Cover Page:
Compilers
CSCI 468
Spring 2024
Aden Hartman, Willow Berryessa

Section 1: Program

Submitted with this portfolio. Path: capstone/portfolio/source.zip

Section 2: Teamwork

I had one team member for this capstone project, Willow Berryessa. She contributed by writing three tests for the CatScriptParser, attached below, which ultimately strengthened our project in various ways. These tests ensured that print and function definition statements were working, as well as the string literals' ability to be assigned to an object. In addition, she wrote documentation for the parser which is also provided below. The documentation she wrote provided insight into CatScript's inner workings, allowing any beginner to understand its usage. I'd estimate her contributions to be equal to mine, as we both supplied each other with tests and parser documentation.

Willow's Tests:

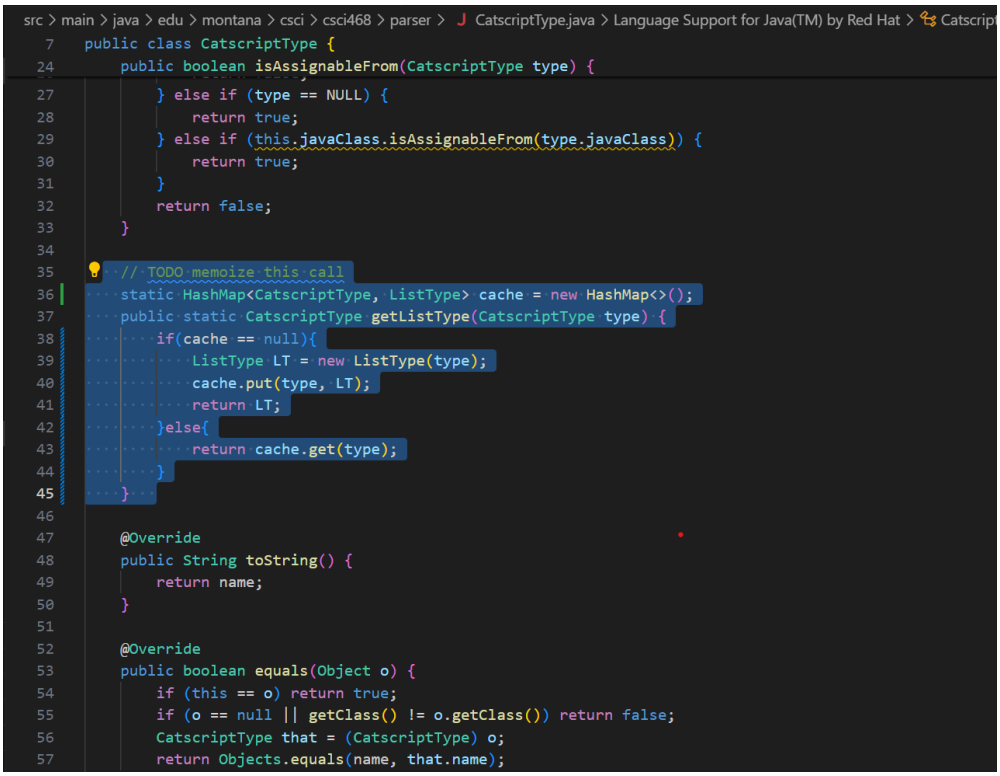
```
public class CatscriptPartnerTest extends CatscriptTestBase {

@Test public void parsePrintStatementWithoutExpressionFails() {
PrintStatement expr = parseStatement("print()", false);//verifies that print statement follows proper grammar
assertNotNull(expr);
assertTrue(expr.hasErrors()); }

@Test public void parseFunctionDefStatementWithoutBodyFails() {//verifies that parse func def statement follows proper grammar
FunctionDefinitionStatement expr = parseStatement("function x();", false);
assertNotNull(expr);
assertTrue(expr.hasErrors()); }

@Test
public void stringAssignableToObject() {
VariableStatement expr = parseStatement("var x : object = "Hello""); //makes sure string is assignable to object
assertNotNull(expr);
assertEquals("x", expr.getVariableName());
```

assertTrue(expr.getExpression() instanceof StringLiteralExpression); }
}

I wrote three tests, commented on them, and provided them to my partner. They can be found at PartnerTests.java in my demo directory.

Section 3: Design Patterns

Attached below is my implementation of memoization, which is a dynamic programming technique used for optimization. Though I didn't get to test it, the logic checks out. I chose to use a design pattern here instead of coding it myself not only because it was recommended, but also because its a good habit that will benefit me in the future.



Section 4: Technical Writing

```
# Features - Expressions and Statements
 ## statement =
##             for_statement
##             if_statement
##             print_statement
##             variable_statement
##             assignment_statement
##             function_call_statement;
 This is the first portion of the recursive descent in catscript. It
covers all the different types of statements.
```

Usage Example:
```
for(this in for this long){}
if(this){}
print(“bingo”);
int x = 35;
y=42;
bubbawins();

```

for_statement = 'for', '(', IDENTIFIER, 'in', expression ')', '{', { statement }, '}';
This class is the for statement class. This class is used to iterate through a for statement much like one you would see in python. This class allows you to declare what you want to happen in the body.
Usage Example:
```
for(x in list) {
print(x) }

```

if_statement = 'if', '(', expression, ')', '{', { statement },'}' ['else', (if_statement | '{', { statement },'}')];
This statement is used to check for a condition before it executes. This grammar allows for any type of expression to be the condition to check, including but not limited to booleans, and comparison expressions.
Usage Example:
```if(true){
 print(“Mr.Bart is my dog”) }
```
```
if(x<y){
print(“This was a great class”) }

```

print_statement = 'print', '(', expression, ')'
The print statement is used to display any message the expression says. This could be simple as words or as complicated as the return value from another method.If displaying a message in the form of words, parentheses must be present.
Usage Example:
```
print("I love my dog");
```

```

## variable_statement = 'var', IDENTIFIER, [':', type_expression, ] '=', expression;
This statement is used to store and manipulate values. WIth catscript, you must be assigning a value to something of the same type. For example, you cannot assign a boolean to 12.
Usage Example:
```
var x = 35
```

## assignment_statement = IDENTIFIER, '=', expression;
This statement is used to assign a value to a variable. Not to be confused with variable statements, this class allows you to reassign a value.
Usage Example:
```
String boo = “stinky” //variable statement
boo = "hoo"; //assignment statement
```

## function_declaration = 'function', IDENTIFIER, '(', parameter_list, ')' + [ ':' + type_expression ], '{',  { function_body_statement },  '}';
This type of statement is used to define a function , its requirements, and its execution code when the requirements are fulfilled. This class is used almost everywhere.
Usage Example:
```
function bubbawins(x, y, z) {
 print(z) }
```

## function_body_statement = statement | return_statement;
This statement is  the execution contents of the function declaration statement. This grammar helps better define the rules of the function body.
Usage Example:
```
function bingo() : int {return 35}
```

## return_statement = 'return' [, expression];
This Statement is used to define what a return statement looks like. This grammar aligns pretty heavily with the grammar for a return statement in java.
Usage Example:
```

```
function x() : int {
return 35}
```

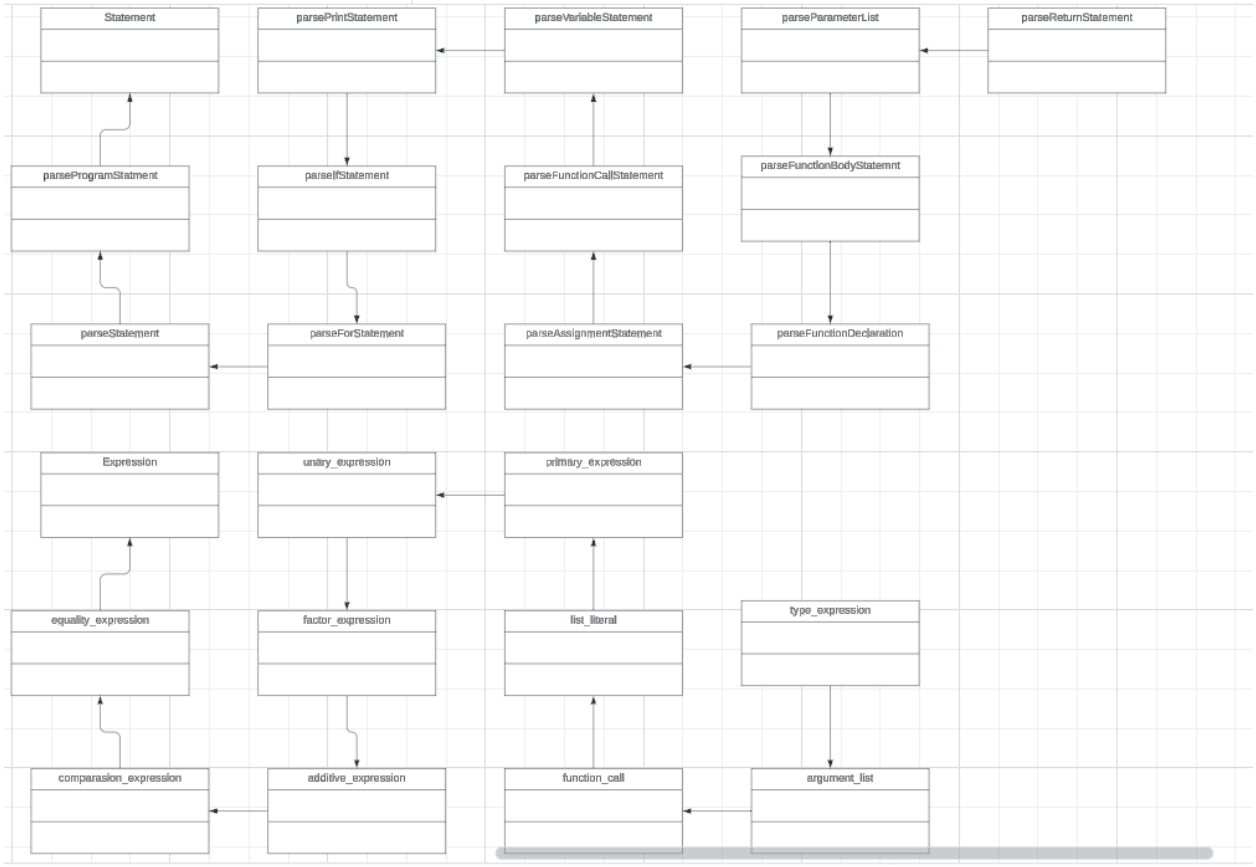
equality_expression = comparison_expression { ("!=" | "==") comparison_expression };
This Expression is used to check if two values are not equal to each other or equal to each other, depending on the sign used. This expression does sneak in the possibility of using a comparison expression as well.
Usage Example:
```
1 == 1
```

comparison_expression = additive_expression { (">" | ">=" | "<" | "<=") additive_expression };
This expression is used to compare the contents of one additive expression to another. This utilizes the less than, less than or equal to, greater than, or greater than or equal to sign.
Usage Example:
```
x >  35
```

additive_expression = factor_expression { ("+" | "-") factor_expression };
This expression is used to change the value of a variable to whatever the body of this statement adds to. It utilizes simple math by using the tokenizer to store the values
Usage Example:
```
35+ 42
```

factor_expression = unary_expression { ("/" | "*") unary_expression };
This expression is the multiplicative and division counterpart of the additive expression. It utilizes the symbols * and / to differentiate between the two of them. This grammar is only implementable on integers.
Usage Example:
```
35 * 42
```

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

```
This expression is used to evaluate boolean values. This expression  can
also use the symbol ! for not.
Usage Example:
```
!true
```

### function_call = IDENTIFIER, '(', argument_list , ')';
This expression is the last expression mentioned in the catscript grammar.
This expression is used to return a value from a function already defined.
This expression does not have a set type and will evaluate to whatever
type of identifier it is retrieving.
Usage Example:
```
bubbywins(mydiploma);
```

## Section 5: UML




## Section 6: Design trade-offs

Recursive descent and parser generators have many similarities, but their differences highlight their best usages. Recursive descent has proven to be easy to implement,
```

whereas a parser generator would be a bit more challenging. After doing some research, I found that parser generators tend to be better for complex grammars whereas recursive descent is better for educational purposes. Everything we experienced so far aligns with what credible sources have to say, so we're excited to try a parser generator.

Section 7: Software development lifecycle model

Test Driven Development worked incredibly well with this project. Having a class style that encourages individuals to work through issues rather than reporting to the professor allowed me to soak up alot more applicable knowledge. I'm used to classes like CSCI338, where a project description is given along with 1-2 vague tests that don't accurately depict how your program will be graded. This class seems to me like it should be the future class model. My personal feelings aside, the only flaw I can think of is a student not finishing the tests for one of the initial checkpoints that the other checkpoints rely on. A simple solution to this problem would be seeking help from the professor, so this class model encouraged independence while the bottom line remained the same, a win for any professor.