability of the code. Instead, by using the Memoization pattern by centralizing the caching logic in the method itself and not coding the logic directly into each call, this Memoization design pattern approach ensured that the code remained optimized for storage, reusable, readable, and scalable if changes needed to be made in the code.

Below are some more details on implementation and benefits:

- Memoization was implemented by using a hashmap to store the previously computed result. The mapping was the CatscriptType input to the ListType output.
- By centralizing the caching to getListType, code complexity was decreased and allowed for more modular code.
- Decreased storage footprint created by the Catscript compiler.

Section 4: Technical Writing

The technical document associated with this project is the documentation of the CatScript programming language. The documentation for the Catscript programming language is provided below:

Catscript Guide

This document should be used to create a guide for Catscript, to satisfy capstone requirement 4

Introduction

Catscript is a simple scripting language. Here is an example:

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

Features

Type Literal Expressions:

Catscript has a few different types of values:

int - integers are whole numbers

```
1, 2, 3, 4, …
```

string - strings are characters enclosed by quotations

```
“foo”, “bar”, “Hello World!”
```

bool - booleans are true or false conditional values

```
true, false
```

object - Any type of value

```
1, “foo”, true
```

list - A list of any of these types, can have a list within a list.

```
[1, 2, 3] // integer typed array (list<int>)
[3, true, “foo”] // generic array (list<object>)
```

null - null value

```
null
var x = null
```

Statements

For Statements:

For statements are a fundamental aspect of Catscript. They work by looping through a list of values and executing a block of code for each value in the list. The syntax for a for statement starts with the "for" keyword, followed by the variable name and list to loop through in parenthesis, followed by the block of code to execute for each value in the list. The name of the variable in the parenthesis is the name of the variable that should be used in the loop to reference the current value in the list. The block of code enclosed in curly braces also has its own local scope, so any variables declared within the block will only be accessible within that block.

For loops are the only type of loop in Catscript but they are very powerful. The difference between Catscript for loops and Java for loops is that Catscript has a more simple syntax, but are therefore easier to understand and write. For loops in Catscript are limited to only iterate over lists, and are not able to iterate over other types of values. They can however handle lists of any type, including lists of lists, and generic lists. This means that you can have the same code run with various different types. A very simple example of a for loop in Catscript is shown below:

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

Output:

```
1
2
3
```

If Statements:

If statements are another fundamental statement in Catscript as well as many other languages. If statements take in some expression and evaluate it to true or false. The value of that expression then determines what code is run. The syntax for if statements in Catscript is nearly identical to Java. It starts with a "if" token, followed by the expression to be evaluated in the parenthesis. After that, the block of code to be run if the expression evaluates to true is next wrapped in curly braces. Catscript if statements can also have an optional "else" block that is run if the expression evaluates to false. The syntax is also identical to Java where after the true block, there is an "else" token, followed by the code to be run wrapped in curly braces.

As said before, if statements in Catscript are nearly identical to if statements in Java, with the only difference being the lack of support for "else if" blocks. Normally these blocks would take in additional expressions that could be evaluated to true, and if false, would continue to the next "else if" or "else" block. To make up for this limitation, Catscript supports the nesting of if statements to allow for these more complex use cases. You can see a simplified example of an if statement in Catscript below:

```
if (true) {
   print(1)
} else {
   print(2)
}
```

Output:

```
1
```

Print Statements:

Print statements enable the developer to get output from their Catscript program. You can pass in any value from your program and have it printed to the console. The syntax for this statement is very simple and identical to Python. You start with the "print" token, followed by some expression in parentheses. This expression is what will be printed out to the console. This expression can be as simple as a plain string like "Hello World", it could be an identifier that evaluates to some value, or can even be a more complex expression such as a concatenation expression like `"Hello" + " World"`.

Printing is the only means to get information about how your program runs since there is no debugger, or way to interact with files in Catscript. This makes the print statement arguably the most important of all the statements as without it there isn't much of a point to the program. As said earlier Catscript's

print statements are essentially identical to Python's and therefore are very easy to use. A simplified example of their use can be seen below:

```
print(1)
print("Hello")
print(null)
```

Output:

```
1
Hello
null
```

Variable Statements:

Variable statements in Catscript are how you store values. In order to store a value in a program you first need to create a variable and give it a name. Variable statements offer a very simple way to do this by using the "var" keyword, followed by the name of the variable, an `=`, then the value you want to assign to that variable. Catscript is a bit interesting however because these variables can be either explicitly or implicitly typed. To implicitly type a variable, you use the syntax described above. However, if you want to assign an explicit type to a variable, you can add an optional `:` after the variable name, followed by the type literal you would like that variable to be.

The syntax for Catscript's variable statements are pretty similar to javascript/typescript, although they are more limited. Catscript variable statements are limited to the following types: strings, integers, booleans, Objects, lists, and null. Unlike in javascript, function definitions cannot be assigned to variables as well as some other more complex statements. Variables are only accessible in the scope they were defined in. If a variable was defined in the global scope it will be available across the entire program, whereas if it was defined in an if statement, it will only be accessible in that block of the if statement. A simplified example of Catscript's variable statements can be seen below:

```
// implicit type declaration
var x = 1
print(x)
```

```
// explicit type declaration
var y: int = 2
print(y)
```

Output:

```
1
2
```

Assignment Statements:

Assignment statements are the other aspect of storing values in Catscript. The main difference between variable statements and assignment statements in Catscript is that variable statements define new variables whereas assignment statements modify variables that have been previously defined in a variable statement. The syntax for these assignment statements is very simple. It is simply the name of the variable followed by an `=` and then the value to change it to.

Assignment statements by nature are more limited in their use compared to variable statements. For starters, the variable you are attempting to change must already be defined. On top of this, that variable must be in the proper scope, meaning you can't assign a variable in one function, that was defined in another. Another interesting limitation of these statements is that if you explicitly typed a variable when defining it, the value you are assigning to that variable MUST be of that type. So if you were to create an int variable called x, you could not write `x = "Hello World"`. A simplified example of assignment statements can be seen below.

```
var x = 1
x = 2
print(x)
```

Output:

```
2
```

Function Declarations:

Function declaration statements are the first of three function-related statements. The declaration of the function is the first step when it comes to using functions in Catscript. The syntax starts with the "function" keyword, followed by the name of the function, parameters, and their types in parentheses, an optional return type, then the body of the function wrapped in curly braces. The name of the function is what is used to reference that function in other places in the code. The parameters, are a set of local variables that are accessible to the function body or in other words, are in the same scope as the function body. The return type determines what type the function call will evaluate to be. Finally, the body is where all of the logic unique to your function is handled. This logic has its own scope that is different from the rest of your program (besides global). This allows the function to run cleanly from a fresh slate each time.

There are a few quirks you should note when using function declaration statements in Catscript. First, the function declaration itself does not evaluate anything. To evaluate a function, you need to use a function call statement which we will cover later. Catscript functions can also be recursively called, meaning a function can call itself from inside itself. This is useful for some applications such as calculating the factorial of a number. A very simple example of a function declaration statement can be seen below:

```
function foo(x: int) {
   print(x)
}
```

Output:

```
No output
```

Return Statements:

Return statements are the second of three function-related statements in Catscript. The purpose of these return statements is to simply return a value from a function call. In other words, the return statement allows a function call to evaluate to some value that is determined in the function during runtime. The syntax for this statement is simply the "return" token, followed by some expression. This expression will be what is evaluated when a function call statement is used.

Something to note about these return statements in Catscript is that they can only be used within the scope of a function declaration. Also, it was said earlier that the "return" token is followed by an expression, however, this isn't always the case. If a function has a return value of void (or one is not set), you can simply use the "return" token to exit out of the function. This can be done to exit a

function early based on some logic in the function itself. A simple example of a return statement can be seen below:

```
function foo(x: int) : int {
   return x + 1
}
print(foo(1))
```

Output:

```
2
```

Function Call Statements:

Function call statements are the final of the three function-related statements in Catscript. These statements are used to actually execute the code within a function declaration. The syntax for these statements is simply the name of the function followed by a comma-separated list of expressions wrapped in parentheses. These expressions represent the arguments that are passed into the function and therefore need to match the types and order of the parameters in the function declaration.

There are a few things to note about function call statements in Catscript. First, the expressions passed in as arguments are evaluated before being passed into the function. This means that you can pass in complex expressions such as other function calls as arguments. Another thing to note is that these function calls can be used as expressions themselves. This means that if the function has a return value, the function call will evaluate that value. This allows you to use function calls in other statements such as variable statements or even other function calls. Functions are a fundamental aspect of Catscript and other programming languages as they allow code to be reused and organized logically. A simple example of a function call statement can be seen below:

```
function foo(x) {
   print(x)
}
foo(1)
```

Output:

```
1
```

Expressions

Equality Expressions:

Equality expressions in Catscript are used to compare values if they are equal to each other or not. You can use `!=` (not equal) or `==` (equal). They evaluate to true or false. For example:

```
1 == 1
true == true
null == null
1 != 1
```

Output:

```
true
true
true
false
```

Comparison Expressions:

Comparison expressions in Catscript are used to compare values to each other. You can use greater than(`>`), less than(`<`), greater than or equal to (`>=`), or less than or equal to (`<=`). They evaluate to true or false. For example:

```
1 > 1
1 >= 1
1 < 1
1 <= 1
```

Output:

```
false
true
false
true
```

Additive Expressions:

Additive expressions in Catscript are used to add or subtract values. They can also be used to concatenate strings. They will return the value evaluated by the expression. For example:

```
1 + 1
1 - 1
1 + 2 + 3
“Hello” + “ World!”
```

Output:

```
2
0
6
Hello World!
```

Factor Expressions:

Factor expressions in Catscript are used to multiply or divide values. You can multiply by using `*` and divide using `/`. For example:

```
1 * 1
1 / 1
1 * 2 * 3
```

Output:

```
1
1
6
```

Unary Expressions:

Unary expressions in Catscript are used to negate a value to its opposite. For example:

```
-1
not true
```

Output:

```
-1
false
```

Parenthesized Expressions:

Parenthesized expressions in Catscript are used to order expressions. If you have a large expression made up of multiple smaller expressions, you can use this to have those smaller expressions in the order you want. For example:

```
(1 + 2) * 3
1 + (2 * 3)
```

Output:

```
6
9
```

Identifier Expressions:

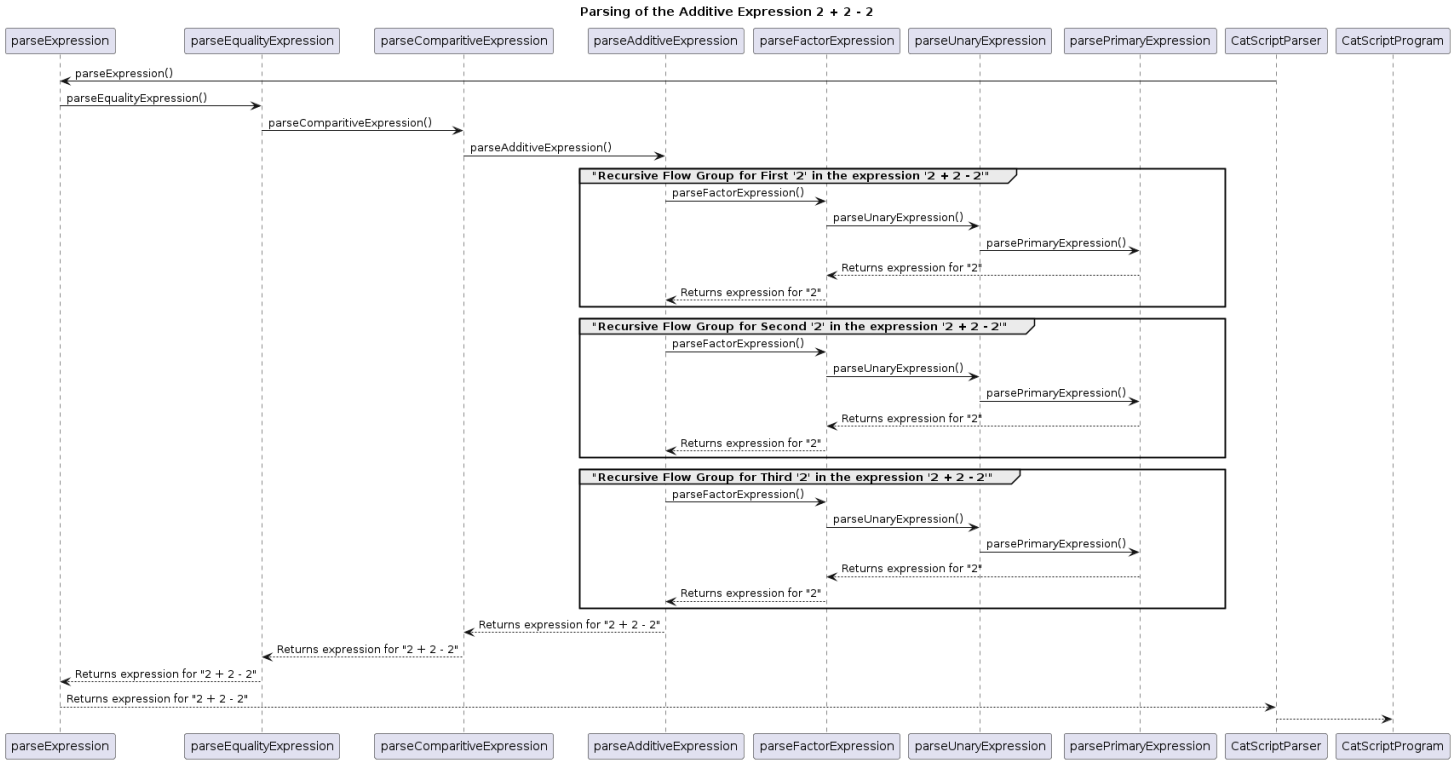
Identifier expressions are the variable and function names in Catscript. Whenever you try to use a variable or function and reference its name, that name is the expression. Identifier expressions are unique in the fact that they aren't defined by any keyword or operator.

Function Call Expressions:

Function call expressions are actually part of the function call statements. The expression is what allows the function call to evaluate to some value, when used in a variable statement.

Section 5: UML

Sequence Diagram for Recursive Descent Algorithm for Parsing Additive Expression of: `2 + 2 - 2`



Explanation of the diagram:

This sequence diagram shows the parsing flow for the expression `2 + 2 - 2` following the recursive descent parser algorithm. The flow begins with the CatScriptParser being the initiator of the parsing by calling the parseExpression. parseExpression then will delegate the parsing process down the hierarchy of the grammar using the following methods and expressions in order to handle the different parts of the expression `2 + 2 - 2` : parseEqualityExpression, parseComparitiveExpression, parseAdditiveExpresion, parseFactorExpression, parseUnaryExpression, and parsePrimaryExpression.

In the expression `2 + 2 - 2` , every '2' is a primary expression. Each primary expression is being parsed individually and in the diagram is represented by the separate groups: Recursive Flow Group for First '2' in the expression '2 + 2 - 2', Recursive Flow Group for Second '2' in the expression '2 + 2 - 2', and Recursive Flow Group for Third '2' in the expression '2 + 2 - 2'. Each group helps illustrate the recursive nature and flow of the parser of calling to the lower-level parsing methods while it breaks down the expression into parts. Taking a look at the first group you can see the flow of how the first '2' is parsed. Parsing starts with the parseAdditiveExpression which then calls the parseFactorExpression, after the parseFactorExpression is done it will call the parseUnaryExpression

to lead to the parsesPrimaryExpression to parse the '2'. Once it is parsed it will flow back up to the higher-level methods of the algorithm to parseFactorExpression which will then return parseAdditiveExpression.

This recursive flow continues for each of the three groups, parsing each of the '2' in `2 + 2 - 2`. Once all groups are parsed, the diagram then illustrates how the additive expression will combine the '2s' using the binary operations. The parseAdditiveExpression returns to the parseComparitiveExpression. The parseComparisonExpression then returns to the parseEqualityExpression which returns to the parseExpression which finalizes the parsing of `2 + 2 - 2`. The parseExpression then returns to the CatScriptParser to continue parsing whatever else it may need to.

Here is the grammar for expressions for reference:

```
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, ")"
```

Section 6: Design Trade-Offs

For the Catscript programming language project, there were two options for generating the parser, either a parser generator or recursive descent. For this project, I implemented a recursive descent algorithm parser instead of a parser generator for several reasons. For this project, recursive descent was chosen because it was fast enough for this language and can express the natural recursive

nature of grammars much more obviously than if I used a parser generator. In the recursive descent parser, the parsing functions all correspond directly to one of the grammar rules laid out for the Catscript language such as the parseExpression() and parseStatement() methods. Going into more detail for parseExpression() I was able to control and have a direct insight into the logic of how my parser operated. For example, by following the grammar laid out, in my code I chose to have parseExpression() as the highest precedence, then it would delegate the rest of the parsing to the more specific methods such as parseEqualityExpression(), onto parseComparisonExpression(), and so forth to the bottom precedence as seen in the example grammar below for these expressions mentioned.

```
expression = equality_expression;
equality_expression = comparison_expression { ("!=" | "==") comparison_expression };
comparison_expression = additive_expression { (">" | ">=" | "<" | "<=" ) additive_expression };
```

By having direct correspondence to the grammar, I was able to have control over how expressions and statements were parsed greater than if I used a parser generator. The code for a recursive descent algorithm is much easier to read and add onto than if a parser generator was implemented instead. Though there were a few trade-offs such as the overhead of implementing a recursive descent parser compared to using a parser generator giving a much faster implementation, the benefits outweighed the drawbacks. If a parser generator had been implemented less code would have been needed upfront, but there would have been a much steeper learning curve to understand a very abstract parser generator such as ANTLR. Looking at the generated code is very difficult to decipher what exactly is going on in a parser generator. Comparatively, when a recursive descent parser is implemented the programmer will know exactly how the parser flows. This level of obscurity with the syntax from a parser generator causes higher levels of confusion for things than it would be if doing it by hand with the recursive descent algorithm I used for my parser.

In the industry today, parser generators are not as widely used as handwritten parsers like the recursive descent algorithm parser that I implemented. This allowed me to gain more valuable knowledge and skills to take into my career and future coding life. Concluding the discussion on trade-offs, deciding to use a recursive descent algorithm parser instead of a parser generator was well thought out and executed. Although it required more effort upfront, the benefits were worth it.

Section 7: Software Development Lifecycle Model

In this project, we used Test Driven Development (TDD). Test Driven Development is the SDLC which involves writing the tests before writing the code. For this project, TDD included Carson Gross writing the tests for the project as a blueprint for how the code we were expected to write should function. I was then responsible for implementing the code written to pass the tests and meet the requirements of the Catscript language compiler. Having these intricately written tests before writing the implementation allowed me to understand that while I write the code what the clear specifications

would be. Using TDD also allowed me to break the project into small maintainable parts which helped with understanding how each part of the project would build on top of the last part. Keeping this modular approach to development allowed the code to be incrementally built and refined as I built on top of each part of the project, which helped lead to much more scalable code.

One of the best parts of using TDD for this project was that by understanding how to use the debugger, I was guided by the debugger to write more correct and robust code that would hold up to the test and future tests. By using the debugger I saw how the code executed line-by-line, letting me see where I could create stronger code that would hold up to future tests and keep the code scalable for future development. An example would be creating the tokenizer, then the parser, eval, and then the bytecode. For one to work properly, the prior tests for each milestone needed to be passed. However, Test Driven Development has its flaws, and it showed in this project. Using TDD gave me a false sense of security because as I was writing the code, it was being written to pass only those tests so I had to believe that as I wrote code it would be able to be reused in different parts of the project and pass other tests. TDD largely involves the person who is writing the tests to be confident that they can cover all edge cases that the program may need to encounter as it executes for TDD to be ultimately effective.

When comparing Test Driven Development with another SDLC model such as Waterfall you can see the flaws that come along with Waterfall. Waterfall is much less flexible to changes to requirements which may slow down development greatly, while using TDD allows for quick refactors to meet any new requirements. Waterfall does not give the immediate feedback that TDD gives. In Waterfall development, you are prone to find your errors only at the time of testing which leads to very slowed down development by having to rewrite code. By using TDD you are given immediate feedback by running the tests set in place to match the requirements allowing for faster and cost-safe development.

Although some overhead came along with using the TDD approach we followed, the benefits of using TDD for the Catscript project outweighed the problems it presented by allowing me to write stronger and more reusable code. Overall, choosing Test Driven Development to be the SDLC model we followed ensured a smooth and directional process to completing the Catscript language compiler project.