

A dark blue vertical bar on the left side of the page. A blue arrow points to the right from the bar, containing the text 'CSCI 468'.

CSCI 468

Senior Compiler Portfolio

Kruize Christensen &
Zach Snyder

Several thin, curved lines in dark blue and light gray originate from the bottom left and curve upwards and to the right.

Christensen, Kruize
MONTANA STATE UNIVERSITY

Catscript Documentation

Section 1: Program

This is located in the directory `csci-468-spring2023/capstone/portfolio/source.zip`

Section 2: Teamwork

With this compilers project, we were told to choose a partner toward the beginning of the semester. The partner I chose was Zach Snyder who I have personally worked with in other classes and projects. Also his reputation of being a hard worker from other students at Montana State University. Like every other class, we have set up a repository on GitHub we each had our separate code to work on. So it made working on this project a little interesting. What I mean by that is we basically worked separately until one of us came to an impasse on one of the tests. Most of the time, working separately caused us to run into different issues, where the others could walk through how they got their test to pass.

Member 1

Contributions:

- Main Coder
- Problem Solver

Percentage Worked: 60% contribution
of time

Member 2

Contributions:

- Tester
- Ideas/methods

Percentage Worked: 40% contribution
of time

Member 2 Tests

```
public class ZachTest extends CatscriptTestBase {  
  
    no usages  🧑 Kruize Derek Christensen  
  
    @Test  
    void comparisonExpressionsReturnCorrect() {  
        assertEquals( expected: true, evaluateExpression( src: "8 + 10 + 34 > 100 - 66 - 24"));  
        assertEquals( expected: false, evaluateExpression( src: "4398 / 4390 > 23 * 2498"));  
    }  
  
    no usages  🧑 Kruize Derek Christensen  
  
    @Test  
    void additiveExpressionsReturnCorrect() {  
        assertEquals( expected: 55, evaluateExpression( src: "1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10"));  
        assertEquals( expected: -53, evaluateExpression( src: "1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10"));  
        assertEquals( expected: 37, evaluateExpression( src: "(7 + 10) - (10 - 11) + (11 + 8)"));  
    }  
  
    no usages  🧑 Kruize Derek Christensen  
  
    @Test  
    void printStatementsReturnCorrect(){  
        PrintStatement printExpression = parseStatement( source: "print(\"Hello World!\")");  
        assertNotNull(printExpression);  
        assertFalse(printExpression.hasErrors());  
        PrintStatement printExpressionTwo = parseStatement( source: "print(\"Hello World!\", verify: false);");  
        assertNotNull(printExpressionTwo);  
        assertTrue(printExpressionTwo.hasErrors());  
    }  
}
```

✓ Tests passed: 3 of 3 tests – 145 ms

Member 1 Tests

```

public class KruizeTest extends CatscriptTestBase {
    no usages  Kruize Derek Christensen
    @Test
    void literalExpressionsCompilesProperly() {
        assertEquals( expected: -11, evaluateExpression( src: "8 + 6 + (7-5) - 3 * (0 + 9)"));
        UnaryExpression expr = parseExpression( source: "not not true");
        assertEquals( expected: true, expr.isNot());
    }
    no usages  Kruize Derek Christensen
    @Test
    void RecursionTest() {
        assertEquals( expected: "9\n7\n5\n3\n1\n1\n1\n", executeProgram(
            src: "function recur(x : int){\n" +
                "    print(x)\n" +
                "    if(x > 0){\n" +
                "        recur((x + 1)-3)\n" +
                "    }\n" +
                "    recur(9)\n"
        ));
    }
    no usages  Kruize Derek Christensen
    @Test
    void match(){
        assertEquals( expected: 2, evaluateExpression( src: "(10 - -2)/6 "));
        assertEquals( expected: 4, evaluateExpression( src: "-1 - -5"));
        assertEquals( expected: -8, evaluateExpression( src: "100*2/25*-1"));
    }
}

```

✓ Tests passed: 3 of 3 tests – 153 ms

Section 3: Design pattern

The memorization pattern, also known as caching, is a design pattern used in computer programming to optimize the performance of an algorithm by storing the results of time-consuming calculations and returning them when the same calculation is required again.

This design pattern is particularly useful when dealing with repetitive computations, as it can significantly reduce the time needed to execute them. By saving the results of a calculation,

the algorithm can avoid repeating the same calculations multiple times, thereby improving the overall efficiency of the code.

In the case of the project, the class decided to implement the memorization pattern by using a HashMap to store the Catscript Types that were determined in the code. This allowed the program to save time-consuming calculations and retrieve them later if the same input was provided again. The HashMap provides a fast and efficient way to store and retrieve data, making it a suitable choice for this purpose.

Overall, implementing the memorization pattern can help improve the efficiency and speed of the code by reducing the number of calculations required. It is a useful design pattern to consider when dealing with algorithms that involve repetitive computations or when the speed of the code is a crucial factor.

```
// TODO memoize this call
3 usages
static final HashMap<CatscriptType, ListType> CACHE = new HashMap<>();
5 usages  Kruize Derek Christensen
public static CatscriptType getListType(CatscriptType type) {
    if (!CACHE.containsKey(type)) {
        CACHE.put(type, new ListType(type));
    }

    return CACHE.get(type);
}
```

Section 4: Technical writing

1 Introduction

This program is a compiler for Catscript. Catscript is a statically typed language that can compile to JVM Bytecode. It uses a recursive descent parser and is able to work with objects, type inference, and primitives. It can also work with other features that will be shown later in the documentation

2 Background

A compiler takes code written in one language and translates it into another language. Mostly, it will take code in text form and export it into binary form. The way our compiler works is through its five main components: the tokenizer, the parser, the JavaScript transpiler, evaluator, and bytecode. We developed it in the JetBrains software, IntelliJ, and used GitHub as our repository. Carson Gross designed the project and gave us everything we needed to finish it.

3 Expressions

An expression is a bit of code that will evaluate to some integer, string, or boolean at some point. All expressions in Catscript inherit an abstract Expression class. The abstract expression class uses a method called `getType()` to return what type that a given expression will evaluate to.

3.1 Primary Expression

Primary expression may be any of the following:

- Identifier
- String literal
- Integer literal
- Boolean literal
- Null literal
- List literal
- Parenthesized expression
- Function call

Any of the expression/statement that has another expression will at some point become a form of a primary expression.

3.2 Factor Expression

Multiplication and division are implemented with the factor expression. The operations are * and /. The following are valid factor expressions in

Catscript:

foo * -2

4 / -2

(x - y) * 5

3.3 Additive Expression

Addition and subtraction is done with the additive expression. The operations are + and -. The + operator can also be used for string concatenation. The following are valid additive expressions in Catscript:

foo + 6

10 - 6

bar + "foo"

3.4 Unary Expression

Not and negative are done with the unary expression. The operators for this are 'not' and -. The following are valid unary operators in Catscript:

-6

not true

3.5 Identifier Expression

The variables are implemented with the identifier expression. This expression holds a string as its variable name. It also contains the variable's type since Catscript is a statically typed language. The name of the variable is then used to look for its value in the symbol table.

3.6 Equality Expression

Equals is implemented with the equality expression. The operators for this is == for equals and != for not equals. The following are valid equality expressions in Catscript:

```
x == y
```

```
x != 10
```

```
foo == bar - y
```

3.7 Comparison Expression

Less than, greater than, less than or equal to, and greater than or equal to are all implemented with the comparison expression. The operators in order are $<$, $>$, $<=$, and $=>$. The following are valid comparison expressions in Catscript.

$x > y$

$x < 3 + 2$

$foo * 3 => bar * y$

3.8 Conclusion of Expressions

This wraps up how the expressions are implemented in Catscript. Next will be the documentation on the implementation of statements.

4 Statements

Just like how the expressions are all inherited from the Expression class, all statements in Catscript inherit from the Statement class. Statements differ from expressions by changing the program's state as opposed to how expressions evaluate to a value.

4.1 Assignment Statement

Variables are modified with the assignment statement. The following includes the syntax for the Catscript assignment statement:

- Identifier
- = symbol
- Expression

The following has valid assignment statements in Catscripts:

```
x = 5
```

```
foo = "Hello world!"
```

```
x = 1 + 1
```

4.2 For Statement

Syntax for For Statement includes the “in” keyword. The for statement is only used for iterating through a list and can't count to a certain value.

Catscript for statements require the following:

- ‘For’ keyword
- ‘(‘ symbol
- A variable name
- ‘In’ keyword

- An expression
- ‘)’ symbol
- ‘{‘ symbol
- A series of lines to evaluate
- ‘}’ symbol

The following is a valid for statement in Catscript:

```
for (x in [1, 2, 3, 4, 5]){  
    print(x)  
}
```

4.3 Function Call Statement

In order to call a statement, you need to use a function call statement. The program makes sure that the number and type of the arguments match the definition of the function. As well, the program checks for a function that exists with the same name and is in the symbol table. Function calls are made with the following:

- Function name
- ‘(‘ symbol

- Function arguments
- ‘)’ symbol

The following is an example of a Catscript function call:

```
foo(x, y)
```

```
aFunction(foo, bar)
```

4.4 Function Definition Statement

Functions that can be used elsewhere are defined by the Function Definition Statement. All functions can have a return type. But they can also return void if there is no given void type. Functions in Catscript are created as the following:

- ‘Function’ keyword
- Function name
- ‘(‘ symbol
- Parameter list
- ‘)’ symbol
- ‘:’ symbol as well as the functions return type (optional)

- ‘{’ symbol
- Lines of statements for the function to evaluate
- ‘}’ symbol

The following is how Catscript functions are defined:

```
function foo (x : string, y : string){  
    return x + y  
}
```

4.5 If Statement

Syntax for if statements require the ‘if’ keyword as well as ‘else if’ and ‘else’ optionally. It is used to determine if the block of code underneath it can run or not. The if statement in Catscript is created as the following:

- ‘If’ keyword
- ‘(’ symbol
- Expression

- ‘)’ symbol
- ‘{‘ symbol
- A series of lines to execute
- ‘}’ symbol
- Optional ‘else’ keyword
 - Optional ‘if’ keyword and statement
 - ‘{‘ symbol
 - Series of lines to execute
 - ‘}’ symbol

The following is a valid if statement in Catscript:

```
if (x => 5){  
    print (x)  
}  
else if (x < 0){  
    print y  
}
```

4.6 Print Statement

The Print Statement is used to print lines. The keyword ‘print’ is required for this. The print statement can only have expressions passed into it. The following is the syntax for print statements:

- ‘print’ keyword
- ‘(‘ symbol
- Argument or string
- ‘)’ symbol

The following is an example of a Catscript print statement.

```
print (“Hello World!”)
```

```
print (x)
```

4.7 Return Statement

In order to exit a function, the return statement is used. This would possibly return any value. This statement can only be used inside of a function.

Otherwise, it throws a syntax error. The return statement in catscript has the following syntax:

- ‘return’ keyword
- Expression that is being returned

The following is an example of a valid return statement in Catscript:

```
function foo (x : int, y : int){  
    return x + y  
}
```

4.8 Variable Statement

New variables are declared and assigned with the variable statement.

Catscript can not declare a new variable without giving an expression that is to be assigned as a value. If a type is not specified, the variable type is guessed from the `getType()` method. Otherwise, Catscript verifies if the type is assignable from a type that is meant to be returned from the expressions `getType()`. If it doesn't, then the program will throw an `Incompatible Types` error. The following is the syntax for Catscript variable statements:

- 'var' keyword
- Variable name
- '=' symbol
- Expression

The following are valid variable statements:

```
var x = 5
```

```
var y : int = 5
```

5 Type System

A very simple type system is used in Catscript. The following include the proper types:

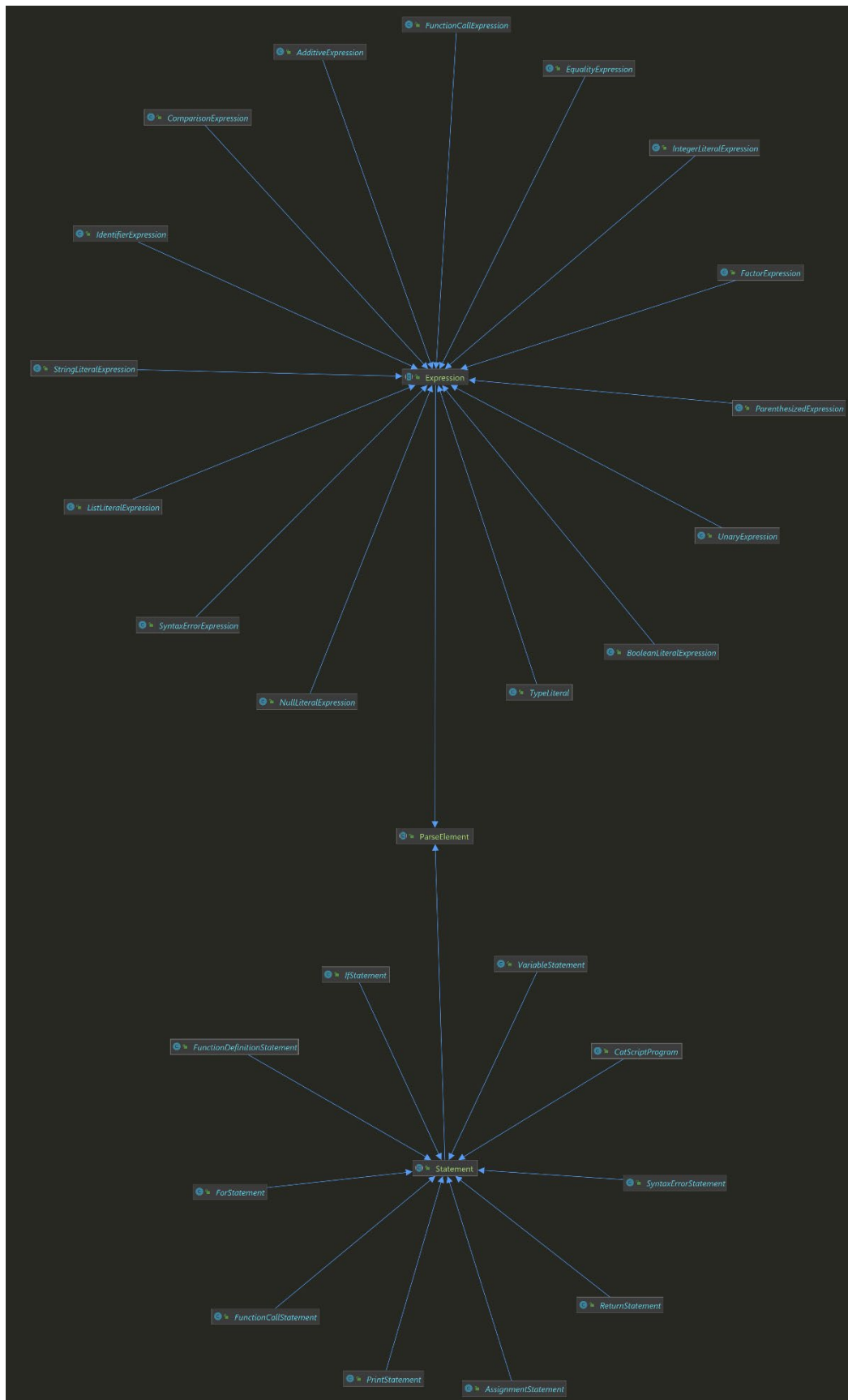
- String - A string similar to how Java operates
- Int - A 32 bit integer
- Bool - a boolean value that can either be true or false
- List<x> - a list of values of a given type (x)
- Object - type of value
- Null - null type

6 Conclusion

Through this project, we learned a lot about how coding languages are made and how we'd go about making our own if needed in the future. We got to put the

pieces together with each stage to learn how a compiler works to translate languages. This is a project that has inspired me to make my own language in the future.

Section 5: UML



Section 6: Design trade-offs

When learning about and using the recursive descent algorithm, which is a way to create parsers. The way this works is that we create a function for each production in the grammar of the language we're working with. These functions are then called recursively throughout the program as needed to parse the input we're given.

For example, imagine we want to create a parenthesized expression like $(6+3)$. We would start by calling the "expression" function and then work our way down to the "primary_expression" function to get the necessary parenthesis. After that, we'd call the "expression" function again. This shows how the algorithm works recursively to parse input like $((((1+5)-2)+7))$.

By using the recursive descent algorithm, we're able to break down complex expressions and parse them efficiently. It's a powerful technique that's widely used in programming, particularly in parsing languages. I also really enjoyed using the diagram below as a reference for the majority of the coding.

```

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, '>' ]

```

The recursive descent algorithm offers several advantages for creating parsers. For one, it clearly demonstrates the recursive nature of parsers, making it easier to understand how they work. Also, it's a commonly used approach that's widely used in the industry.

However, there are some trade-offs to consider when choosing this design approach over using a parser generator. For instance, you may need to create more infrastructure for the program, and you may end up having to write more code to achieve the same tasks. Despite these drawbacks, the recursive descent algorithm remains a popular and effective technique for creating parsers.

Section 7: Software development life cycle model

To guide the code's development in the capstone project, we used the Test Driven Development (TDD) model. Carson Gross set up the code's base model, including all necessary classes and framework, and created a test section in our repository with various "assertEquals()" tests for our expressions, statements, and other functionality. These tests helped us identify which methods needed work and which code we had to create ourselves to pass the checkpoints and ensure that our program was running correctly and smoothly.

I had prior experience with TDD in my earlier classes such as Computer Systems and Data Mining, where we were required to complete tests for our final grades. I found TDD to be an excellent model to work with because it provided a clear indication of what was necessary to achieve a passing grade and a functional code base. However, some tests were not useful in

indicating what code was missing or if something was not functioning as intended. To address this, we used the debugging method in IntelliJ, stepping through the tests to pinpoint problems. One of the issues that arose was an infinite loop that took a lot of stepping through or to find. It was disheartening when this happened because when one thing broke a lot of others broke as well.

Working with a large codebase such as this helped us gain experience in working with legacy code and solving problems with our written code. This experience was valuable for building teamwork skills and closely resembles how projects are executed in the corporate world. Sometimes it was based on collaboration and other times it involved a lot of searching through the code itself.