Capstone Portfolio

CSCI 468: Compilers

Montana State University

Spring 2024

Terris Dietz & **Charlie Weitzenberg**

Section 1: Program

All the catscript source code is provided in the folder labeled “source.zip”

Cascript 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 2: Teamwork

Charlie was a dedicated worker on this project and did 100% of the coding work for the project himself. Most of the work was finished early, and despite a slight hiccup on the third checkpoint, everything else was finished efficiently and on time. All of the documentation in this document was provided by Terris Deitz, along with the three tests shown at the bottom of the page. Likewise, Terris did 100% of the coding for his own project, and I wrote all of his documentation along with three tests I provided for him.

Section 3: Design pattern

One design pattern we used in our code was the memoization design pattern. The memoization design pattern is a technique where you store the results of expensive function calls and return the cached result when the same inputs occur again, instead of recalculating them. This can

greatly reduce the amount of memory used, as the program will not have to repeat memory-heavy operations and can instead reuse their results. The memoization design pattern is used in the type system of catscript. Types are cached, so if we need a similar type in the future, the system can pull the type form memory instead of creating an entirely new one. Should the type system be changed in the future, memoization could potentially cause problems, but at the moment the benefit to the running time of the program makes the use of the design patter worth it. Memoization is implemented into our program as follows:

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

Section 4: Technical writing

```
# Catscript Guide


## Introduction

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

```
var x = "foo"
print(x)
```
## Data Types
CatScript is statically typed, with a small type system as follows

**int** - a 32 bit integer

**string** - a java-style string

**bool** - a boolean value

**list** - a list of value with the type 'x'

**null** - the null type

**object** - any type of value

## Features
```

Expressions

Additive Expressions:
Catscript contains functionallity to use the + and - operatures.
When uses on Integerts it computes the mathimatical Sum or Difference.
When Given two string Concatination is performed.
```
Statement: 1+1
Result:2

Statement: 5-2
Result: 3

Statement: "Hello " + "World"
Result: "Hello World"
```
Comparison Expressions:
Catscript contains functionallity to use the >, >=, <, and <= operatures.
These can be used to compare Integer Values and will return the Boolean Values True or False.
```
Statement: 1 < 0
Result: True

Statement: 5 >= 5
Result: True

Statement: 2 < 500
Result: False
```

Equality Expressions:
Catscript contains functionallity to use the !=, == operatures.
These can be used to compare the equality of Integer Values and Strings and will return the Boolean Values True or False.
```
Statement: 1 == 1
Result: True

Statement: 1 != 2
Result: True

Statement: "Cat" == "Dog"
Result:
```

factor_expression = unary_expression { ("/" | "*") unary_expression };
Factor Expressions:
Catscript contains functionallity to use the / and * operatures.
When uses on Integerts it Multiplys or Divides the Vaules.
```
Statement: 6/3
Result:2

Statement: 51 * 2
```

```
Result: 102
```
```
unary_expression = ( "not" | "-" ) unary_expression | primary_expression;
#### Unary Expressions:
Catscript contains functionallity to use the not and - operatures.
not will flip boolean operators
"-" Will negate integer values
```
```
Statement: not true
Result: false

Statement: -(-1)
Result: 1
```
```
### Statements

#### For loops
Will Iterate through provided list and will perform Operation till iteration is
complete
```
```
Command: for x in [1, 2]{1 + x}
Result:
2
3
```
```
#### If Statement
Will perform operation if condition is met
```
```
Command:
var x:int = 5
if (x < 5){print(true)}
Result:
true
```
```
#### Print Statement
Will display result of expression to the terminal
```
```
Command:
print("The man is over there")
Result
The man is over there
```
```
#### Variable Statement
Will Assign an allias the result of an expression.
A data type can be assigned to the variable using the : operator
If no type is provided variable will be staticly typed
```
```
var x:String = "Cat"
var x = 12
```
```
#### Function Calls
Ability to define a block of operations that can be called from the code
Can be called from code with function calls
```
```
Function:
```

```
foo(x:string) {print(x)}

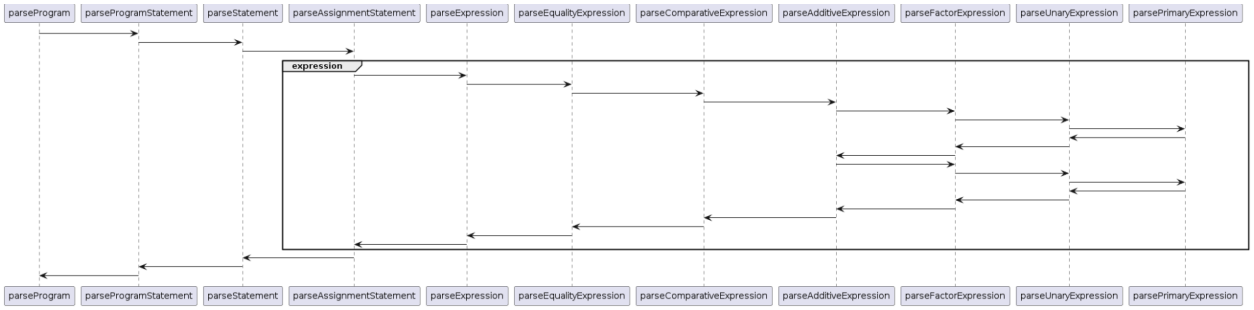
foo("cat")

Result: cat

```
```

Section 5: UML

The UML diagram provided in this document is a sequence diagram of the parsing of the statement ‘x = 1 + 1’. This sequence diagram exemplifies the recursive nature of the parser, giving an idea of how it works down the grammar until it arrives at the correct expression or statement, and then works back up the tree afterward.



Section 6: Design trade-offs

We had to choose between a parser generator and a recursive descent parser for our code. We chose recursive descent because it is fast enough and expresses the natural recursive nature of the grammar more obviously. Using a parser generator generally requires less code than a recursive descent parser, however, it is more difficult to understand what the code is specifically doing when using a parse generator over a recursive descent parser. On the other hand, a recursive descent parser, despite being more code-heavy, offers a much more accessible and understandable approach to the process. With a recursive descent parser, the code structure mirrors the grammar rules more closely, making it easier to trace and debug. Additionally, since it's implemented directly in code, there's no need to learn additional syntax or tools, which can streamline the development process and reduce dependencies.

Section 7: Software development life cycle model

We used Test-Driven Development to create our compiler. This methodology was highly effective in the creation of highly intricate projects such as this, as it makes it easier to identify where the issues of your code are occurring. However, while Test-Driven Development ensured the reliability and correctness of our compiler's functionality through a rigorous testing process,
```

adopting an agile approach could have provided additional benefits. Instead of using tests, Agile would provide a list of tasks that need to be completed for the project to be finished. This methodology, although useful, would not have worked as well for this specific project due to the intricacy of the codebase we were working with. I believe this would have slowed down development and made it more difficult to find where errors are occurring in the codebase.

Partner tests written by Terris Dietz:

```
@Test

public void advanceTokenizerTest(){

   assertTokensAre("(1 + 1) * n", LEFT_PAREN INTEGER, PLUS, INTEGER, RIGHT_PAREN,
STAR, IDENTIFIER, EOF);

   assertTokensAre("print (1)", STRING, RIGHT_PAREN, INTEGER, LEFT_PAREN)

}



@Test

public void parseNestedUnaryWithParensExpression() {

   UnaryExpression expr = parseExpression("not (not false)");

   assertEquals(false, expr.isNot());

   assertTrue(expr.getRightHandSide() instanceof UnaryExpression);}



@Test

public void parseFactorIsHigherPrecendenceThanAddwhenpairedwithParens() {

   AdditiveExpression expr = parseExpression("(1 * 1) + (1 / 1)");

   assertTrue(expr.isAdd());

   assertTrue(expr.getLeftHandSide() instanceof FactorExpression);

   assertTrue(expr.getRightHandSide() instanceof FactorExpression);}
```