Matthew Keck, Ivan Cline
CSCI-468
5/3/2024

CSCI-468-Capstone

Introduction

Catscript is a strongly typed, compiled programming language. The Catscirpt compiler has three main computation phases: tokenization, parsing, and completion. Catscript is a statically scoped language, meaning that local variables declared in loops and functions precede global variables declared outside of them. The recursive descent algorithm is used to parse Catscript programs. The target of the compilation phase is Java byte code.

The following document will describe our team's process of creating the Catscript compiler. The first section, Program, has a link to a zipped source file containing the code for the Catscript compiler. The second section, Teamwork, describes each team member's contribution to this project and the estimated time the contributions took. The third section identifies the design pattern memoization we used in our Catscript compiler.

The fourth section is a technical document that describes two main features of the Catscript programming language statements and expressions. The fifth section gives a UML diagram our group made, which describes how our Catscript compiler parses a for statement. Section six discusses our team's design decisions and the trade-offs involved with our decisions. The final section describes the model we used to develop our capstone project.

Program

The Catscript compiler source code in the link (Catscript Source Code) was created with Java SE 22.0.1.

Teamwork

For our capstone project, I created a program that contains all the logic for tokenizing, parsing, evaluating, and compiling the Catscript programming language we designed in this class. The actual program can be viewed in the first section of this documentation. My teammate contributed the documentation in this file and added three tests to the test suit. My partner's tests are in the code block below this paragraph. The first test ensured that my compiler does not allow floating point numbers, which is illegal in Catscript. The second test ensured that my compiler does not enable function definitions to be declared inside another function definition, as it is not allowed in the Catscript grammar. The third and final test ensured the program could execute nested unary expressions. My partner estimates that the creation of this document and the tests took him eight hours to complete. Creating the Catscript compiler took me an estimated forty hours of programming.

```
package edu.montana.csci.csci468.demo;
```

Matthew Keck, Ivan Cline
CSCI-468
5/3/2024

```
import edu.montana.csci.csci468.CatscriptTestBase;
import edu.montana.csci.csci468.parser.ParseErrorException;
import
edu.montana.csci.csci468.parser.statements.FunctionDefinitionStatement;
import edu.montana.csci.csci468.parser.statements.VariableStatement;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

public class PartnerTests extends CatscriptTestBase {



    /*
    The correctRestrictionsOnIntType test ensures that floats cannot be
parsed for CatScript. CatScript does not allow
    for floats to exist at all, so this test also ensures that we cannot
hide floats in lists, and it also ensures that
    floats cannot be created through the use of the division operator.
     */

    @Test
    public void correctRestrictionsOnIntType(){

        assertThrows(ParseErrorException.class, () -> {
            VariableStatement expr = parseStatement("var x = 1.0");
        });
        assertThrows(ParseErrorException.class, () -> {
            VariableStatement expr = parseStatement("var x: list =
[1.0,2.0]");
        });
        assertEquals("0\n",executeProgram("var x: int = (0/3)\n"
                +"print(x)"));
    }

    /*
    The functionCallInFunctionCall_NoFunctionDefInFunctionDef tests ensure
that function definitions cannot be nested within
    other function definitions, as this is NOT allowed in the CatScript
grammar. This test, however, also ensures that you can include
    a function call within a function definition, so long as the function
```

```
you are adding within that function definition is defined
    elsewhere.
    */
@Test
    public void functionCallInFunctionDef_NoFunctionDefInFunctionDef(){
        assertEquals("2\n4\n6\n8\n10\n", executeProgram(
                "function printValueTimes2(x : int){\n"+
                        "print(2*x)}\n"+
                        "function starter(){\n"+
                        "for(i in [1,2,3,4,5]){\n"+
                        "printValueTimes2(i)}}\n"+
                        "starter()"
        ));

    assertThrows(ParseErrorException.class, () -> {
        FunctionDefinitionStatement def = parseStatement(
                "function outerDef(){\n" +
                " function innerDef() {" +
                        "return 100 }\n" +
                "}\n");
    });
}

    /*
    The nestedUnaryExpressions tests ensure that nested unary operations
using the "-" symbol or the "not" key word are
    correctly evaluated, and will continue to negate itself if called
recursively. This test also ensures that the unary
    expressions are evaluated concerning their correct precedence in the
grammar, which means they should be evaluated before
    all addition, subtraction, multiplication, division, comparison, and
equality operations.
     */
    @Test
        public void  nestedUnaryExpressions_UnaryExpressionPrecedence(){
        assertEquals("-1\n",executeProgram(
                "print(--6 + -----1 * --7)"));

        assertEquals("true\n",executeProgram(
                "print(not not not not not false)"));

        assertEquals("true\n", executeProgram(
```

Matthew Keck, Ivan Cline
CSCI-468
5/3/2024

```
                "print(not not false != not not true)"
        ));

        assertEquals("true\n", executeProgram(
                "print(--1 != ---1 + 1)"
        ));
        assertEquals("-9\n", executeProgram(
                "function returnNegatedPlus1(x : int) : int {\n" +
                        "return -x + 1" +
                        "}\n" +
                        "print(returnNegatedPlus1(" +
                        "returnNegatedPlus1(" +
                        "returnNegatedPlus1(10)" +
                        ")))"
        ));
    }
}
```

Design Pattern

One of the design patterns we used in our Catscript compiler is memoization. The code for the memoization pattern can be found on line 35 in this file path csci468\parser\CatscriptType.java, or it can be viewed in the code block below. When creating our list types, we used the memoization pattern to increase memory efficiency. If we didn't use the memoization pattern, we could have multiple instances of the same list type; instead, we store the list type when we create it, and if it already exists, we get it from storage instead of creating a new list type instance.

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

Matthew Keck, Ivan Cline
CSCI-468
5/3/2024

Technical Document

CatScript Comments

The CatScript programming language allows users to denote certain pieces of their written code that are to be ignored by the compiler so that users can leave notes for later that do not affect the code's compilation. CatScript comments are denoted by a (//) at the beginning of the line at which they are to be ignored. If a (//) is in front, the parser will ignore the rest of the line and treat the comment as whitespace.

```
var greeting = "hello" //inline comment example
// everything on this line will be treated as a comment :)
print(greeting);
```

```
OUTPUT: "hello"
```

CatScript Print Statement

The CatScript print statement takes a single expression as an argument and displays that argument from a string buffer to the user. The print statement also appends a newline character to the end of the string buffer for each argument before displaying it to the program's user.

```
print("Hello world")
print(123)
print(true)
print(null)
```

Below is what the raw output of the above program would look like.

```
OUTPUT: "Hello world\n123\ntrue\nnull\n"
```

However, for this documentation, I will deliberately reformat the output of the print statement to look like the output below. A comma will separate each print statement, and the outputs will be the actual expressions after evaluation and compilation.

```
OUTPUT: "Hello world", 123, true, null
```

Matthew Keck, Ivan Cline
CSCI-468
5/3/2024

CatScript Expressions

Integer Literals

Integer literal expressions, represented by the type “int” in CatScript, refer to discrete integers that can be used within your code. These expressions refer specifically to the base representation of a number in CatScript. Integer literal expressions can be used to do mathematical operations or to exist as numbers.

```
var x: int =  100
var x =  2
```

Boolean Literals

Boolean literas in CatScript, represented by the CatScript type (bool), can either be “true” or “false”. Boolean literal expressions can change control flow within if statements, in equality and comparison expressions, or just exist as values on their own.

```
var x: bool = true
var y = false
```

String Literals

String literals in CatScript, represented by the CatScript type (string), contain a sequence of characters that can be declared and used again within your program. To denote a string, you must wrap any character sequence in double quotes, such as in the examples below. CatScript does also allow string concatenation, though strings are immutable.

```
var x = "Hello World!"
var y = "!@&#(!@&&$@_)null!@123HELLOWORLD"
```

List Literals

List literals, represented by the CatScript type (list<component_type>), is a complicated data structure in CatScript. Lists are iterable collections of CatScript expressions, which we will call components. The order of the components in the list is maintained when the list is created.

The component type of a list can either be declared directly by a user through a variable assignment statement, or the type can be inferred. If a CatScript list’s components are all of a single type, let’s say (int), for example, CatScript will treat that list as a (list<int>) type. However, if a CatScript list’s components have two different types, such as (int) and (string), CatScript will treat that as a list of type (list<object>).

Lists can also store other lists within themselves. For example, let’s assume that a list contains multiple other lists of integers. The type of this list would be (<list<list<int>>), and can also be determined during runtime. Finally, lists are immutable and are read-only.

Matthew Keck, Ivan Cline
CSCI-468
5/3/2024

```
var x = [1,2,3,"hello"]
var y = [x,[3,4,5]]
var z: list<int> = [1,2,3,4]

print(y)
```

```
OUTPUT: [x,[3,4,5]]
```

Null Literals

Null literals in CatScript, represented by the CatScript type (null) and the value null, denote the absence of a CatScript object. The null value is assignable to any other object in CatScript except void.

```
var x_1: int = null
var x_2: bool = null
var x_3: string = null
var x_4: list<object> = null
var x_5: object = null
```

Additive Expressions

An additive expression in CatScript is a binary operation denoted through a (+) or (-) symbol sandwiched between two expressions for which you would like the operation to be performed. Additive expressions can perform addition and subtraction on two CatScript integers. CatScript addition and subtraction has a lower precedence than multiplication and division and is left associative. Additive expressions will either be evaluated to a CatScript string or integer.

The (+) symbol has another purpose; it can also be used for string concatenation. If you try to add two strings, concatenation will be applied. If you try to add a string and an integer or an integer and a string, CatScript will cast the integer to a string and perform concatenation automatically. Strings and integers are the only types in CatScript that can be utilized in additive expressions. See the examples below.

```
// Add and subtract integers
var add_ints = 2+2
var sub_ints = 2-2

print(add_ints)
print(sub_ints)

// String concatenation and casting example
var str_cat_1 = "hello" + " world"
```

Matthew Keck, Ivan Cline
CSCI-468
5/3/2024

```
var str_cat_2 = "hello" + 1
var str_cat_3 =  1 + "world"

print(str_cat_1)
print(str_cat_2)
print(str_cat_3)
```

```
OUTPUT: 4, 0, "hello world", "hello1", "1world"
```

Factor Expressions

A factor expression in CatScript is a binary operation denoted through a (*) or a (/) symbol sandwiched between two expressions for which you would like the operation to be performed. The (*) symbol represents the numeric multiplication operation in CatScript, and the (/) represents numeric division. Factor expressions will be evaluated using a CatScript integer.

The binary operations associated with factor expressions can only be applied to CatScript expressions that will evaluate to CatScript type integers. CatScript multiplication and division operations are left-associative and have higher precedence than addition and subtraction.

```
//Simple factor expression example
var x_fact = 4*3
print(x_fact)

//Factor expression, higher precedence than additive expression
var precedence_demo = 10+5*4/10-1
print(precedence_demo)
```

```
OUTPUT: 12, 11
```

Comparison Expressions

A comparison expression in CatScript is a binary operation denoted through the use of a (<=), a (>=), a (<), or a (>) symbol sandwiched between two expressions for which you would like the operation to be performed. The (<=) symbol represents “is less than or equal to,” The (>=) symbol represents “is greater than or equal to,” The (<) symbol represents “is less than,” and The “>” symbol represents “is greater than.”

The binary operations associated with comparison expressions can only be applied to CatScript expressions that will evaluate to CatScript type integers. Comparison expressions will always be evaluated using a boolean. CatScript comparison expressions are left-associative and have lower precedence than additive expressions.

Matthew Keck, Ivan Cline
CSCI-468
5/3/2024

```
print(100 >= 100)
print(100 <= 100)
print(10 <= 100)
print(10 < 100)
print(10 >= 100)
print(10 > 100)
```

```
OUTPUT: true, true, true, false, false
```

Equality Expressions

An equality expression in CatScript is a binary operation denoted through a (==) or a (!=) symbol sandwiched between two expressions for which you would like the operation to be performed. The (==) symbol represents “is equal to,” and the (!=) symbol represents “not equal to.” The binary operations associated with equality expressions can only be applied to CatScript expressions that will evaluate to CatScript type integers, booleans, or strings. However, the equality expression used on strings will check whether the two are the same string object instances. On the other hand, the equality expression used on integers and booleans will just check if the values are the same. Equality expressions will always be evaluated to a boolean. CatScript equality expressions are left-associative and have lower precedence than comparison expressions.

```
//Equality expressions for integers
var x = 10
print(x == 10)
print(x != 10)

//Equality expressions for booleans
var y = true
print(y == true)
print(y != true)
```

```
OUTPUT: true, false, true, false
```

Unary Expressions

There are two symbols in CatScript associated with unary expressions: the (-) symbol and the (not) keyword. Unlike binary expressions, unary expressions require the symbol before the expression to which you are trying to apply the unary operation. For example, the unary expression “not true” will evaluate and return the negated boolean value. The (-) operation will negate any integer expression, returning the negated integer value. CatScript unary expressions are of higher precedence than factor expressions, additive expressions, comparison expressions, and equality expressions.

Matthew Keck, Ivan Cline
CSCI-468
5/3/2024

```
//negate an integer
var x = 1
var x_neg = -1

//negate a boolean
var y = true
var y_neg = not true

print(x_neg)
print(y_neg)
```

```
OUTPUT: -1, false
```

Parenthesized Expressions

The CatScript language allows programmers to deliberately control the precedence and associativity for which their code is compiled and evaluated. Anything put within parenthesis will be recognized as having higher precedence than anything else not within parentheses and will always be evaluated first. If you have multiple expressions within one pair of parentheses, the same precedence rules of the CatScript language will be applied within.

```
//Change of precedence example
var x = 9*10-2
var x_paren = 9*(10-2)
var y_paren = (9*10-2)
print(x)
print(x_paren)
print(y_paren)
```

```
OUTPUT: 88, 72,88
```

Identifier Expression

An identifier is a name used to refer to some other data in the program. Identifier names cannot start with numbers but can be included later in the identifier name.

```
//Variable assignment statement
//the identifier in this situation is "x1"
var x1: string =  "hello";
```

Matthew Keck, Ivan Cline
CSCI-468
5/3/2024

CatScript Statements

Variable Statement

There are many nuances and specific implementation details about variable assignment statements in the CatScript programming language. Variable assignment statements will always begin with the keyword (var) and contain some identifier that will act as the variable's name. There will always be an (=) symbol and an expression, or value, on the right-hand side to which the variable will be set.

The location of the variable assignment statement in the program will determine the scope within which the variable will be accessible and consequently also determine its lifetime. The variable statement itself allocates memory for the values of the defined variables. The examples below provide more details about the variable assignment statement.

Initialization of Variables: Inferred and Declared Typing

The programmer can give variables a statically declared type, which may speed up performance and improve readability in the future. However, this feature is completely optional, as CatScript supports type inference and static typing for variable assignments.

```
var x: int = 5
var x_b: boolean = false
var y = 5
```

Note: If you create a variable and declare that its type be, say something like, (int), you will face an “incompatible type” error if you ever try to set that variable to anything other than an expression that evaluates to an (int). This also applies to almost every other CatScript type. However, regardless of its declared type, every variable can be set to equal null. Please see the example of an incompatible type error below.

```
var x: bool = true
x = 10
```

```
OUTPUT: "Error, incompatible types"
```

Initialization of List Variables: Inferred and Declared Typing

Lists can similarly be given a declared type by the programmer, which may speed up performance and improve readability in the future. This feature is also optional, as the type of the list and the component type can be determined through type inference as well.

```
var x: list<int> = [10,20,30]
var y = [10,20,30,40]
```

Note: If you create a list type variable and declare its component type, you will face an “incompatible type” error if you try to set it to a list with a different component type. This also

applies to almost every CatScript list type. However, regardless of its declared component type, every list variable can be set to equal null. Please see the example of an incompatible type error below.

```
var x: list<string> = [10,20,30]
print(x)
```

If the programmer does not declare a list's component type, the compiler will infer its component type by observing the types of the expressions in the list itself. Please see the "List Literal" section under Expressions for details about list component type inference. See an example below of CatScript list type inference.

```
var x = [10,"20",30]
var y:  list<object> = x
print(y)
```

```
OUTPUT: [10,"20",30]
```

Scope of Variables

The scope of the variables within a variable statement is initially globally scoped. If you make a variable assignment within a function call or a for loop, though, the scope of that variable will exist locally within that section of code only. Suppose the variable is reassigned or called later. In that case, the CatScript compiler will look up the variable name in a local symbol table before checking the global scope and use the local definition first.

```
var x = 1

function foo(){
      x = 10
      print(x)
}
foo()
print(x)
```

```
OUTPUT: 10, 1
```

Note: If you attempt to assign a variable with an identifier that already exists within the local scope, you will be faced with a "variable already defined" error.

```
var x: bool = true
var x: bool = false
```

```
OUTPUT: "Error, 'x' already defined"
```

Matthew Keck, Ivan Cline
CSCI-468
5/3/2024

Assignment Statement

CatScript assignment statements are used to reassign already declared variables. The CatScript assignment statement involves using some identifier, which acts as the variable's name, an (=), and some valid expression on the right-hand side of the statement. See the example below. The variable will only be redefined within the local scope in which it is called.

```
//Initial variable declaration, puts aside memory for the variable's name
// and value
var x: int = 5
print(x)
// since variable "x" already declared in this scope, we can reassign it to a
// value of the same type
x = 30
print(x)
```

```
OUTPUT: 5, 30
```

Note: As mentioned in the Variable Statement section, once a variable has been declared using a "var" statement, it can be reassigned so long as it matches its previously determined CatScript type. If you attempt to assign a variable with an identifier that already exists within the local scope, you will be faced with a "variable already defined" error.

If Statement

CatScript if statements are used by a programmer to implement control flow in their CatScript program. The if statement begins with the designated keyword (if) and immediately requires open parentheses. Within those parentheses will be some expression that will eventually evaluate to a CatScript-type (bool); this will be the 'test" of the if statement. Finally, after closing the test expression with a right parenthesis, you can add statements that will be evaluated if the test expression evaluates to true. The statements will be found in the curly braces after your if statement test expression. The else statement block will be evaluated instead if the if statement test expression is evaluated to false, though it is not required to have an else statement. See the example below.

```
function greaterThan0(x){
    if(x > 0){
        print("Greater than 0")
    }else{
        print("Less than or equal to 0")
}
print(greaterThan0(0))
print(greaterThan0(1))
```

```
OUTPUT: "Less than or equal to 0", "Greater than 0"
```

CatScript does not allow the use of “else if” statements; however, you can achieve a similar effect by nesting if statements underneath else statements.

```
function positiveOrNeg(x){
    if(x > 0){
        print("Positive")
    }else{
        if(x == 0){
            print("Zero")
        }else{
            print("Negative")
        }
}
print(positiveOrNeg(0))
print(positiveOrNeg(1))
print(positiveOrNeg(-1))
```

```
OUTPUT: "Zero", "Positive", “Negative”
```

Function Definition Statement

CatScript function definition statements, similar to variable assignment statements, are used to declare and describe the behavior of reusable functions and provide them a name. Function names share the same namespace as variables. Functions can have a declared return type, or CatScript can infer it. Functions in CatScript can also take an argument list (found within the parentheses); these argument identifiers will only exist within the function's scope and will be deleted after invocation. The identifiers in the argument list can be used for computation within the function body statement and can also be given a declared type, though it is unnecessary. Finally, functions can use the CatScript return statement to return some value to wherever the function was initially invoked. If the function definition statement does not have a return statement within its body, it will simply return CatScript type “void.”

```
//All valid function definitions
function foo(x,y){
    return x
}

function foo_int(x: int, y: int): int{
    return x
}
```

Matthew Keck, Ivan Cline
CSCI-468
5/3/2024

```
function foo_int(x: int, y: int): void{
   print(x) }
```

Return Statement

CatScript return statements can only be found within a function definition statement. Executing a return statement will cause the current invocation of the function whose definition it was a part of to cease execution and return a particular expression to wherever the function was initially called.

```
function foo(x: int): int{
    return x //return x to return statement in foo_2(x)
}

function foo_2(x: int): int{

    return foo(x) //return value returned from foo(x)
}
var t = 100
print(foo_2(t))
```

```
OUTPUT: 100
```

Function Call Statement

CatScript function calls can be evaluated as expressions and/or statements. This means that you can make function calls that return a value (in the case of it being treated as an expression) or simply produce side effects (in the case of it being treated as a statement). All CatScript function names share the same namespace as variables and are all declared in the global scope. CatsScript function calls can take an argument list to perform operations, with the argument list, function name, and function behavior all being defined within the function definition statement.

```
//Function definition statement
function add_oneHundred(x){
    return x + 100;
}
//Function call expression
var y = add_oneHundred(1)

print(y)
```

Matthew Keck, Ivan Cline
CSCI-468
5/3/2024

```
OUTPUT: 101
```

For Loop Statement

The CatScript for loop is the only iteration tool in the language. The start of a for loop is denoted by the keyword (for), a set of parentheses containing an identifier, the keyword (in) and an iterable object (CatScript list), and finally, a set of curly braces optionally containing executable statements. The identifier used within the set of parentheses will be the value that updates the current element of the iterable and will be used in the statement execution within the curly braces. The identifier is also declared locally within the for loop, meaning that once the for loop is finished, the identifier name can be declared elsewhere in the program.

```
// Iterable object
var iterable: list<int> =[1,2,3,4,5]
 for(i in iterable){
    print(i)  //for each integer in iterable, print the integer
}
//variable name “i” is now available to be declared because we exited the
// for loop
var i = true
print(i)
```

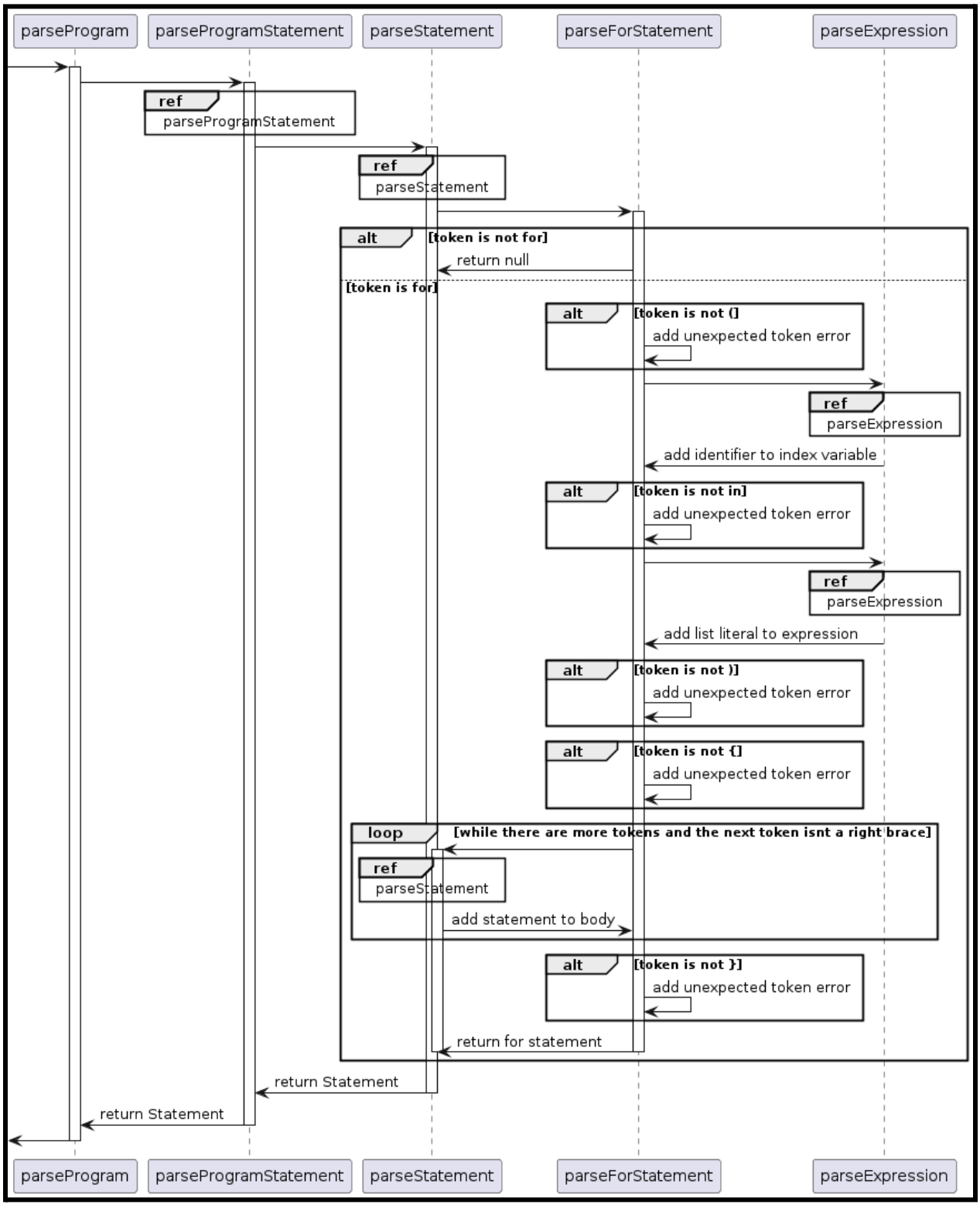
```
OUTPUT: 1, 2, 3, 4, 5, true
```

Sequence Diagram

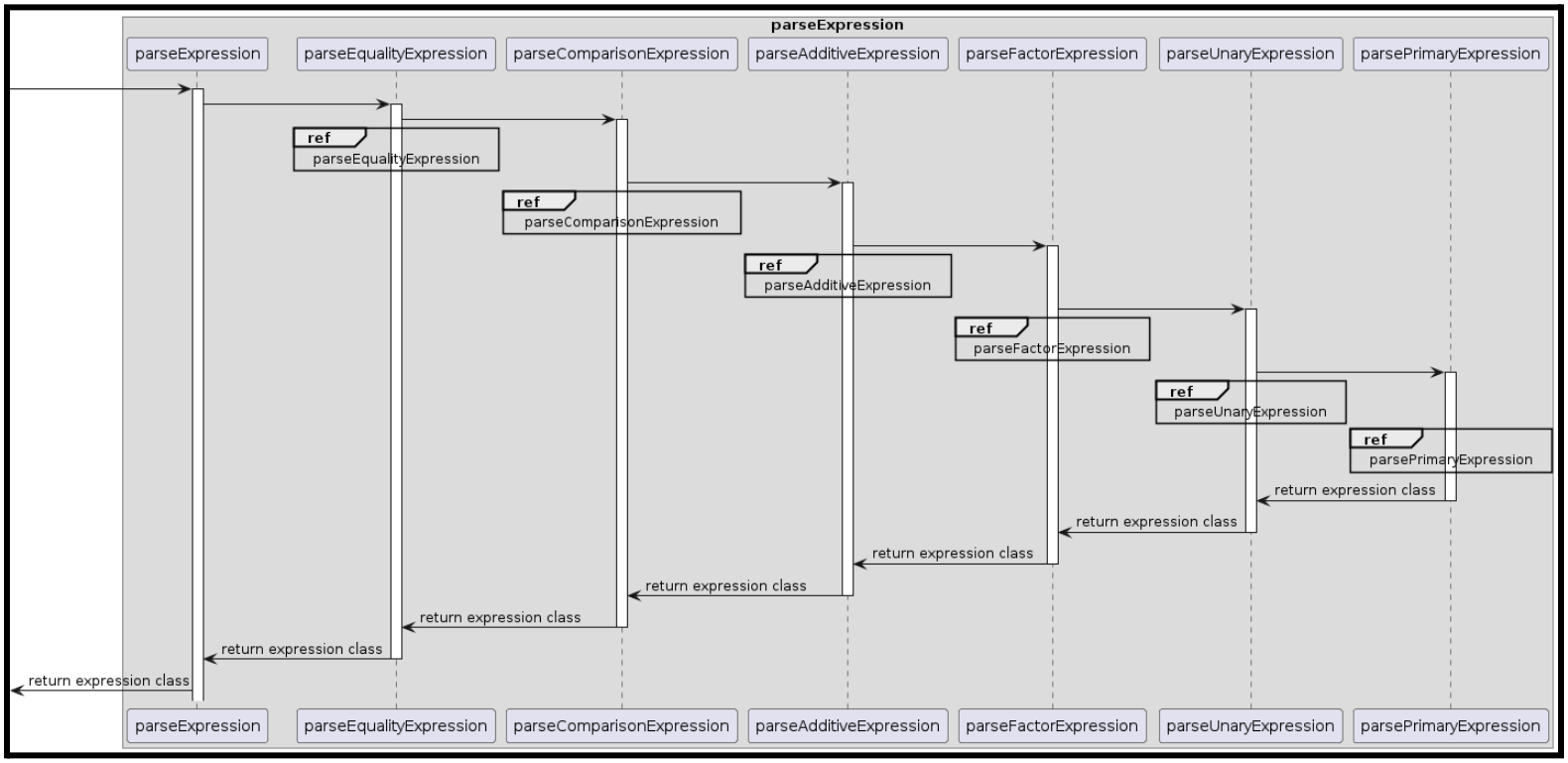
Below is a sequence diagram that graphically describes how the parser we created parses a for loop. As with parsing all Catscript programs, the first function called is parseProgram. The parseProgram function then calls parseProgramStatement, which the intern calls parseForStatement. Once the parseForStatement function has been called, the first step is to check if the current token is the keyword (for). If it is, we continue parsing. If not, we return null, and the parsing moves on and tries to match different statements. Next, we check if the next token is an opening parenthesis. If the next token is not an opening parenthesis, we add an error to the ForStatement class. The next step is to parse an Identifier expression we can use in the for loop's body to retrieve the list element for that current loop iteration. To do this, we go through a series of function calls that parse an expression, culminating in matching an identifier expression, which is then returned to the parseForStatement function. Next, we check if the next token is the (in) keyword; if not, we add an error to the for statement. Next, we go through the same process of parsing an expression. However, the expression that will be returned to the for statement is a literal list expression. The following two steps check if the subsequent tokens are closing parentheses and an opening brace. If they are missing, we add an error to the for statement. Next, we enter a loop that performs the parseStatement function for every iteration.

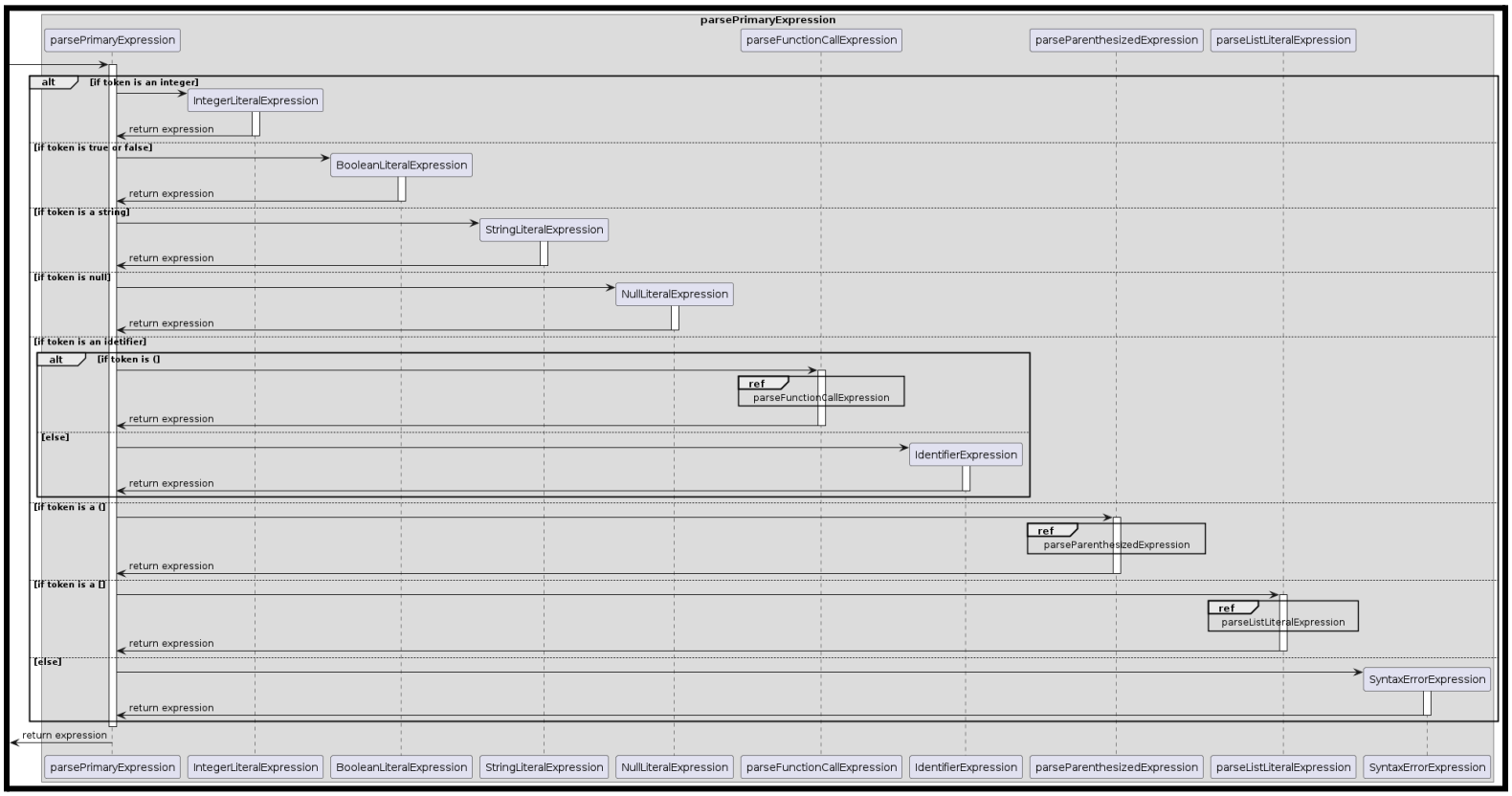
Matthew Keck, Ivan Cline
CSCI-468
5/3/2024
The parseStatement function then calls the other statement parsing functions until one doesn't return null. There are seven statement types that the parseStatement function can match: for loop statements, if statements, variable statements, assignment statements, function call statements, and return statements. The loop exits when the next token is a closing brace (}) or there are no more tokens. After leaving the loop, we check if the current token is the closing brace (}), and if it's not, we add a parse error to the for statement class. We then return the parsed for statement object to the parseProgram function.

Matthew Keck, Ivan Cline
CSCI-468
5/3/2024

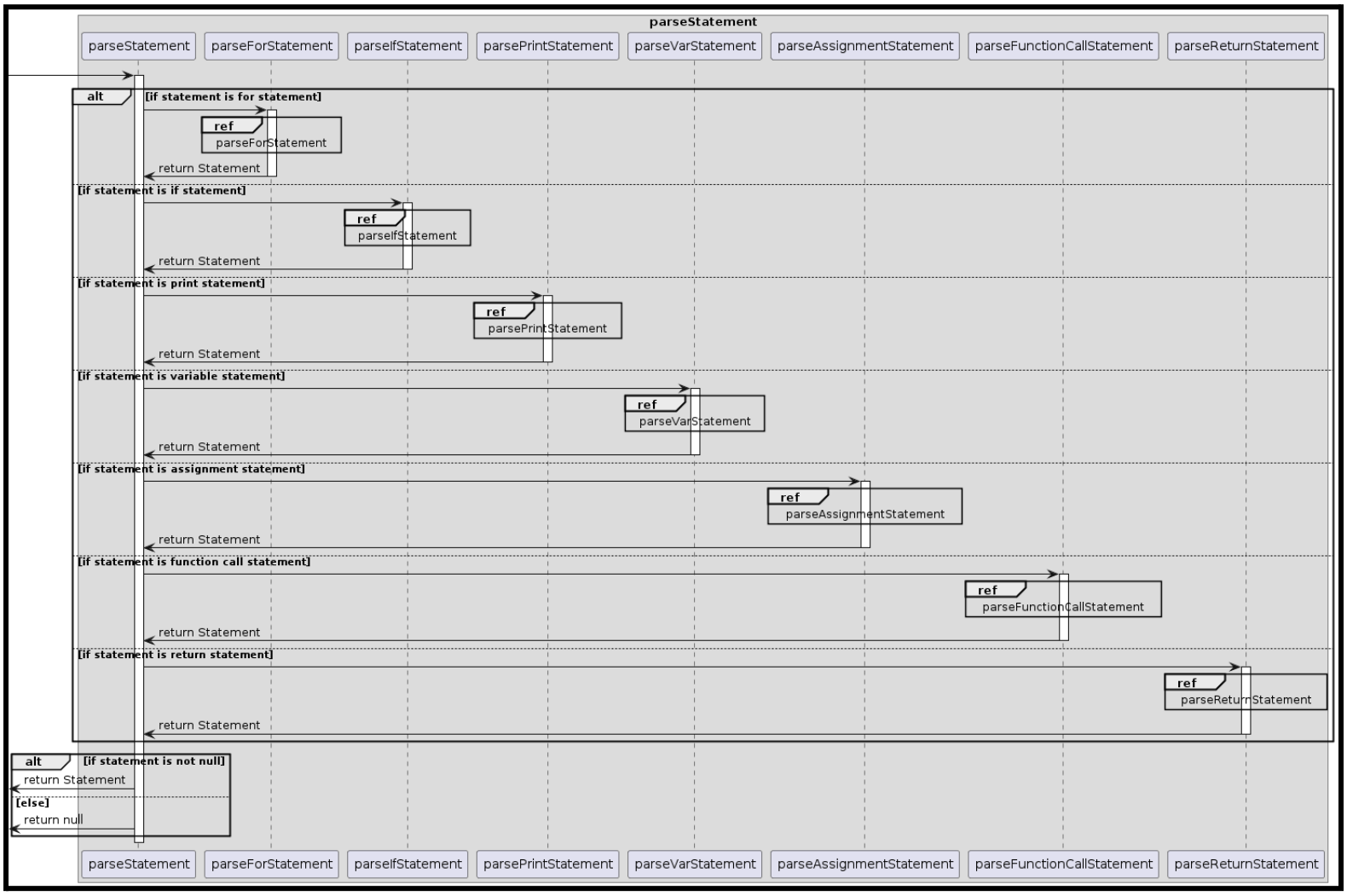


Matthew Keck, Ivan Cline
CSCI-468
5/3/2024





Matthew Keck, Ivan Cline
CSCI-468
5/3/2024



Design Trade-Offs

As mentioned in the introduction of this document, we decided to use the recursive decent algorithm to parse our programs and produce a parse tree. Instead of recursive decent, we could have used a parser generator to parse the programs. This section will discuss our reasoning for using recursive descent instead of a parser generator and walk through decisions made in constructing the recursive descent algorithm.

The main advantage of recursive descent over parser generators is control. Because recursive descent required creating the functions for each rule in the Catscript grammar, we had more control over how the programs were parsed. However, this strength is also one of the drawbacks of using recursive descent, which is that it requires writing a lot more code that gets into the nitty-gritty of splitting up text and creating a parse tree.

On the other hand, parser generators like YACC (Yet Another Compiler Compiler) or ANTLR (ANother Tool for Language Recognition) would have allowed us to abstract much of the

process of parsing by allowing us to input the Catscript grammar directly into the parser generator, which would produce the code for the parsing stage. While this would have allowed us to write much less code, it would have removed the control we had with the recursive decent algorithm. The loss of power would prevent us from modifying the parsing logic. Another drawback of parser generators is the generated code could be less readable and more challenging to understand and debug. Because we used recursive descent, our team had visceral knowledge of how the parser parses Catscript programs. Another advantage of using parser generators is they are highly optimized and quickly generate code that parses even the most extensive grammar quickly and accurately.

In summary, using the recursive decent algorithm made parsing Catscript programs more transparent. It gives us greater control and is well suited for parsing a small, straightforward grammar like Catscript. On the other hand, a parser generator would have been better suited for a more complex grammar that would take longer to program by hand. A parser generator would also guarantee the parsing phase's performance and correctness. However, because Catscript is a simple programming language, optimizing performance was not the most pressing concern.

Software development life cycle model

Test-first development was The software development approach we used to build the Catscript compiler. Test-first development is a software development approach requiring developers to create a test suit before writing the corresponding code. Our professor developed a test suit for the Catscript compiler, which included 148 tests for all four compilation phases: tokenizing, parsing, evaluation, and bytecode generation. However, as mentioned in the teamwork section of this document, my partner implemented and added three new tests to the provided test suit. Test first development was an excellent choice for developing Catscript because the method ensures the code meets the desired functionality. Having a test suit before creating code also helped us maintain the code functionality as the project evolved. Because the test suit was persistent, any changes could be run against the tests we had previously passed, which would tell us if our changes had created bugs in the compiler. One of the drawbacks of test-first development is that the code developed could be structured to fit the test requirements, which could cause us to neglect aspects not covered in the test suit. Test-first development was an excellent choice for this project and helped us create an efficient and correct Catscript compiler.