Emma Marie Veldhuis

Montana State University- May 2024

Gianforte School of Computing

Senior Capstone CSCI 468 Compilers

Partner: Jordy Hexom

Section 1: Program

The entire project source code can be found on GitHub: https://github.com/emmav316/csci-468-spring2024-private/tree/main/src

Section 2: Teamwork

On our team, we each wrote documentation for one partner's project and also created 3 new test cases for their code. This allowed our group to further test and identify any areas for improvement enhancing the functionality of our codebase. I wrote the parser for this project (95%) and my team members provided documentation and tests (5%). Since our group consisted of three members, we each provided the documentation and test cases for one member. Partner 1 provided for Partner 2 who provided for Partner 3 who provided for Partner 1.

Section 3: Design pattern

One of the design patterns that was used in this project was memorization. Memoization is the technique of saving the results of a function call so that they can be accessed by identical calls. By saving the results of such function calls, the program does not need to spend time and computation power to compute these results again, instead, it can pull these results from storage. This means that the pattern eliminates redundant operations that would cause a worse time complexity. This storage is often set up in a HashMap due to their get and put methods. HashMaps also allow parameter values which is perfect for saving arguments to a function call in addition to the return value. When a memoized function receives a call, it checks to see if that specific call exists in the map. If it does, it will return that value, and if not, it will complete the calculations and save the result in the HashMap. In this project, memoization was implemented in the getListType function shown below.

```
1 usage
static final ConcurrentHashMap<CatscriptType, ListType> LIST_TYPE_CACHE = new ConcurrentHashMap<>();

10 usages  Carson Gross *
public static CatscriptType getListType(CatscriptType type) {
    return LIST_TYPE_CACHE.computeIfAbsent(type, ListType::new);
}
```

Section 4: Technical Documentation

Introduction

CatScript is a simple scripting language that supports a variety of features including variables, loops, conditional statements, functions, and data types. This document aims to provide a comprehensive guide to CatScript, analyzing its grammar and functionalities.

Example:

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

Language Features

For Loops

For loops in Catscript require the reserved word ‘for’ followed by an identifier variable, the reserved word ‘in’, and the expression to be iterated over, all enclosed in parentheses. For loops, iterate over a sequence, such as a list, and execute a statement for each element in the iteration.

Example:

```
var lst = [1, 2, 3]
for (i in lst) {
    print(i)
}
```

Print Statements

Print statements in Catscript are implemented using the reserved word 'print' followed by the content to be printed enclosed in parentheses. This content can be a string literal, variable value, or even the result of an expression.

Example:

```
var x  = "hello catscript!"
```

```
print(x)
```

If Statements

If statements in Catscript start with the reserved word ‘if’ followed by the condition to be evaluated, enclosed in parentheses. If the condition is true, the statement within the curly braces { } following the if statement is executed. Optionally, the reserved word ‘else’ can be added to execute an additional if statement or an ending statement when the condition is false.

Example:

```
var x = 10
var y = 5

if (x > y) {
    print(x)
} else {
    print(y)
}
```

Parenthesized Expressions

In the CatScript, parenthesis are utilized to specify the evaluation order of expressions, overriding default precedence rules. Parentheses explicitly indicate expression precedence, allowing control over expression evaluation. This feature is included in the primary expression grammar, as demonstrated in the language syntax example below.

Example:

```
var sum = (10 + 5) * 2
```

CatScript Grammar

The CatScript programming language utilizes a formal grammar to define its syntax rules. This document outlines the structure of CatScript programs through a context-free grammar representation. Each section describes different elements of the grammar, including program structure, statements, expressions, and declarations.

Program Structure

A CatScript program consists of one or more program statements enclosed within curly braces {}.

```
catscript_program = { program_statement };
```

Program Statements

Program statements in CatScript can be either regular statements or function declarations.

```
program_statement = statement | function_declaration;
```

Statements

CatScript supports various types of statements, including loops, conditionals, print statements, variable declarations, assignments, function calls, and more.

```
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;
```

Additive Expression

Additive expressions in CatScript are used to perform addition and subtraction operations on numeric values. These expressions are composed of factor expressions combined with addition or subtraction operators.

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

Comparison Expression

Comparison expressions in CatScript are used to compare values and determine relationships between them. These expressions evaluate to boolean (true or false) based on the result of the comparison. Catscript supports various comparison operators for comparing numeric and string values.

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

Equality Expression

Equality expressions in CatScript compare two values for equality or inequality (*e.g. numeric and string values*). These expressions evaluate to a boolean (true or false) based on whether the comparison holds true or not.

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

Factor Expression

Factor expressions in CatScript can include unary expressions combined with multiplication or division operations.

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

Unary Expression

Unary expressions in CatScript consist of unary operators applied to primary expressions. These operators include negation (-) and logical negation (not).

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

Primary Expression

Primary expressions in CatScript represent the most basic elements of expressions. They can include identifiers, string literals, numeric literals, boolean literals, null literals, list literals, function calls, and parenthesized expressions.

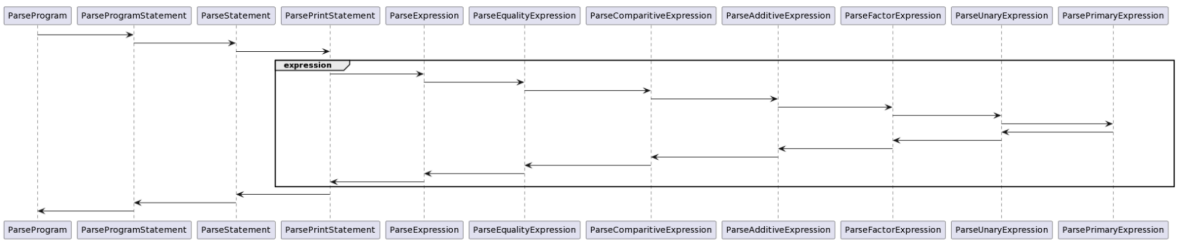
```
primary_expression = IDENTIFIER | STRING | INTEGER | "true" | "false" | "null"|
                     list_literal | function_call | "(", expression, ")"
```

Type Expression

Type expressions in CatScript specify the data type of a variable, parameter, or return value. They can include primitive types such as integers, strings, and booleans, as well as generic types such as lists.

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

Section 5: UML



This UML diagram above illustrates the overall architecture of the parser, which is designed using a recursive descent approach, where each class handles a specific grammar rule and calls other classes to parse sub-expressions or sub-statements. The diagram shows a series of classes, each responsible for parsing a specific part of the language's syntax. The classes are organized in a hierarchical structure, with the "ParseProgram" class at the top, followed by "ParseProgramStatement", "ParseStatement", and "ParsePrintStatement". The classes "ParseExpression", "ParseEqualityExpression", "ParseComparativeExpression", "ParseAdditiveExpression", "ParseFactorExpression", "ParseUnaryExpression", and "ParsePrimaryExpression" are responsible for parsing different types of expressions within the language. This UML diagram provides a visual representation of the parser's class structure, making it easier to understand the organization and relationships between different components of the parsing process.

Section 6: Design trade-offs

As a core component of this project, we opted to construct the parser by hand. This choice was driven by our desire to gain an understanding of the intricacies of the grammar through the recursive descent parsing approach. By using recursive descent, we deconstructed the grammar into smaller, more manageable units, facilitating a deeper comprehension of its intricate workings. This not only made the parsing process more accessible but also unveiled details that might have been missed had we relied solely on automated tools. The crafting of the

parser proved to be an exercise in problem-solving, as we navigated the challenges posed by the grammar's complexities. Each step contributed to the honing of our analytical and critical thinking abilities, fostering a growth mindset that will undoubtedly serve us well in future endeavors.

Beyond the benefits of understanding the grammar, our decision to construct the parser manually has provided an appreciation for the intricacies of language processing and translating abstract rules into executable code. This experience has not only deepened our comprehension of the subject matter but has also instilled a sense of accomplishment and confidence in our ability to tackle complex challenges through perseverance and a willingness to embrace unconventional approaches.

Section 7: Software development life cycle model

Throughout the course of this project, I have embraced the Test-Driven Development (TDD) model, a methodical approach that utilizes predefined tests to outline the desired code functionality and guide the development process. This strategy has proven invaluable, as it has enabled me to identify and address potential issues or bugs early on, mitigating the risk of compounding errors and ensuring a more robust and reliable codebase.

The TDD model has been instrumental in defining the explicit requirements necessary for the successful completion of the compiler project. By breaking down these requirements into testable units, we have been able to make incremental progress, systematically tackling each aspect of the project with a clear roadmap and measurable milestones. This iterative approach has not only fostered a sense of accomplishment with every successfully passed test but has also provided us with a structured framework for continuous improvement and refinement.

As an advocate of TDD, I have witnessed firsthand the profound impact it has had on our development workflow. The satisfaction derived from witnessing each test case transition from a state of failure to success is truly rewarding, instilling a sense of pride and motivation that propels us forward. Moreover, the comprehensive nature of the TDD process has afforded me a deep understanding of the software development lifecycle, from conceptualization to implementation and testing.