Capstone Portfolio

Aleksandr Means **(Team Member 2)**
&
Wei You **(Team Member 1)**

CSCI 468 Spring 2023

Wei You

Section 1: SRC Zip file is attached

Section 2:

Team member 2 contributed three tests and the documentation to our project, while I was responsible for the implementation. The documentation he provided me with is located in section 4 and the tests he provided me with are located below. The three tests are also located within the demo folder of the test folder of the project, with the class name of partnerTest, and the provided documentation can be found in the capstone folder named catscript.md. The estimated time contribution was about 50% 50% because it took about equal time and effort to get this project completed successfully and simply because of the ways that we used to complete it.

```
@Test
public void testingParenthesis() {
    AdditiveExpression expr = parseExpression( source: "1 + (3 + 5)");
    assertTrue(expr.isAdd());
    assertTrue(expr.getLeftHandSide() instanceof IntegerLiteralExpression);
    assertTrue(expr.getRightHandSide() instanceof ParenthesizedExpression);
}

@Test
void figureight() {
    assertEquals( expected: "10\n9\n8\n7\n6\n5\n4\n3\n2\n1\n0\n", compile(
            src: "function left(x : int) : int { if (x != 0) {print(x) return right(x - 1)} else {return 0} }\n" +
                    "function right(x : int) : int { if (x != 0) {print(x) return left(x - 1)} else {return 0} }\n" +
                    "print(left(10))\n"));
}

@Test
public void parseListLiteralExpression() {
    ListLiteralExpression expr = parseExpression( source: "[[2, 3, 4], [2], [3, 68, 91, 81]]");
    assertEquals( expected: 3, expr.getValues().size());
    ListLiteralExpression inside = (ListLiteralExpression) expr.getValues().get(0);
    assertEquals( expected: 3, inside.getValues().size());
    ListLiteralExpression inside2 = (ListLiteralExpression) expr.getValues().get(1);
    assertEquals( expected: 1, inside2.getValues().size());
    ListLiteralExpression inside3 = (ListLiteralExpression) expr.getValues().get(2);
    assertEquals( expected: 4, inside3.getValues().size());
}
```

Section 3:

The memoization pattern in the getListType method of the CatscriptType class in the compilers project utilizes a HashMap to cache instances of ListType. This approach ensures that when a ListType is requested for a specific CatscriptType, it is created only once and reused on subsequent requests, rather than being recreated each time. This method improves efficiency by reducing the number of object creations and the associated overhead, especially in performance-critical applications where getListType might be called frequently. Moreover, it helps in maintaining a smaller memory footprint since identical ListType objects are not needlessly duplicated. By using memoization, the system can scale better and operate faster as it avoids the computational cost of repeated instantiation.

```
static HashMap<CatscriptType, ListType> cache = new HashMap<>();
3 usages  new *
public static ListType getListType(CatscriptType type) {
    return cache.computeIfAbsent(type, t -> new ListType(t));
}
```

Section 4:

This is the documentation of the CatScript program that was provided by Team member 2.

```
# Catscript

## Introduction
Catscript is a scripting language that is easy to learn and highly flexible to use! The language supports the use of a
simple set of expression types, addition and multiplication operators, the ability to print to the standard
output, as well as basic control flow abstractions.

## Features

### Type System
Catscript supports a small simple set of expression types. The compiler uses type inference, but the type for the
variable can be declared.

* `IntegerLiteralExpression` - 32-bit integer expressions, which can be modified by simple arithmetic operations.
* `BooleanLiteralExpression` - simple true/false expressions.
* `StringLiteralExpression` - string expressions like those used in Java or C.
* `NullLiteralExpression` - used to signify when something is unassigned or empty.
* `ListLiteralExpression` - a list of elements with a specified type, if given. Lists are immutable once declared.

### IdentifierExpression
Catscript uses identifiers to store values for variables and make calls to functions.

### VariableAssignmentStatement
Variables in Catscript are assigned using the `var` command. If explicit type declaration is desired, a programmer may
also format the statement like so: `var foo : type`. This allows stricter definition of types when needed.
```
var x : int = 32
var y : string = "string"
```

### AdditiveExpression
Catscript supports the use of the `+` operator for arithmetic addition and string concatenation, as well as the use of
the `-` operator for arithmetic subtraction.
```
var x = 1 + 1   // x evaluates to 2
var y = 4 - 3   // y evaluates to 1
var z = "hello " + "world" // z evaluates to "hello world"
```

### Multiplication and Division
Catscript also supports the use of the `*` operator for multiplication, and the `/` for division. Note that division
will result in an integer value, so any non-integer evaluation will truncate the remainder of the result.
```
var x = 3 * 3 // x evaluates to 9
var y = 5 / 2 // y evaluates to 2
```

### ComparisonExpression
Comparison expressions can be made using `>`, `<`, `>=`, and `<=`. This allows a programmer to compare the values of
integers in a script
```
var x = 3
var y = 4

x >= x // true
x > y // false
x <= y // true
x < x // false
```

### EqualityExpression
Equality expressions, using `==`, allow a programmer to compare any two objects and determine if they are equal to one
another. This includes null values, strings, integers, booleans, and lists. Furthermore, inequality expressions are als
supported, using `!=`.
```
var a = 0
var b = 1

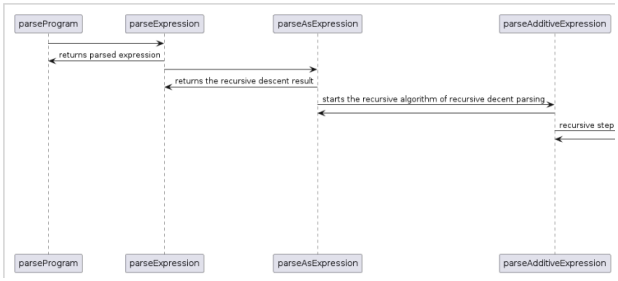
a == a // true
a != b // true
```

### UnaryExpression
Catscript has two unary operators. Using `not`, a user can express a logical negation of a boolean expression.
Meanwhile, using `-` negates an integer expression.
```
var x : int = 2
x = -x   // x evaluates to -2
var notPos : bool = not x > 0 // `x > 0` evaluates to 'false', thus `not x > 0` evlauates to 'true'
```

### ParenthesizedExpression
Expressions can also be written with parentheses to establish a desired precedence or to reduce ambiguity in more
complex expressions when necessary.
```
var x : bool = ("int 1" == ("int " + 1)) == true // x evaluates to 'true'
```

### SyntaxErrorExpression
The Catscript compiler uses Syntax Error Expressions to help the programmer find errors in code during evaluation.
```
var x : list<int = [1, 2, 3] // the user will be notified that there is a syntax error
                ^
```

### SyntaxErrorExpression
The Catscript compiler uses Syntax Error Expressions to help the programmer find errors in code during evaluation.
```
var x : list<int = [1, 2, 3] // the user will be notified that there is a syntax error
                ^
```

### PrintStatement
Print calls in Catscript allow the user to send text to the standard output. The print call formats the text to a new
line.
```
print("hello world")
```
will print
```
hello world
```

### IfStatement
Catscript includes if-else control flow statements, allowing a programmer to create conditional branching algorithms if
necessary. The statement requires a boolean expression enclosed within parentheses immediately following the `if` and
allows the programmer to define a body of statements that are enclosed within curly braces. An optional `else` can
follow, and must also be followed by a body of statements contained within curly braces.

```
if (x) {
  print("true")
} else {
  print("not true")
}
```
Consider the output if `x` is true; the output would be `true`. However, if it is false, then the output is `not true`

### ForStatement
For loops are another control flow tool implemented in Catscript. They are in the for-each format, and are designed to
iterate over a list, storing each value to a uniquely defined identifier at each loop starting at the beginning of the
list, and finally finishing when it has iterated through each element.

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

### FunctionDefinitionStatement
Functions in Catscript are an additional control flow abstraction that can be used by a programmer to write a script.
They are designed to allow a programmer to declare an indefinite number of arguments when defining the function, and
execute a body of statements when the function is called. A function can be declared with no arguments, and defining th
types of the arguments of the function is optional. The design of the Catscript compiler also allows a programmer to
call functions anywhere in the script relative to where the function was declared, allowing a valid script to
be constructed like this:
```
foo()

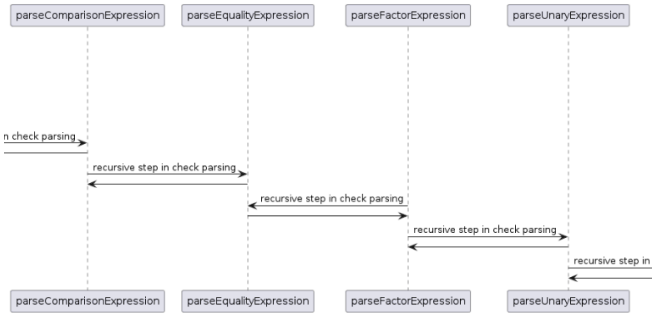
function foo() { print("bar") }
```
Note that this function does not contain a return statement. Since there is no type assigned to the function, the retur
is not necessary for the function to work, and can be omitted, unless it is desired for the purpose of managing control
flow. Otherwise, when a type *is* assigned to a function, it is necessary to return a value that is of that specified
type.

### FunctionCallExpression
When a function is typed, they can be called as expressions in a script. A function call expression is used when a
function needs to be used to evaluate to a value.
```
function foo() : int { return 0 }
print(1 + foo())
```
will print
```
1
```

### FunctionCallStatement
Since functions in Catscript are able to manipulate data in a script as well as make print statement calls, they can wo
as statement calls as well. Since functions can be defined without a return statement under certain conditions, it is
necessary that they are allowed to be used as statements since they will not evaluate.
```
var x = 3
function foo() {
  print(x)
  x = x - 1
  if (x != 0) { foo() }
}
foo()
```
will print
```
3
2
1
```
```

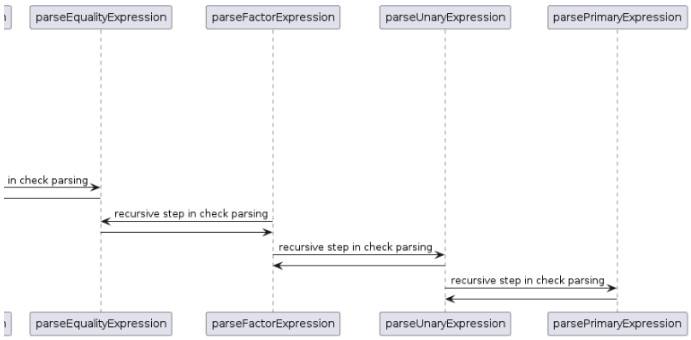
Section 5:

This is the UML diagram for the recursive descent parsing algorithm utilized by the compiler's parsing project.

```
'sequence diagram for parsing "1 + (1 + 1)"'

'calls parseExpression to start expression parsing
parseProgram->parseExpression
parseProgram<-parseExpression: returns parsed expression

'begins the parseAsExpression so that the expression
'can be parsed as an expression using recursive descent
parseExpression->parseAsExpression
parseExpression<-parseAsExpression: returns the recursive descent result

'calls the first part of the recursive tool and checks
'for an additive expression
parseAsExpression->parseAdditiveExpression: starts the recursive algorithm of recursive decent pa
parseAsExpression<-parseAdditiveExpression

'calls the second part of the recursive tool and checks
'for a comparison expression
parseAdditiveExpression->parseComparisonExpression: recursive step in check parsing
parseAdditiveExpression<-parseComparisonExpression

'calls the third part of the recursive tool and checks
'for an equality expression
parseComparisonExpression->parseEqualityExpression: recursive step in check parsing
parseEqualityExpression->parseComparisonExpression

'calls the third part of the recursive tool and checks
'for a factor expression
parseFactorExpression->parseEqualityExpression: recursive step in check parsing
parseEqualityExpression->parseFactorExpression

'calls the third part of the recursive tool and checks
'for a unary expression
parseFactorExpression->parseUnaryExpression: recursive step in check parsing
parseUnaryExpression->parseFactorExpression

'calls the third part of the recursive tool and checks
'for a primary expression
parseUnaryExpression->parsePrimaryExpression: recursive step in check parsing
parsePrimaryExpression->parseUnaryExpression
```







Section 6:

When comparing parser generators and recursive descent parsers, the choice often depends on the specific needs of the language being implemented and the preferences of the developers. Parser generators, like ANTLR or Yacc, provide a powerful tool for creating parsers from a high-level description of the grammar. These tools automatically generate the parsing code based on grammar rules defined in a formal syntax, often BNF (Backus-Naur Form). This can accelerate development and ensure that the grammar is correctly and consistently implemented, as changes to the grammar are centrally managed and propagated through the generated code. For a language like Catscript, which seems to have a structured type system and potentially complex syntax rules, a parser generator can offer a robust framework to handle parsing efficiently, minimizing manual coding errors and inconsistencies.

On the other hand, recursive descent parsers are hand-coded and provide the developer with finer control over the parsing process. This method involves writing functions or methods that directly implement the grammar rules. Each function typically corresponds to a non-terminal in the grammar, and these functions call each other recursively to match the input tokens against the grammar rules. Recursive descent parsers are particularly advantageous when the language features context-sensitive constructs or when the parsing logic needs to intertwine tightly with other aspects of the interpreter or compiler, such as type checking and error handling. In the context of Catscript, if the language demands intricate error reporting or specific handling of its unique types like CatscriptType and ListType, a recursive descent parser provides the necessary control to integrate these features seamlessly. Additionally, since recursive descent parsers do not require external tools, they are preferred for smaller or more experimental languages where external dependencies are to be minimized.

Section 7:

Test Driven Development (TDD) has been a central strategy in our approach to developing the capstone project, shaping how we plan, write, and verify our code. Fundamentally, TDD involves writing tests for specific functionalities before even writing the code that implements those functionalities. This method promotes a well-organized development process where requirements are translated into specific test cases, ensuring that all features are thoroughly planned and tested from the outset. Personally, I have found TDD immensely beneficial as it compels the team to clarify requirements and establish clear, measurable objectives for every feature before proceeding with implementation. This upfront investment in testing helps avoid the common pitfall of feature creep and ensures high coverage and code quality from early in the project lifecycle.

From a personal perspective, TDD aligns well with my methodical approach to problem-solving. It allows me to focus intensely on one specific aspect of the system at a time, ensuring that each piece of functionality is correctly implemented and robust against future changes before moving on. This approach to building a project not only enhances my sense of accomplishment but also instills a discipline of continuous verification, which is crucial for maintaining long-term code health. Moreover, TDD's emphasis on testing first reduces the likelihood of encountering severe bugs later in the development cycle, which can be both demoralizing and costly to fix. While TDD can sometimes slow the initial stages of development as tests are formulated and refined, I find that this is mitigated by the smoother and quicker integration phases, as well-developed tests pave the way for integrating complex systems more reliably.