Montana State University

Compilers Capstone Document

Joshua Green

Amanda Guilland

CSCI 468 Compilers

Carson Gross, Professor

Spring 2024

Section 1: Program

The source code for this capstone project can be found in the source.zip file, which is located in the same folder as this document.

Section 2: Teamwork

Team Member 1:

- Primary programmer
- Responsible for implementation of the tokenizing, parsing, evaluating, and compiling stages of the parser.
- Estimated work contribution: ~90% of total time contributed

Team Member 2:

- Provided technical documentation (see Section 4)
- Provided 3 tests as detailed below to test functionality of Team Member 1's code
- Estimated work contribution: ~10% of total time contributed

Team Member 1 was the primary programmer for this capstone project, completing checkpoints throughout the project that all built on one another to successfully develop the compiler. They started their work in the tokenizer located in CatScriptTokenizer.java, where they expanded on a few functions, such as consumeWhitespace() and scanIdentifier(). They also almost completely wrote the scanString() and scanSyntax() functions with only a small amount of those functions being provided upon starting this project. With that functionality in place, the compiler was able to break input strings into tokens and even figure out the type of each token. They then moved onto the parser located in CatScriptParser.java and completely or almost completely wrote the functions parseProgramStatement(), parseStatement(), parseForStatement(), parseIfStatement(), parseVarStatement(), parseFunctionDefinition(), parseAssignmentOrFunctionCallStatement(), parseReturnStatement(), parseEqualityExpression(), parseComparisonExpression(), parseFactorExpression(), parsePrimaryExpression(), and parseFunctionCallExpression(). The work for the parser was completed using the provided grammar for the CatScript language. Once these functions were implemented, the compiler was able to take a string input and parse it, figuring out what type of statement / expression it belonged to.

Team Member 1 then started work on evaluating and executing these expressions and statements. All coding for this checkpoint is located in the various .java files located in the statements and expressions folders in the parser folder. The majority of those files have an evaluate() or an execute() method which needed to be implemented. To clarify, there is a separate evaluate() or execute() method that needed implementing for every type of expression and statement utilized in CatScript. As the name suggests, implementing those methods allows for each type of expression / statement to actually be evaluated or executed. For example, if the compiler is given an equality expression, Team Member 1 wrote the code that will look at both sides of the expression and determine whether the values are equal to each other or not. This was done for the majority of .java files in the expressions and statements folders of the source code. The final checkpoint that Team Member 1 completed was the code generation targeting JVM bytecode for the compiler. In a basic sense, bytecode is machine code consisting of method codes that interact with an operand stack, slots, and fields. This work was done in the same files as the evaluation and execution checkpoint, but this time the goal was to implement the compile() function. To clarify again, there is a separate compile() method for every type of expression and statement with each compile() method actually generating bytecode to handle that specific type of expression / statement. Once all the compile() methods were implemented by Team Member 1, the primary coding for the capstone project was complete.

Team Member 2 wrote and provided technical documentation, which is a detailed description of the various features of the CatScript programming language. CatScript has many different types of statements and expressions that can be used, and they all have different formats. The documentation goes into detail as to what each type of statement or expression is, how it is formatted, and what an example of each would be. The technical documentation from Team Member 2 can be found in Section 4 of this document.

In addition to the technical documentation of the CatScript language, Team Member 2 also provided three tests for the capstone project to test various aspects of the CatScript language and the compiler itself. The first test ensures that a function can return the correct value in an if statement. The second test ensures that a function can have multiple different types as parameters. Finally, the third test ensures that a for loop can iterate over a list of strings. Those three tests are all passing and can be found below along with comments explaining what each one is testing and how it is being tested.

```
// Tests that a function can return the correct value in an if statement
@Test
public void functionWithIfStatement() {
    FunctionDefinitionStatement expr = parseStatement("function returnVal(x : int){if(x >"+
        "10) { return(x) } else { return(10) }} \n returnVal(10) \n returnVal(5)", false);
    // Make sure the expression exists
    assertNotNull(expr);
    // Check the name of the function is "returnVal"
    assertEquals("returnVal", expr.getName());
    // Check that there is only one param in the function
    assertEquals(1, expr.getParameterCount());
    //  Check that the param name is "x"
    assertEquals("x", expr.getParameterName(0));
    // Check the type of x is int
    assertEquals(CatscriptType.INT, expr.getParameterType(0));
    // Check there is only 1 statement in the function body
    assertEquals(1, expr.getBody().size());
}

// Tests that a function can have all types of expressions
@Test
public void functionDefinitionTypesWork() {
    FunctionDefinitionStatement expr = parseStatement("function checkAll(ints : int, " +
        "strings : string, objects, booleans : bool) {}");
    // Check the expression isn't null
    assertNotNull(expr);
    // Check the function is named "checkAll"
    assertEquals("checkAll", expr.getName());
    // Check it has 4 parameters
    assertEquals(4, expr.getParameterCount());
    // Check the first one is named ints
    assertEquals("ints", expr.getParameterName(0));
    // Check the fist one is of type int
    assertEquals(CatscriptType.INT, expr.getParameterType(0));
    // Check the second one is called strings
    assertEquals("strings", expr.getParameterName(1));
    // Check the second one is of type string
    assertEquals(CatscriptType.STRING, expr.getParameterType(1));
    // Check the third is called objects
    assertEquals("objects", expr.getParameterName(2));
    // Check the third is of type object
    assertEquals(CatscriptType.OBJECT, expr.getParameterType(2));
    // Check the fourth is called booleans
    assertEquals("booleans", expr.getParameterName(3));
    // Check the fourth is of type boolean
    assertEquals(CatscriptType.BOOLEAN, expr.getParameterType(3));
}

// Tests that a for loop can iterate over a list of strings
@Test
public void forLoopStrings() {
    ForStatement expr = parseStatement("for(i in [\"a\", \"b\", \"c\"]){ print(i) }");
    // Check expression is not null
    assertNotNull(expr);
    // Check the variable name is i
    assertEquals("i", expr.getVariableName());
    // Check the expression is an instance of a list literal
    assertTrue(expr.getExpression() instanceof ListLiteralExpression);
    // Check there is only one expression in the body of the statement
    assertEquals(1, expr.getBody().size());
}
```

Section 3: Design Pattern

A design pattern utilized in this capstone project is memoization. Memoization stores results of method execution in a data structure so that a method does not need to execute more than once for the same inputs. This helps increase program optimization as method executions are not needlessly duplicated.

Looking at the compiler source code for this project, this design pattern can be found in the CatscriptType.java file located in the parser folder, starting at line 36, just above the getListType() function. Below is the section of that file where memoization was implemented.

```
static Map<CatscriptType, CatscriptType> CACHE = new HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
    CatscriptType potentialType = CACHE.get(type);
    if (potentialType != null) {
        return potentialType;
    } else {
        ListType newListType = new ListType(type);
        CACHE.put(type, newListType);
        return newListType;
    }
}
```

If this compiler did not implement memoization, the getListType() function would simply take a CatscriptType as an input, create a ListType of the same type that was passed into the function, and return it. However, that is not the most optimized decision. There are only a handful of types in the CatScript language, which means that there is a high chance that this getListType() function could be called multiple times with the same input. This is where memoization comes in.

Taking a look at the first line in the above section, a HashMap called "CACHE" is initialized. Going into the function itself, instead of immediately creating a new ListType, the get() method is called on the CACHE, using the CatscriptType that was passed into the function to look in the CACHE map. This is where the positive effects on memoization really start to show. Say this is the first time the program is being run, so the CACHE is empty, meaning nothing will be found when the get() method is called. So, the potentialType variable will be null, which means a new ListType will be created with the CatscriptType passed into the getListType() function. But before the new ListType is returned, it is put into the CACHE with the original CatScript type as the key and the new ListType as the value. That means if the getListType() function is called again with the same input, then the CatscriptType will be used to look in the CACHE and the ListType that was just created will be retrieved and returned instead of creating a whole new ListType object.

If this function had been implemented without this design pattern, there could be many duplicate calls to the function where a new ListType would be created every single time. With the use of memoization, this function has been improved upon to avoid needlessly creating objects and to improve optimization of the compiler as a whole.

Section 4: Technical Writing

Introduction

CatScript is a simple scripting language with many different features and functionalities. Here is a simple example:

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

In this example, the variable x is assigned to the string 'foo'. The variable x is then used in the print statement to print 'foo'.

CatScript has a small type system. Below are the types available:

- string - a java-style string, enclosed in quotation marks
- int - a 32 bit integer
- bool - a boolean value (true or false)
- list< x > - a list of value with the type 'x', ex. list< int > is a list of integers
- null - the null type
- object - any type of value

As a statically typed language, CatScript has many expressions and statements that can be utilized.

Features

For Statement

A for statement is defined as a loop that iterates over a range of values. These values can be in the form of a list or a generic range of values. A for statement is declared by the keyword 'for', followed by an open parentheses, an identifier, the keyword 'in', an expression, close parentheses, and a body of at least one statement wrapped in brackets. The statement(s) could be used to print variables, perform mathematical computations, call a function, and more. On run, the loop will iterate over each value in the list or over the range of values. Each value can be assessed in the statement(s) using the identifier that is found in the parenthesis, as the identifier represents the value that the loop is currently on before iterating to the next.

Here is an example of a generic for loop with the CatScript formatting. The identifier in this example is i, with the value of i changing after each iteration. The data structure in this example is the list ["a", "b", "c"]. The statement of the body of the for loop in this example is a print statement, which prints the value of i. In summary, this for loop iterates over a list and prints the values.

```
for (i in ["a", "b", "c"]){
    print(i)
}
```

In this example the list ["a", "b", "c"] is iterated through using the identifier 'i'. This identifier is used in the print statement inside the statement body to print each value of the array. The result of this loop will be:

```
a
b
c
```

If Statement

An if statement is defined as a series of conditional statements with an optional negative case. This is called using the 'if' keyword with an expression wrapped in parentheses followed by a body of at least one statement that will execute if the condition in the expression is met. An if statement can be followed by an else-if statement to test more conditions. That way, if the first statement's conditions are not met, then you can test another set of conditions. However, an else-if statement is not necessary. Similarly, an else statement can follow an if statement (or an else-if statement, if one has been included), but again, it is not necessary. An else statement has no conditions and acts as a "default" that will execute if none of the other conditional statements are met. An else-if and else statement will both be followed by a body of at least one statement wrapped in brackets, just like an if statement. If the conditions of an if statement are met, then the code inside the statement executes. Otherwise, it moves on, either to a following else-if statement, a following else statement, or it simply moves onto the next line of code outside of these statements.

Here is an example of an if statement which utilizes an else-if statement as well as an else statement. The first if checks if the value of x is equal to 1 using an equality expression. If this condition is true, then it will print the string "One!". If x does not equal 1, x is checked if it is greater than 2 using a comparison expression in the else-if statement. In the event that it is, the program will print the string "Greater than 2!". If none of the conditions are met, then the program will go to the else statement and the string "Unknown" will be printed regardless of the value of x since there are no conditions in an else statement.

```
if(x == 1){
    print("One!")
}
else if( x > 2){
    print("Greater than 2!")
}
else{
    print("Unknown")
}
```

Print Statement

A print statement is defined as a statement that prints the value that is enclosed in the statement. This is called using the 'print' keyword followed by an expression enclosed in parentheses, which is the value that one wishes to print. The expression can be a direct value as well as a variable representing a value. Mathematical computations can also be executed inside the parentheses, which would lead to the statement printing the result of the computation. A function can also be called inside the parentheses, which would print whatever value is returned from the function. To summarize, a print statement can be used to print a value, and there are many ways that that value can be represented.

Here is an example of the string "Hello" being passed to the print statement, which would result in the string "Hello" bring printed:

```
print("Hello")
```

Here is an example of a print statement with a mathematical computation inside the parenthesis. This statement would add together the values 3 and 7 and print the result. Therefore, this print statement would print the value 10.

```
print(3 + 7)
```

Variable Statement

A variable statement is defined as a statement that assigns an expression to some identifier, which can then be used later in the program to refer to that expression. This statement can be identified by the "var" keyword, followed by the variable name. The variable name can be followed by an optional colon and a type to assign a type to the variable, but it is not necessary. The statement then must have an equal sign '=' and some expression that will then be assigned to the newly declared variable. As mentioned previously, the expression being assigned to the variable can take many forms, such as a direct value or the result of a mathematical computation.

Here is an example of a variable statement declaring the variable "x". The value of 5 is being assigned to the newly declared variable:

```
var x = 5
```

Here is an example of a variable statement declaring the variable "y". This example utilizes the optional type declaration as well. A colon follows the variable name "y", which is then followed by "string", signifying that the variable is representing a string value. The string "Hello" is assigned to the newly declared variable:

```
var y : string = "Hello"
```

Assignment Statement

An assignment statement is similar to a variable statement, as it is assigning a value to a variable. However, an assignment statement is defined as a statement that modifies a preexisting variable. It is called using the identifier name, such as the variable name, followed by an equal sign '=' and some expression. This will assign the value of the expression to the variable. Note, a variable must be declared using a variable statement before it can be modified using the assignment statement.

Here is an example of the value 5 being assigned to the variable x, which has already been declared. It is important to note that if the new value being assigned does not match the type of the variable that has already been declared, then an error will be thrown. Once the new value is assigned, the old value is essentially overwritten as it has been replaced with the new value:

```
x = 5
```

Function Declaration Statement

A function declaration statement is defined as a statement that declares a function to be used later in the program. After declaration, a function can be called using a function call statement (see Function Call Statement below). A function declaration statement starts with the 'function' keyword followed by an identifier, which is the name of the function. A parameter list wrapped in parentheses follows the identifier. A colon and a return type of the function can follow the closing parenthesis, but it's not necessary. There is then a function body with at least one statement wrapped in brackets. The statement(s) could be used to print variables, perform mathematical computations, call a function, and more. If a return type was specified for the function, then there will need to be a return statement that returns a value that matches that return type.

The parameter list is a list of parameters that are passed to the function. These can be just identifiers or identifiers followed by a colon with a type expression to declare an explicit type for that specific parameter.

A type expression can be an integer with the 'int' keyword, a string with the 'string' keyword, a boolean with the 'bool' keyword, an object with the 'object' keyword, or a list with the 'list' keyword. An explicit type for each parameter is optional in the function as they can be implied in the value declaration, but they are always encouraged wherever possible.

Here is an example of a function declaration statement declaring a function called 'test'. 'test' requires a string parameter, which is referred to as x in the function, and will return a boolean value. The function body contains an if statement. If x is equal to the string "Yes", then the function would return true. Otherwise, the function would return false.

```
function test (x : string) : bool{
        if(x == "Yes"){
                return(true)
        }
        else{
                return(false)
        }
}
```

Function Call Statement

A function call statement calls an existing function. The format of this statement varies depending on the function that it is calling, but the statement starts with the name of the function being called followed by parameters separated by commas in parentheses. If the function does not require any parameters, then the pair of parentheses would be empty. Similar to the print statement, the parameters can be the values themselves, variables representing the values, results of another function call, and more. As previously stated, there are many ways that a value can be represented, so there are many ways that a value can be passed to a function via a function call statement.

Here is an example of the function "test()" from the function declaration statement above (see Function Declaration Statement) being called. To be able to call a function, the function would have to have already been declared to successfully call the function. The number and type of the values passed into this statement must match the number and type of the parameters outlined in the function declaration. If a function is declared to accept only an integer and a string, the function call would need to pass in only an integer and a string separated by commas. The order of the parameters in the parentheses of the function call statement must match the function declaration as well. Since the string being passed into the function, "No", is not equal to "Yes", the test() function as declared in the Function Definition Statement section will return false:

```
test("No")
```

Note that the values passed to the function must be the same type as the function is expecting and the return value must be the same type as what is called the function.

Here is an example of a variable statement with an explicit type based on the return type of the function. The return type of test() is a boolean so the variable must be declared as a boolean:

```
var y : bool = test("Yes")
```

Since test("Yes") would return true, the value of y would be true.

Return Statement

A return statement is defined as a statement that returns an expression from a function. It can be called using the 'return' keyword. This return can then be followed by an optional expression. There are many different types of expressions (see below), so there are many different uses for a return statement. This statement can return variables of any type, but if the return statement is inside a function, it must return an expression with a type that matches the function's return type. A mathematical computation can be performed in this statement, and the result of that computation would be returned. Comparisons can also be computed, with the statement returning true or false depending on the result. To summarize, a return statement returns an expression, which can come in many different forms.

Here is an example of a return statement with no value:

```
return
```

Here is an example of a return statement that returns the boolean 'true':

```
return true
```

Here is an example of a return statement that utilized the factor expression to return the value of x times two:

```
return x * 2
```

Equality Expression

An equality expression is defined as two objects that are split by an equality symbol. An equality symbol can be two equal symbols '==' to check if the two objects are equal or an exclamation mark and an equal symbol '!=' to check if the two objects are not equal. An equality expression returns a boolean. If checking equality, true will be returned if the two objects are equal and false if they are not. If checking inequality, true will be returned if the two objects are not equal and false if they are equal. These can be used in many places throughout a program, but they are commonly used in conjunction with if statements to provide a condition to test as it returns a true or false value.

Here is an example of the integer 1 being checked for equality with the integer 2. This will result in the 'false' boolean since 1 and 2 are not equal:

```
1 == 2
```

Here is an example of the string "Hello" being checked for inequality with the string "Goodbye". This will result in the 'true' boolean, since inequality is being checked and they are not equal:

```
"Hello" != "Goodbye"
```

Comparison Expression

A comparison expression is defined as two objects, usually integers or variables representing integers, that are split by a comparison symbol. The supported symbols are greater than '>', less than '<', greater than or equal '>=', and less than or equal '<='. The first object is compared to the second using one of these symbols. The result of this comparison will be a boolean. A comparison expression is similar to an equality expression in the way that there are many places that a comparison expression can be used throughout a program, but they are commonly used in conjunction with if statements to provide a condition to test since they also return a true or false value.

Here is an example of the integer 1 being compared to the integer 5 using the less-than comparison expression. This will result in the boolean 'true':

```
1 < 5
```

Here is an example of the integer 3 being compared to the integer 6 using the greater-than comparison expression. This will result in the boolean 'false':

```
3 > 6
```

Here is an example of the integer 8 being compared to the integer 2 using the less-than-or-equal comparison expression. This will result in the boolean 'false':

```
8 <= 2
```

Here is an example of a comparison of the integer 1 and the integer 1 using the greater-than-or-equal comparison expression. This will result in the boolean 'true':

```
1 >= 1
```

Additive Expression

An additive expression is used to add two objects together or subtract one object from the other, such as adding two integers, concatenating two strings together, or subtracting the value of a variable from an integer. The format of this expression is a plus sign between the two objects to add or concatenate them together, or a minus sign between the two objects to subtract one from the other.

Here is an example of the strings "Hel" and "lo" being concatenated to create the string "Hello" using an additive expression:

```
"Hel" + "lo"
```

In the case of integers, the value of the second integer will be either added or subtracted from the first integer depending on the symbol being used. In the event a larger number is being subtracted from a smaller number, the result will be negative.

Here is an example of 6 being subtracted from 10, making the value 4:

```
10 - 6
```

Factor Expression

A factor expression is used to multiply together or divide two objects, such as integers or the values of variables. The format for multiplication is the '*' symbol between the two objects, while the format for division is the '/' symbol between the two objects. For division, the first object is divided by the second object.

Here is an example of the integer 1 being multiplied by the integer 4, making the value 4:

```
1 * 4
```

Here is an example of the integer 6 being divided by the integer 3, making the value 2:

```
6 / 3
```

Unary Expression

A unary expression in CatScript is defined as an expression with a symbol that reverses the value of an object. This can be done with the string 'not' or the minus '-' symbol.

Here is an example of the positive integer 1 being converted to a negative 1 using a unary expression:

```
-1
```

The minus sign operator can also be used on variables to turn the variable value negative. However, that would not permanently alter the value of the variable itself. An assignment statement using a unary expression would be required.

Here is an example of the boolean 'true' being changed to 'false' utilizing a unary expression with the "not" operator:

```
not true
```

Primary Expression

There are many types of expressions that fall under the umbrella of a Primary Expression. Below you can find descriptions and examples of the various types.

Integer Literal Expression

An integer literal expression is defined as any non-decimal value that directly represents said value.

Here is an example of the integer literal expression for the number '15':

```
15
```

String Literal Expression

A string literal expression is defined as a string of characters wrapped by quotation marks.

Here is an example of a string literal expression "Hello!":

```
"Hello!"
```

Boolean Literal Expression

A boolean literal expression is defined as an expression having a value of either 'true' or 'false'.

Here is an example of a 'true' boolean literal expression:

```
true
```

List Literal Expression

A list literal expression is defined as a series of expressions of the same type. A list must be wrapped with both an open bracket and a close bracket, and each expression inside the brackets must be separated by a comma.

Here is an example of a list containing the strings "a", "b", and "c":

```
["a","b","c"]
```

Null Literal Expression

A null literal expression is defined as the value 'null'. It is the only expression of the type null.

Here is the only example of a null literal expression available in CatScript:

```
null
```

Parenthesized Expression

A parenthesized expression is defined as any expression contained within a complete set of parentheses. This can be utilized to prioritize mathematical equations by order of operations. Any open parenthesis must be closed at some point by a closing parenthesis or an error will be thrown.

Here is an example of an additive expression contained within a complete set of parenthesis:

```
(5 + 6)
```

Identifier Expression

An identifier expression is defined as any identifier used to reference another value, such as a variable representing a value or a name of a function representing the function itself.

Here is an example of the identifier expression 'x', which could be an example of a variable name, function name, etc.:

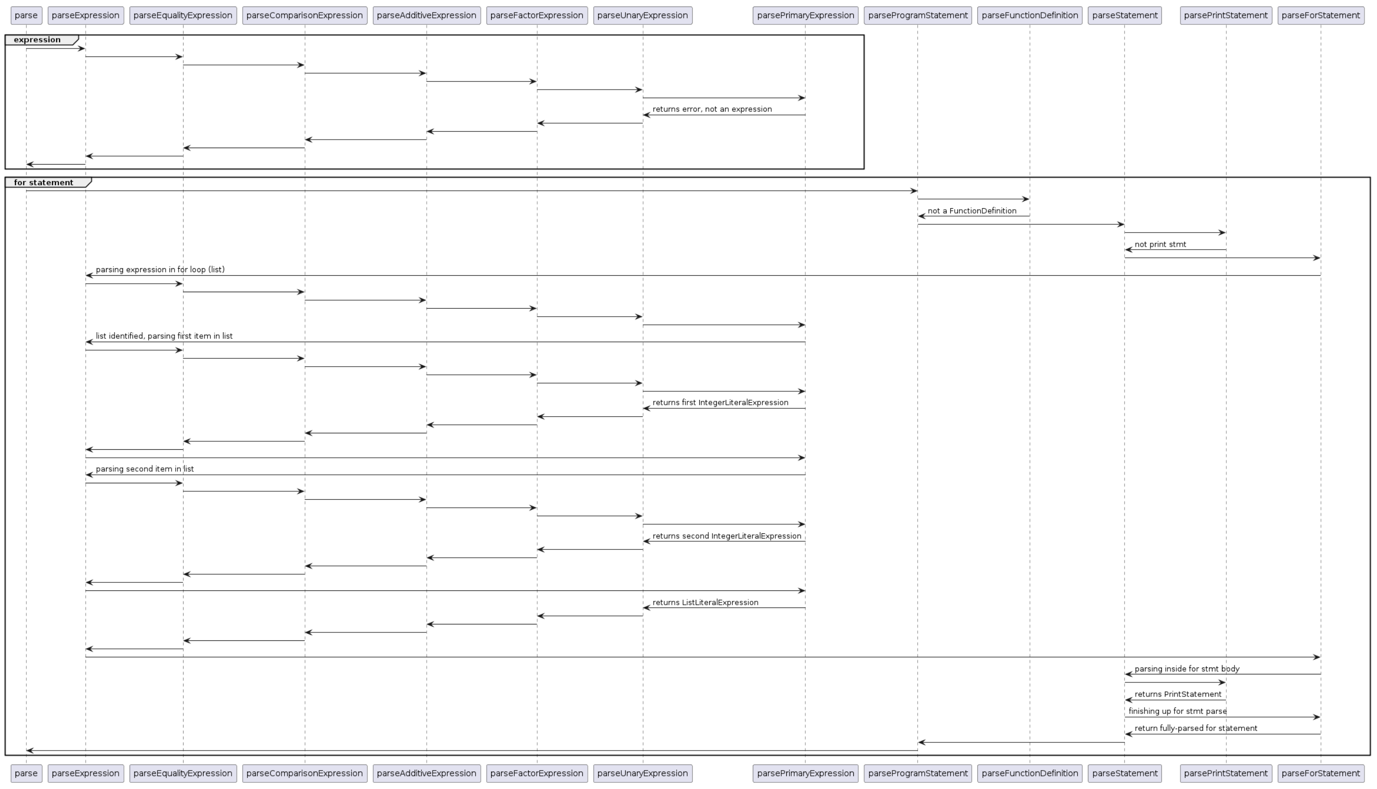
```
x
```

Section 5: Diagram

The statement that will be demonstrated by a sequence diagram for this section is a for statement iterating over a list with a print statement in the body. Below is the exact statement that is parsed in the diagram.

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

Below is the sequence diagram that demonstrates the process of parsing the above statement with the compiler for this capstone project.


parse
parseExpression
parseEqualityExpression
parseComparisonExpression
parseAdditiveExpression
parseFactorExpression
parseUnaryExpression
parsePrimaryExpression
parseProgramStatement
parseFunctionDefinition
parseStatement
parsePrintStatement
parseForStatement
expression
returns error, not an expression
for statement
not a FunctionDefinition
not print stmt
parsing expression in for loop (list)
list identified, parsing first item in list
returns first IntegerLiteralExpression
parsing second item in list
returns second IntegerLiteralExpression
returns ListLiteralExpression
parsing inside for stmt body
returns PrintStatement
finishing up for stmt parse
return fully-parsed for statement

Before the parser can get to parsing the statement, it needs to check whether the input is an expression first. The parser for this capstone project uses Recursive Descent Parsing. Working in close connection with the CatScript language grammar, the main idea for Recursive Descent Parsing is that for each production in the grammar, a method is named after it, such as parseEqualityExpression() to parse an equality expression. Then, inside that method, all the methods that are defined on the right hand side of the grammar for that specific production are called. This results in the parser working its way deeper into the grammar after every call before returning all the way back out to where the first call was made once the parser found the correct method to parse the given statement or expression.

This Recursive Descent Parsing idea can be seen in the way the diagram has multiple arrow-like shapes where the parser made multiple calls before returning back to where the first method was called. Starting in the parse() function, the parser slowly works its way deeper by calling parseExpression(), parseEqualityExpression(), parseComparisonExpression(), parseAdditiveExpression(), parseFactorExpression(), parseUnaryExpression(), and finally, parsePrimaryExpression(), which is the deepest level of the expression calls. Since the given input is a statement and not an expression, parsePrimaryExpression() returns an error, working its way back through all the calls until it is back at parse().

Now, the parser tries to parse the input as a statement. From parse(), the parser calls parseProgramStatement() followed by a call to parseFunctionDefinition(). The input is not a function definition, so the parser returns back to parseProgramStatement(). From there, the parser calls parseStatement() followed by a call to parsePrintStatement(). Our current token is still the first "for" keyword in our input and is therefore not a print statement, so the parser returns back to parseStatement(). The parser then calls parseForStatement(), which matches the token we are looking for. Once the parser has found what type of statement the input is, it starts looking at the content inside the parentheses. It makes note of the identifier "x" before calling parseExpression() to parse the expression in the parentheses, which in this case is the list [1, 2]. From here, the parser works its way through the expression calls, once again making it to the deepest level with a call to parsePrimaryExpression(). Here, the parser identifies the first bracket of the expression to mean that it is a list, and it calls parseExpression() again to parse whatever is inside the list. The parser once again calls its way down to parsePrimaryExpression(), where it matches the first value inside our list, "1", to an IntegerLiteralExpression. The parser returns back all the way to where it was first called to look at the inside of the list, which was the call in parsePrimaryExpression() after the first bracket of the list was identified. Since we still have the value "2" inside our list, those steps to return an IntegerLiteralExpression repeat, calling parseExpression() again before making recursive calls until it is once again at parsePrimaryExpression(). The "2" value is identified as an IntegerLiteralExpression and returned all the way back to where parseExpression() was called for the second time in parsePrimaryExpression(). The closing bracket of the list is identified, and with the list now fully parsed, the parser returns a ListLiteralExpression to where it was first called, which is back in parseForStatement() when the expression after the identifier needed to be parsed. Now, it is time to parse the body of the for statement. The parser grabs the first (and only) statement of the body, which is the "print(x)" statement, and calls parseStatement(). The first call the parser makes is to parsePrintStatement(), where it identifies that that is the type of statement that we are parsing. It parses the statement and returns a PrintStatement back to parseStatement() before returning back to where the body was being parsed in parseForStatement(). With the body of the for statement now parsed, the parser finishes up parsing the for statement before again working its way back to where it was initially called, returning a ForStatement back to parseStatement(), then to parseProgramStatement(), and finally, back to parse() which was called at the beginning of parsing the for statement. The provided for statement has now been fully parsed with this project's parser utilizing Recursive Descent Parsing.

Section 6: Design Trade-offs

As mentioned in Section 5, the compiler for this capstone project utilizes Recursive Descent Parsing, which uses recursive method calls to parse inputs in regard to the targeted grammar. However, a popular technique in the world of academia is Parser Generators. A parser generator is a program that takes some sort of language input, typically a lexical grammar in the form of a regular expression or a language grammar in EBNF notation, and generates a parser for it. One of the more popular parser generators is ANTLR, which stands for "Another Tool for Language Recognition". There are some advantages to using a parser generator. For example, since the parser is generated, there can be less code that needs to be actually written by the developer. Furthermore, there is also less infrastructure that the developer would need to immediately know about and understand to be able to use the generated parser.

On the other hand, there are also quite a few disadvantages to parser generators when compared to the recursive descent parsing used in this project. First, because the parser is generated for the developer when using a parser generator, there is a lot of obscure syntax for things that should be obvious. This would of course make it very difficult to debug and fix any errors that may arise since the developer is not familiar with the inner workings of the generated parser. Furthermore, this technique does not give developers a good feel of the naturally recursive aspect of grammars. In addition, while parser generators may be popular in academia, they are not very common in industry, so gaining experience with parser generators may not be very helpful in real-world applications outside the classroom.

In contrast to parser generators, recursive descent parsing has many more advantages. First, one of the main advantages is that since recursive descent already mimics the naturally recursive aspects of grammar, writing a recursive descent parser is actually quite simple. Furthermore, since the developer actually writes the parser themselves, it is easier to use and understand compared to a parser that is generated with obscure syntax. This makes it easier to debug and fix errors associated with the parser when recursive descent parsing is used. In addition, since the parser is handwritten, the developer has full control and can tailor the parser to fit whatever needs they have that need to be met. Finally, in regard to working in industry, recursive descent parsers are much more common than parser generators, so it is beneficial to gain experience with recursive descent parsing.

While it may be more time-consuming to manually develop a recursive descent parser compared to using a parser generator, recursive descent parsing still has many more advantages. Writing the parser instead of generating it allows the developer to have greater knowledge of how the parser actually works, which in turn helps with identifying and fixing any potential errors in a timely manner. Recursive descent parsing is also a very popular technique for developing parsers in industry work, so it is helpful to gain experience using this approach. Overall, recursive descent parsing is very useful and convenient with advantages extending well beyond the classroom, making it a great technique to use for the compiler for this capstone project.

Section 7: Software Development Life Cycle Model

Test-driven development (TDD) is the software development process used for this capstone project. Test-driven development has a very short development cycle that repeats as necessary. It starts with an initially failing test being written that tests specific functionalities or desired improvements. Next, the minimum amount of code needed to get the test to pass is then written. The code is then refactored and touched up until it is of an acceptable standard while ensuring the test is still passing. This development cycle is repeated to thoroughly test functionality in all parts of the software. One key idea for this development model to reiterate is that the test cases are written before the actual function or feature has been written to ensure that the developed software meets all the desired requirements. The repetition of this cycle allows for adequate test coverage of the software being developed.

In regard to this capstone project, there were 4 separate checkpoints (tokenizing, parsing, evaluating / executing, and generating bytecode) which were described in more detail in Section 2 of this document. Each checkpoint had numerous failing tests, and various functions and methods had to be written and implemented to meet the necessary requirements and get those tests to pass. Over all 4 checkpoints, there were just about 150 tests that tested different aspects of the compiler, such as parsing a print statement or evaluating a function declaration with multiple arguments. The tests for each checkpoint tested many different features of each functionality, such as strange inputs that would still need to be properly handled. The provided tests that were initially failing helped ensure that the compiler for this project was working as expected and that there weren't any missing or flawed functionalities.

Test-driven development worked very well for this project. As previously mentioned, the tests were very thorough so the different features and functionalities were tested in many different ways. Getting the tests to pass certainly inspires confidence in the code that is being written. The tests are also helpful in the development of the code itself. For example, if a test is failing because of a wrong output, the expected and actual outputs can be compared to see what is different and of course can be used to determine what is making them different. The tests themselves certainly came in handy when debugging the code as it's easier to narrow down what specific functionality is causing the tests to fail. As a bonus, once all the errors have been fixed, it is incredibly satisfying to run the tests and see all of them go green. Overall, test-driven development was a very helpful and rewarding software development model to use for this capstone project as it helped to ensure that the compiler and its various functionalities were thoroughly tested.