Montana State University
Gianforte School of Computing

CSCI 468 Compilers

Jada Bryant
Walker Ward
May 2021

Table of Contents

Section 1: Program

The source code can be found at this location: csci-468-spring2021-private/capstone/src.zip

Section 2: Teamwork

This project was divided such that each team member would have dedicated tasks and sections of the project to focus on. Team Member 1 was responsible for management of the GitHub repository and was the primary contributor to the code base. Their programming focused heavily on completing the tokenizer, parser, evaluation, and bytecode checkpoints to the GitHub repository prior to each one's deadline. This team member also resolved any merge conflicts when they arose, and for ensuring that the program passed all tests. Team Member 2 was responsible for writing the technical documentation, such as this report, and for writing program tests to augment our test driven development paradigm. All communication between team Member 1 and team Member 2 was facilitated via discord and email. Due to the coronavirus pandemic in-person contact and meetings were deemed an unnecessary safety hazard.

Estimated Contributions

Total estimated hours: 120

Team Member 1:

Contributions: Code management and primary programmer.
Estimated Work Hours: 100 hours, ~83% contribution in time.

Team Member 2:

Contributions: Documentation and tests.
Estimated Work Hours: 20 hours, ~17% contribution in time.

Section 3: Design Pattern

Memoization

In programming memoization is a technique applied to expensive recurring function calls in order to optimize it. At a high level, this approach stores the result of this function call so, that the next time this function is called, it will not have to repeat the same computation. It is essentially caching a set of inputs and outputs.

To accomplish this functionality the function must store its parameter values and return value in a cache any time it is called with new arguments. The cache is typically implemented as a HashMap or another type of data structure. Anytime this function is called it compares its parameters to those in the cache and, if they are logically equivalent the function returns the originally computed value from the cache that is associated with those arguments.

The benefits of this technique are that for extremely time-consuming computations it saves time that would otherwise be spent on the computation. It could also be argued that this approach could save system resources such as ram and CPU time. However, this system is not without drawbacks. For one, this technique must take time to check whether this call has already been stored in the cache or not. Secondly, this pattern is generally only viable for functions that are called often, can only take a narrow range of input and do not rely on random numbers. Furthermore, this approach is not thread safe. Finally, this technique would not be viable in a real production system, but it is sufficient for this project.

Implementation: src.zip\...\parser\CatscriptType.java

Section 4: Technical Documentation

Introduction to Catscript

This documentation will include a brief overview of the programming language known as Catscript as well as some detailed information about using Catscript, its features and the type system and is not meant to be utilized as a language reference. Catscript is a very small language that is considered to be 'toy-like' because of its limited features and functionality that is broadly based off of Java.

Catscript programs consist of a series of statements they will be executed sequentially. The typical first program in any language is the "Hello World" program (see below) using a variable. Within this program you can see the variable statement as well as the print statement.

```
1 var x = "hello world!"
2 print(x)
```

Catscript Web App

Catscript uses a small web app to run. It includes five features to display different functionality of the compiler. To access the web app run the file 'CatscriptServer.java' located at src.zip\...\CatScriptServer.java in your IDE of choice. Once that is running open a browser and connect to localhost:6789 or 127.0.0.1:6789 via the address bar. The web app's features are described below:

- Tokenize: Detects the type and location of each token in the program and outputs that information.
- Parse: Identifies each statement and expression from the identified tokens in the program and outputs the type of statements.
- Evaluate: Runs the program and outputs the result as would normally be done in an IDE.
- Transpile: Not functional
- Compile: Generates the bytecode for the program and outputs it. Unfortunately, this command works on the web app but currently displays the bytecode in the terminal of the IDE used to run CatscriptServer.java.
- Load: Also 'Load A File'. Loads a Catscript program from a file with the extension '.cat'. Included with the web app is demo.cat that demonstrates most of the functionality and includes the syntax of Catscript for first time users.

Type System

Catscript is statically typed and supports the following types:

- int – a 32-bit integer
- string – a java-style string
- bool – a Boolean value
- list<x> – a list of value with the type ‘x’
- object – any type of value
- null – the null type

Features

Catscript is a statically typed language with support for list literals, local variable inference, basic control flow, function definitions, recursion, some common expressions, autoboxing, and some common types.

Expressions

An expression is a segment of code that evaluates to a value. A single expression can be a combination of many expressions. Catscript supports some of the most common expression types for mathematical, logical, and relational operations such as the Unary and Comparison expressions. The parser will segment the tokenized code into a series of expressions and statements. Once parsed Catscript knows the type of each expression and can use that to preform operations such as addition or equality.

Literal Expressions

Literal expressions are a sub-type of expression. These expressions evaluate to the literal value of a type.

Boolean Literal

A Boolean literal is a literal that evaluates to a Boolean value. In Catscript a Boolean can be either true or false.

Integer Literal

An integer literal is a literal that evaluates to some int value. In Catscript a integer can be any whole number within 32-bits.

List Literal

A list literal is a literal that contains a list of values. In Catscript a list can contain any other type. Lists in Catscript are read-only. They values inside cannot be modified.

Null Literal

A null literal is null. That’s it, it’s just null. It also evaluates to null.

String Literal

A string literal is a literal that evaluates to some String value. In Catscript a string is a java-style string can contain any characters that a java-style string can.

Type Literal

A type literal is a literal that evaluates to a type. In Catscript this means that a type literal can be any one of the six types listed under the Type System.

Additive Expression

An additive expression is an expression that evaluates to the sum of its left-hand and right-hand sides. The additive expression supports addition denoted with the '+' operator and subtraction denoted with the '-' operator. Below is an example of a simple and a complex additive expression.

```
1 + 2

1 + (2 + (19 - 231 + (1209))) - 94
```

Comparison Expression

A comparison expression is an expression that evaluates to a Boolean value by comparing the values of its left-hand and right-hand sides based on the a given operator. The comparison expression supports greater than denoted with the '<' operator, less than denoted with the '>' operator, greater than or equal to denoted with the '<=' operator and less than or equal to denoted with the '>=' operator. Below is an example of a simple and a complex comparison expression.

```
1 < 2

1 + (2 + (19 - 231 + (1209))) - 94 <= 468
```

Equality Expression

An equality expression is an expression similar to a comparison expression as it evaluates to a Boolean value by comparing the values of its left-hand and right-hand sides based on the a given operator. The equality expression supports equals denoted by the '==' operator and not equals denoted with the '!=' operator. Below is an example of a simple and a complex equality expression.

```
true == true

1 + (2 + (19 - 231 + (1209))) - 94 == 468
```

Factor Expression

A factor expression is an expression that evaluates to the product or quotient of its left-hand and right-hand sides based on the provided operator. The equality expression supports both multiplication denoted with the '*' operator and division denoted with the '/' operator. Below is an example of a simple and a complex factor expression.

```
8 * 11

1 * (20000 * (19 * 231 / (1209))) / 94 == 468

```

Parenthesized Expression

A parenthesized expression is an expression that contains another expression. The parenthesized expression requires that it have a matching '(' and ')'. This expression is primarily used to separate other expression as a means of ensuring that one expression evaluates before another. Please note that Catscript does not support decimals and all mathematical operations are rounded accordingly. Below is an example of a simple and a complex parenthesized expression.

```
(-1)

(1 * (20000 * (19 * 231 / (1209))) / 94)

```

Unary Expression

A unary expression is an expression that contains an operator and is followed by either an integer literal, Boolean literal or an '=' operator. The unary expression requires that the unary operator either '-' for negative or '!' for logical not be the first token in a unary expression. This expression is primarily used to invert a Boolean value, to verify that two values are not equal or to make an integer negative. Below is an example of a simple Boolean negation, negative integer and a not equals comparison.

```
!true

-1

x != y

```

Statements

A statement is a segment of code that preforms and action, they execute. A single statement can contain many nested statements. Statements also have side effects and can interact with the environment.

Assignment Statement

An assignment statement is a statement that changes the value of a given variable within the current scope. The assignment statement requires a variable identifier followed by an '=' operator finally followed by an expression. See the grammar below.

```
assignment_statement = IDENTIFIER, '=', expression;
```

Var Statement

A var statement is a statement that declares a variable within the current scope with a given name. The var statement requires a var keyword a variable identifier and '=' operator and a expression. It should be noted that a type can be provided but is not required. See the grammar below.

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

Print Statement

A print statement is a statement that acts as an output and can display information to the user. The print statement requires the print keyword, a set of parentheses containing an expression. See the grammar below.

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

If Statement

An if statement is a control flow statement that acts as a method for Catscript to choose a set of statements to execute based on the result of one or more expressions. The if statement requires the if keyword and a set of parentheses containing an expression followed by a set of braces that contain a set of statements to be executed if and only if the before mentioned expression evaluates to true. The Catscript if statement supports else clauses, else if clauses and nested if statements. See the grammar below for details.

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

For Loop

A for loop or for statement is another method of control flow that essentially allows Catscript to iterate over a set of statements until an expression evaluates to true. The for statement requires the for keyword and a set of parentheses containing an identifier the in keyword and an expression followed by one or more statements contained within a set of braces. Catscript does not support nested for loops. See the grammar below for details.

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

Functions

Catscript supports function declarations and function call statements. Functions can be created to allow for multiple sequential code segments to be abstracted into a single keyword for later use. Function calls take a list of parameters that match the definition of a function. Functions may also return a type once they have executed the rest of their statements using a return statement. A return statement is a statement the sends a value to the program upon a function call after the rest of this function has been executed. See the following images for details.

```
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];
```

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

```
function foo(x : int, y : String) {
    print(x)
}

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

Tokenizing

Tokenizing is the process of taking in a program as one long string and segmenting it into small identifiable chunks know as tokens that can be understood by the parser. The tokenizer also includes details about each token such as its type and location within the line that it is on. Once successfully tokenized the token stream is sent to the parser.

Parsing

Parsing is the process of taking in a string or stream of tokens and converting them into something expressions and statements. This parser uses a parse tree to help determine the function of each token, whether it's supposed to be part of an expression or statement. Parsing also allows us to detect syntax errors. It should be noted that Catscript is left associative, meaning that the left most operator bind more tightly. Similarly, division bind more tightly than subtraction.

The Catscript Parser used a technique called recursive decent. This form of parsing appears to mimic the Catscript grammar. Essentially, when approached with a token stream the parser works its way down though the grammar until the current operator is matched. Upon matching it does the same for both the left-hand and right-hand sides of this line until both sides have been identified as a type of expression. Once this happens the parse returns up the grammar and moves on to restart the process for the next line until the entire program has been parsed. Note that statements were emitted from this explanation for simplicity.

Evaluating

Evaluation is the process of taking the parse tree, expressions and statements from the parser and evaluating them. This is the section that actually 'runs' the code. Preforming all of the logical and mathematical operations sequentially through the program.

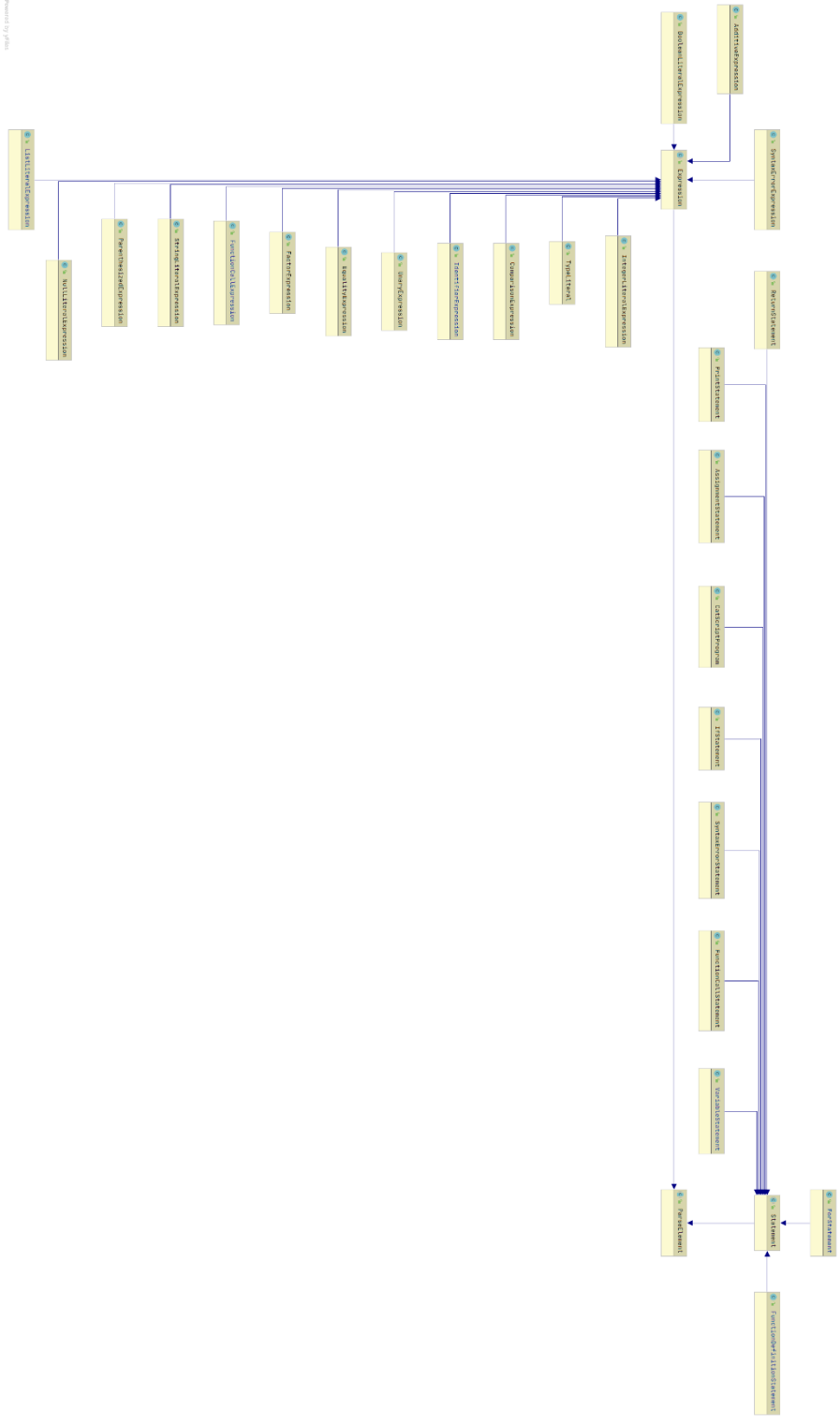
Compiling

Compiling or Compilation is the process of taking the evaluated Catscript code and converting it into bytecode. Bytecode is a form of program code that has been compiled from source code, in this case Catscript source code, to be interpreted by lower-level software.

References

Please note: Some of this documentation indirectly references notes, documents, course lectures and other materials provided by Instructor Carson Gross for CSCI 468: Compilers in the Spring 2021 semester.

Section 5: UML



UML Diagram of Parse Element Hierarchy

Section 6: Design Trade-Offs

Recursive Decent vs. Parser Generators

Recursive Decent

Recursive Decent is a top-down parsing algorithm, widely used in industry to parse tokenized source code. This method of parsing allows the programmer to develop a much better understanding and intuition about grammars and parsing because of its simplicity. The primary idea of this technique that for each production in the grammar there exists a method for it that is named after it. In that method, call the methods defined on the right-hand side of the production, matching as needed. This idea of providing every production its own method allows the recursive nature of parsers to be explained more clearly and understood better by those learning them for the first time.

Parser Generators

Parser Generators are made of two chunks: A lexical grammar as a REGEX (Regular Expression) and a long grammar as a EBNF (Extended Braccus Naur Form). The lexer is generated by mixing code generation with the actual grammar. The parser looks very similar to the lexer and produces an AST (Abstract Syntax Tree). The parser actually generates a recursive decent parser it's just much more abstract and harder to read. Theoretically, Parser Generators require less code and infrastructure to get going. However, more time is typically spent learning about their idiosyncrasies that the compiler and parser itself. Furthermore, Parser Generators are not used widely in industry, are much more complicated for our purposes and much harder to read than a Recursive Decent parser.

So, after careful consideration and paying close attention to out complexity budget and schedule it makes much more sense to implement a recursive decent parser. To reiterate, they are simpler, easier to understand, more widely used and gives us a better understanding of parsers and compilers.

Section 7: Software Development Life Cycle Model

Test-Driven Development (TDD)

Test-Driven Development is a style of programming that focusses on coding, testing and design at the same time. With this in mind, developers design the bare minimum to make tests pass, no more, no less. This has been attributed with much cleaner, clearer and simpler code that is often less prone to bugs. This allows the programmers to focus on only what is important, ignoring, for the time being, the more advanced, complicated, or unrelated features.

Personally, TDD has been an eye-opener. Being able to run tests against code you're writing is an extremely helpful feature. Before being introduced to this method of programming I would write my own makeshift tests and run tests by hand, over and over and over again to test features, work out bugs and verify outputs. Now, all I must do is write a single test, or twenty, without even having to even start the code. Then, I can work on each test at a time, slowly but surely, getting each one to pass until the entire project is complete. Not only did this style of programming help me ensure my code was functioning properly but it also gave me a sort of roadmap, like chapters in a book. Each of them allowing me to get closer and closer to then end while leading me in the right direction.