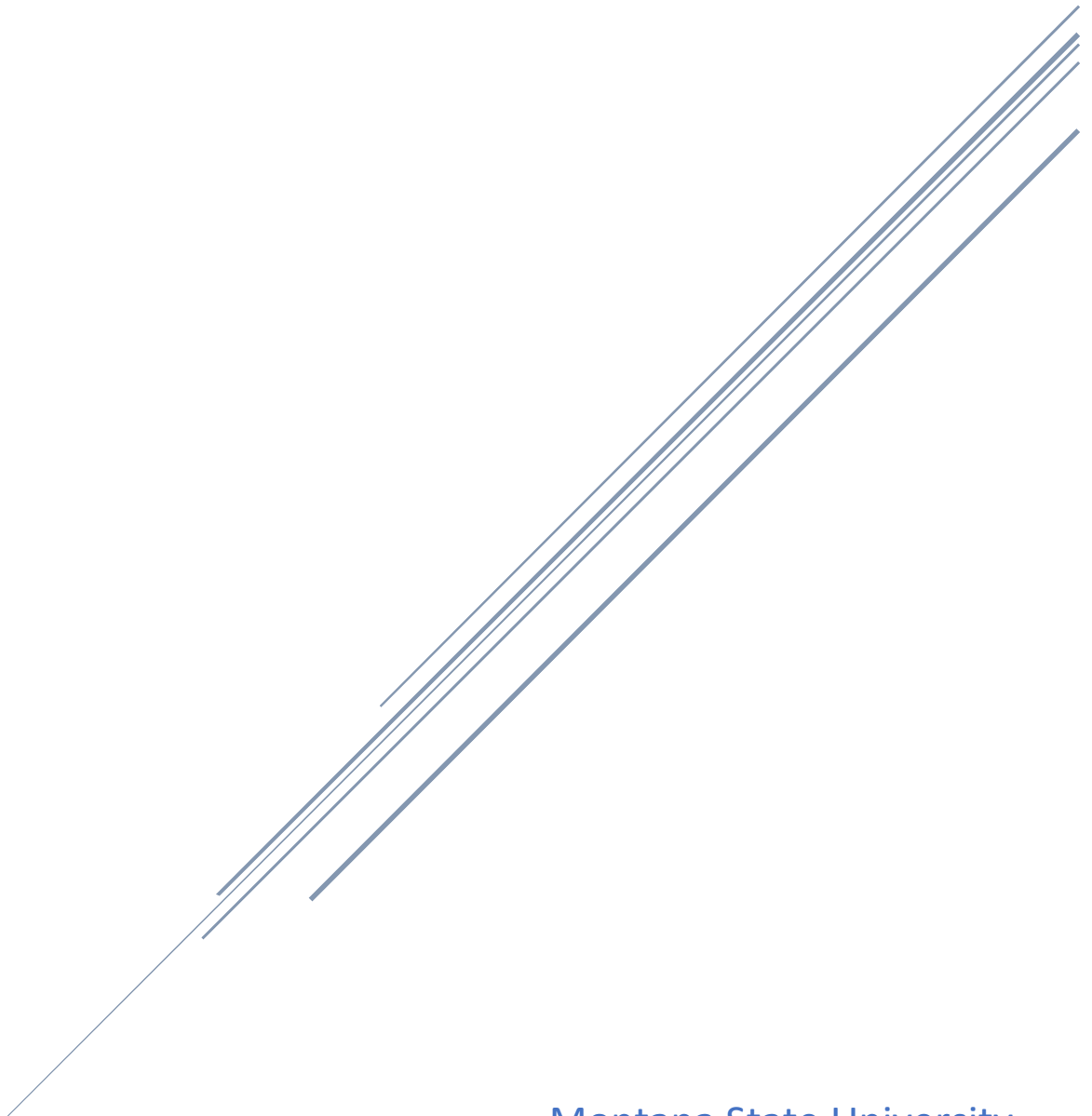


CAPSTONE PORTFOLIO

Jaden Schultz, Megan Weide



Montana State University
CSCI 468 – Compilers, Spring 2023

Section 1: Program

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

Section 2: Teamwork

For this project, team member 1 and team member 2 provided both Section 4 of each other's portfolios, and small test suites for each other to run and see if our compilers work as intended. Team member 1 and team member 2's contributions to this project were equal, or 50% each. Below are screenshots of the previously mentioned test suite passing when run in IntelliJ:

```
1 package edu.montana.csci.csci468.misc;
2
3 import edu.montana.csci.csci468.CatscriptTestBase;
4 import org.junit.jupiter.api.Test;
5 import static org.junit.jupiter.api.Assertions.assertEquals;
6
7 public class CatscriptPartnerTests extends CatscriptTestBase {
8     @Test
9     void ifInAFor() {
10         assertEquals( expected: "1\ny equals 2\n3\n", executeProgram( @C "var x = [1,2,3]\n" +
11             "for(y in x){if(y == 2){print(\"y equals 2\")}else{print(y)}}");
12     }
13
14     @Test
15     void ifElseIf() {
16         assertEquals( expected: "x equals 1\n", executeProgram( @C "var x = 1\n" +
17             "if(x > 1){\" +
18             "print(\"x is greater than 1\")}\" +
19             "else if(x == 1){\" +
20             "print(\"x equals 1\")\""
21         ));
22     }
23
24     @Test
25     void quotes() {
26         assertEquals( expected: "this is a \"quoted\" passage\n", executeProgram( @C "print(\"this is a \\\"quoted\\\" passage\")");
27     }
28 }
```

The test suite provided by the other team member contains three tests; `ifInAFor()`, `ifElseIf()`, and `quotes()`. `ifInAFor()` tests an if statement inside of a for loop, `ifInAFor()` tests an if and else if statement that prints “x equals 1” if x is equal to 1. Last but not least, `quotes()` checks if a given string value has quotation marks when it's run by the compiler.

```
✓ Tests passed: 3 of 3 tests – 34 ms  
C:\Users\space\.jdk\openjdk-19.0.2\bin\java.exe ...  
  
Process finished with exit code 0
```

```
✓ CatscriptPartnerTests (edu.montana.csci.csci468, 34 ms)  
  ✓ quotes() 29 ms  
  ✓ ifElseIf() 2 ms  
  ✓ ifInAFor() 3 ms
```

Apart from the test suites and the respective Section 4s of team member 1 and team member 2's documents, the Catscript compiler program and the accompanying capstone portfolio and documentation were written independently.

Section 3: Design Pattern

For this project, our team utilized the memoization design pattern. Memoization is a very useful optimization technique that involves storing values in a cache, which can be accessed by the program when needed. This can be accomplished through the use of data structures such as symbol tables and hash maps, which store pairs of values and keys. This helps decrease the running time of a program; when memoization is implemented, the computer won't have to make the same calculations repeatedly and can just call the stored value from the cache when it is needed for another calculation. This is essential if a program has enough complex calculations to significantly hinder its performance if those calculations are unoptimized. In particular, it helps improve the runtime of recursive functions, which are designed to run themselves repeatedly until the specified conditions in the program are met.

Symbol tables are very prominent in our compiler. In the `validate()` function of our `IfStatement` class, symbol tables are used to store pieces of an if statement and check if they constitute a valid if statement or not. If not, an `INCOMPATIBLE_TYPES` error is thrown.

```
@Override
public void validate(SymbolTable symbolTable) {
    expression.validate(symbolTable);
    if (!expression.getType().equals(CatscriptType.BOOLEAN)) {
        expression.addError(ErrorType.INCOMPATIBLE_TYPES);
    }
    symbolTable.pushScope();
    for (Statement trueStatement : trueStatements) {
        trueStatement.validate(symbolTable);
    }
    symbolTable.popScope();
    symbolTable.pushScope();
    for (Statement elseStatement : elseStatements) {
        elseStatement.validate(symbolTable);
    }
    symbolTable.popScope();
}
```

Section 4: Technical Writing

Introduction

Catscript is a simple scripting language. It is a statically-typed functional programming language. This document will cover the type system and various features included in the Catscript programming language.

Types

`int` - a 32-bit integer

`string` - a java-style string

`bool` - a boolean value

`list<x>` - a list of values with type `x`

`null` - the null type

`value` - any type of value

Reserved words

The reserved words in Catscript are as follows:

`else`

`false`

`function`

`for`

`in`

`if`

`not`

`null`

`print`

`return`

`true`

`var`

These words are not allowed to be used as variable or function names. They have specific uses in the programming language itself and trying to use them in other ways will result in errors.

Comments

In the same fashion as java, you can create a single-line comment using a `//`.

Declaring variables

To declare a variable, simply write the word `var` followed by your variables name, and set it equal to the desired value. Because Catscript is statically typed, a value must be declared with the variable upon creation. You can also declare your variables type upon creation of your variable. Please note that each variable/function name can only exist once in your program. For example, you may not name two variables `x`, nor can you name a variable

`x` and also have a function named `x`.

```
1 //declaring a variable without a type
2 var x = 1
3
4 //declaring a variable with a type
5 var y : string = "hello"
```

if-statements

The Catscript if-statement works much like if-statements in other languages:

```
1 var x = 0
2
3 if(x < 1){
4     print("x is less than 1")
5 }
```

The keyword `if` indicates the start of an if-statement. In parentheses, an expression is evaluated. If that expression evaluates to true, then the code in the brackets following will be executed. Otherwise, the code in the brackets will not be executed.

In addition to the `if`, users can make use of `else`:

```
1 var x = 2
2
3 if(x < 1){
4     print("x is less than 1")
5 }else{
6     print("x is not less than 1")
7 }
```

If the user so desires, they can also compound `else` with `if` for an `else if` statement:

```
1 var x = 2
2
3 if(x < 1){
4     print("x is less than 1")
5 }else if(x == 2){
6     print("x equals 2")
7 }else{
8     print("something went wrong")
9 }
```

for-statements

The for-statement in Catscript requires a list to iterate through:

```
1 var x = [1,2,3]
2 for(y in x){
3     print(y)
4 }
```

The list can be stated in the for-statement (as opposed to the way demonstrated above, where a separate variable was used to reference the list).

print statement

The print statement works as follows:

```
1 print("this is a print statement")
```

Please note, it is required that you use double-quotes around strings. Using single quotes will result in an error. You can only assign values of the same type as the variable to the variable.

Declaring functions

Functions are declared with the `function` keyword, followed by function name, a list of parameters, an optional return type, and the function body:

```
1 function myFunction(x){  
2     print(x)  
3 }  
4  
5 myFunction(1)
```

Calling functions

You can call a function by simply typing out the functions identifier and putting the appropriate arguments in place for the parameters, as demonstrated above.

Parameters

When listing parameters, you have the option of just listing variable names, or listing variable names with their associated types:

```
function myFunc(a, b, c) {}  
//or  
function newFunc(a: int, b: string, c:bool){}
```


Grammar

This is the Catscript 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 , ')'

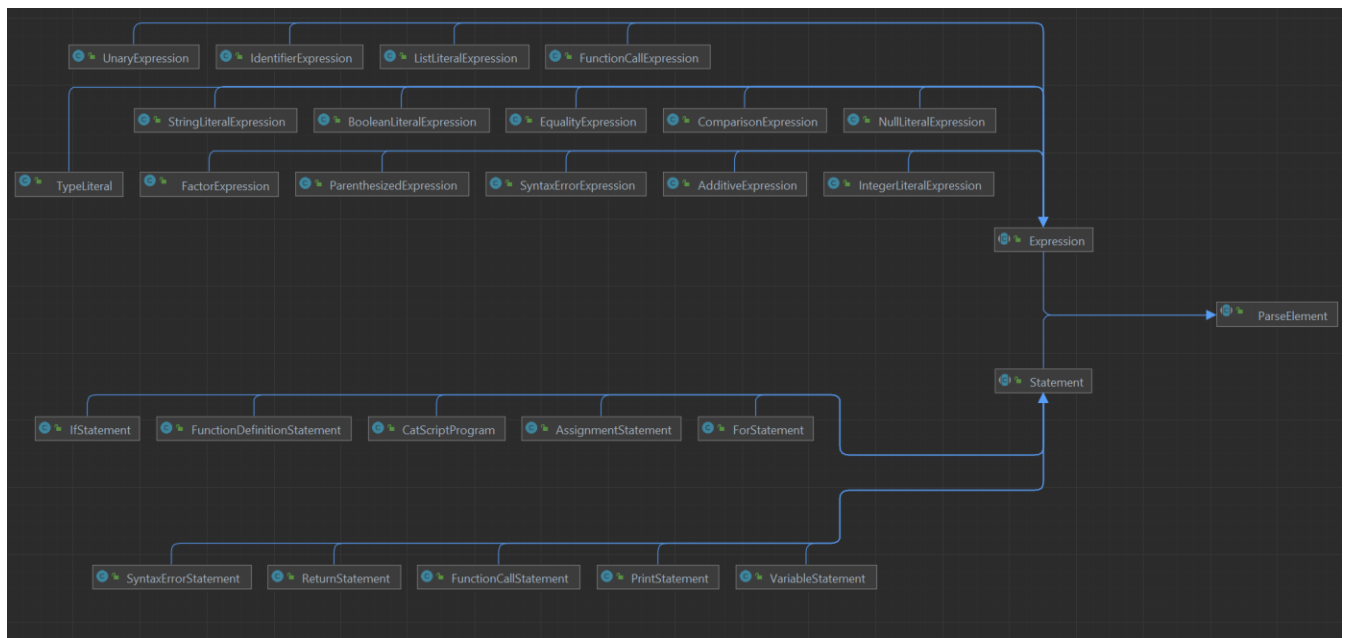
argument_list = [ expression , { ',', expression } ]

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

```

Section 5: UML Diagram

Attached below is an UML diagram generated in IntelliJ that provides a visual of how the Catscript compiler is put together, and of how all the classes used in the program are related to each other. All the expression subclasses in the program are inherited by the `Expression` superclass, and all the statement subclasses are used by the `Statement` superclass. Both `Expression` and `Statement` are subclasses of the `ParseElement` class, which helps handle much of the parsing process in the compiler. All of these classes are like gears in a machine; if even one of them is missing, the compiler itself won't be able to function and errors will be thrown.



Section 6: Design Trade-offs

The main design trade-off for the Catscript compiler was the decision to write a parser using the recursive descent system, as opposed to a parser generator. A parser generator is a tool that takes in a grammar for a programming language and uses it to generate a parser for a compiler, which reduces the amount of code that needs to be written entirely by hand. There are also fewer systems to break during the programming process, which is a clear upside. However, writing a parser generator to do some of the work for you as opposed to writing a compiler from the ground up doesn't do the programmer any favors when it comes to learning about why recursion is so important in programming language grammars. The crux of the recursive descent system is using recursive calls throughout the parser to parse the input value until the program has run through the parse tree, all without any backtracking.

Additionally, parser generators are very rare in the industry compared to recursive descent parsers, which means that for students looking to graduate and look for a job in the CS field, writing a parser generator as opposed to a recursive descent parser wouldn't give them as much experience that would be relevant in the job market. Since recursive descent parsers are far clearer about how language grammars work, our team gained a better understanding of how the compilation process works, and in turn, a better understanding of programming as a whole.

Section 7: Software Development Life Cycle Model

For the purposes of this project, we used Test Driven Development (TDD) as our software development model. Test Driven Development is a model where a project is developed around one or more test suites that test for specific outputs to determine whether the program is working as intended.

Test Driven Development was an incredibly useful development model for our team. It made writing the Catscript compiler far less of a nightmare than it otherwise would have been. For this project, we were provided with multiple test suites for each assignment/checkpoint of the project, and the main objective of each checkpoint was to get all the test suites for a specific implementation of the compiler passing. In the provided source.zip folder, there are test suites for tokenizing, statement and expression parsing, and bytecode. Transpiling test suites are also present, but for the purposes of this project, we were not required to get those working. During development, the test suites were instrumental for helping us keep track of our progress with writing the inner workings of the compiler. The test suites also helped us debug issues with our code that we encountered during the development process, and provided our team a starting point whenever we were trying to figure out the origin of a failing test in the code. Throughout the four checkpoints we were assigned throughout the semester, we had to get a total of 148 tests passing. 29 additional tests across the transpiling test suites exist that bring the total up to 177. However, since transpiling involves taking code written in one programming language and translating it into another language, it did not need to be implemented for the purposes of this project.