Section 2

May 2024

For this project, The work was split between 2 students, Student 1 (Myself), and Student 2 (Partner). Student 1 was responsible for the implementation of the code. their work can be found in the source.zip file that was provided. student 2 was responsible for the documentation, as well as developing tests for the project to find mistakes in student 1's code.
Here are 3 tests that were written by student 2:

```
    @Test
void PartnerTest1() {
    assertEquals("42\n", executeProgram("var x : int = 6\n" + "print(x * (x+1))"));

}

@Test
void PartnerTest2(){
    assertEquals("13\n", executeProgram("function foo() : int { return 3}\n" + "function boop()" +
            " : int { return 10 }\n"+ "print(foo() + boop())"));

}
@Test
void PartnerTest3(){
    assertEquals("1\n2\n3\n4\n5\n6\n7\n8\n9\n", executeProgram("for(x in [[1,2,3],[4,5,6],[7,8,9]]){" +
            "for(i in x){ print(i)}}"));
}
```

PartnerTest1 was to test the parser's ability to maintain the correct order of operations and to make sure that it was able to reference a variable multiple times in the same expression.
PartnerTest2 was designed to test expressions that were made up of function call statements. This was to ensure that the typing of function call statements was working correctly and allowed them to be used in this manner.
PartnerTest3 was designed to test multi-dimensional arrays and nested for loops. This test ended up being the most insightful among these 3 as Student 1 was made aware that variables in his 'for statements' were not being scoped correctly.
In terms of time, Student 1 contributed roughly 70% of the total time that was spent on this project and Student 2 contributed about 30% of the total time for this project.

Section 3: Design Pattern

May 2024

The design pattern that was used for this project was Memoization. The implementation of this pattern can be found in the CatScriptType.java file located in source.zip that was provided. specifically, I have implemented it on line 38 to memoize the creation of type list literals. The reason that I opted for this pattern instead of just coding directly is because not memoizing would be wasteful of both memory and computing resources. the other main reason is that the CatScriptType list call is often made with the same if not similar arguments, thus making this an even better option.

The pattern in use:

```
static HashMap<CatscriptType,ListType> cache = new HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
    return cache.computeIfAbsent(type, k -> new ListType(type));//TODO this computeIfAbsent statement
    memoizes the creation of type lists.

}
```

CatScript Documentation

May 2024

1 Introduction

CatScript is a simple programming language that does not support the use of classes or structs as you would find in more object-oriented programming languages. The CatScript parser is written in Java and runs on the Java Virtual Machine (JVM).

2 Features

CatScript has multiple features such as:

- Data Types
- Variables
- Expressions
- Control Structures
- Statements

2.1 Data Types

CatScript Uses a type_expression class that supports all of the class types found in CatScript. These types include: String, Integer, Boolean, Object, and List.

```
type_expression = ('string' | 'int' |  'bool' | 'object' | '<', 'type_expression','>')
```

2.2 Variables

CatScript uses static scoping while assigning values to user-generated variables. CatScript supports both implicit and explicit typing when declaring a variable. The syntax of a variable statement in CatScript is defined as:

```
variable_statement = 'var', IDENTIFIER, [':', type_expression, ] '=', expression;
```

Example variable statement:

```
var x = 10
var x : int = 10
```

Both of these statements will create a variable named x, of type 'int', with a value of 10

2.3 Expressions

Here is a list of the expressions that are available in CatScript:

Equality Expression

Equality expressions takes two values or expressions of the same data type and checks if they are either equal or not equal. It then returns a boolean value of either 0 (if false) or 1 (if true). The syntax for an equality expression in CatScript is defined as:

```
equality_expression = comparison_expression { ("!=" | "==") comparison_expression };
```

Example equality expression:

```
1 == 1
'true' != 'false'

1 != 1
'hello' == 'hi'
```

The first two expressions would return a boolean value of 1 (true)
The second two expressions would return a boolean value of 0 (false)

Comparison Expression

The Comparison Expression takes two values or expressions of integers(int) and checks them with one of 4 available comparison operators to return a boolean value of either 0(false) or 1 (true).
The available comparison operators are:

- $>$ greater than
- $\geq$ greater than or equal
- $<$ less than
- $\leq$ less than or equal

The syntax of a comparison expression is defined as:

```
comparison_expression = additive_expression { (">" | ">=" | "<" | "<=" ) additive_expression };
```

Example use of a comparison expressions:

```
 x > y
return 2 <= 3
```

The second statement would return a boolean value of 0.

Additive Expresion

Additive expressions take two values or expressions of types int or string. If both values are of type int, the additive expressions will return either the sum or difference of those two values. if one of the values is of type string, the additive expression will cast the other value to a string (if it is type int), and will concatenate the two strings and return the new value.
the syntax of an additive expression is defined as:

```
additive_expression = factor_expression { ("+" | "-" ) factor_expression };
```

Example use of an additive expression:

```
var x : int = 2
return x - 3

return 'Hello' + 'Goodbye'

return 'Item number ' + x
```

The first expression returns -1
The second expression will return 'HelloGoodbye'
The third expression will return 'Item number 2'

Factor Expression

Factor Expressions in CatScript take in 2 values or expressions of type int, and return either the product or the quotient of those two values.
the syntax of a factor statement is defined as:

```
factor_expression = unary_expression { ("/" | "*" ) unary_expression };
```

Example use of a factor statement:

```
var x = 2 * 3
var y = 6/2
```

The value of x would be set to 6
The value of y would be set to 3

Unary Expression

Unary expressions in CatScript either return the inverse of a given value or expression of type bool, or they return a given value or expression of type int multiplied by a negative one.
The syntax of a unary expression is defined as:

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

Example use of a unary expression:

```
x = -1

return not true
```

The value of x will be set to -1
The return statement would return false

Primary Expression

Primary expressions in CatScript represent identifier values, data values, boolean values, the null value, and parenthesized expressions. The syntax of a primary expression is defined as:

```
primary_expression = IDENTIFIER | STRING | INTEGER | "true" | "false" | "null"|
        list_literal | function_call | "(", expression, ")"
```

2.4 Control Structures

There are two control structures available for use in CatScript, the If statement, and the For statement.

If Statement

The if statement allows for the conditional execution of a code block based on a condition given in the if statement. The If statement takes takes an expression of type bool and will execute a code block if that expression returns a 'true' value. Another if statement is able to be appended to the back of an if statement allowing for if, else/if statements to be constructed.
The syntax of an if statement is defined as:

```
if_statement = 'if', '(', expression, ')', '{',
                { statement },
            '}' [ 'else', ( if_statement | '{', { statement }, '}' ) ];
```

Example use of an if statement:

```
if (x > 3){
    print('Its's bigger!')
}else{
    print('Its's smaller!!')
}
```

In this example, if x was assigned a value greater than 3, the first code block would be executed, and 'It's bigger!!' would be printed. If x was assigned a value less than or equal to 3, the second code block would execute instead and would print 'It's smaller!!'.

For Statement

The for statement in CatScript takes in an expression of type list and executes a block of code for each element in that list while assigning a variable to that element.
The syntax for a for statement is defined as:

```
for_statement = 'for', '(', IDENTIFIER, 'in', expression ')',
             '{', { statement }, '}';
```

Example use of a for statement:

```
var x = [1,2,3,4]
for (i in x){
    print(i)
    print(\n)
}
```

This for loop will print:
1
2
3
4

2.5 Statements

Print Statement

The print statement in CatScript takes an expression and outputs the value of that expression to the standard output stream
The syntax for the print statement is defined as:

```
print_statement = 'print', '(', expression, ')'
```

Example print statement:

```
print('Hello')
```

This will output 'Hello'

Variable Statement

The variable statement in CatScript is used to create a variable and assign a value to it. An explicit type can be assigned at initialization as well.// The syntax for the variable statement is defined as:

```
variable_statement = 'var', IDENTIFIER,
 [':', type_expression, ] '=', expression;
```

Example use of a variable statement:

```
var x = 10
var y : int = 10
var z : list<string> = ['good', 'bad', 'ok']
var a = [1,2,'yo']
```

Both x and y will be assigned a value of 10 and be of type int// z will be assigned a list of strings Because a has multiple data types that make up the list, it will implicitly be typed as a list of objects.

Function Call Statement

The function call statement invokes a function, passing arguments to it if required; it executes the code within the function block and returns control to the caller once the function completes its execution.
the syntax for the function call statement is defined as:

```
function_call_statement = function_call;
```

Example function call statement:

```
foo()
```

This invokes the function foo(), which has no arguments.

Assignment Statement

The assignment statement assigns a value or expression to an already defined identifier of the same type as the value being assigned.
The syntax of an assignment statement is defined as:

```
assignment_statement = IDENTIFIER, '=', expression;
```

Example of an assignment statement:

```
var x = 10
x =2
```

In this example x = 2 is an assignment statement that assigns the int value of 2 to x.

Function Definition Statement

The function definition creates a user-defined function, encapsulating a block of code with a name and parameters for reuse.
The syntax for a function definition statement is defined as:

```
function_declaration = 'function', IDENTIFIER, '(', parameter_list, ')' +
                    [ ':' + type_expression ], '{',  { function_body_statement },  '}';
```

Example function definition statement:

```
function foo (x : int, y : string){
    var z = y + x
    print(z)
}
```

This function takes in an int(x) and a string(y). It then concatenates them and prints the result.

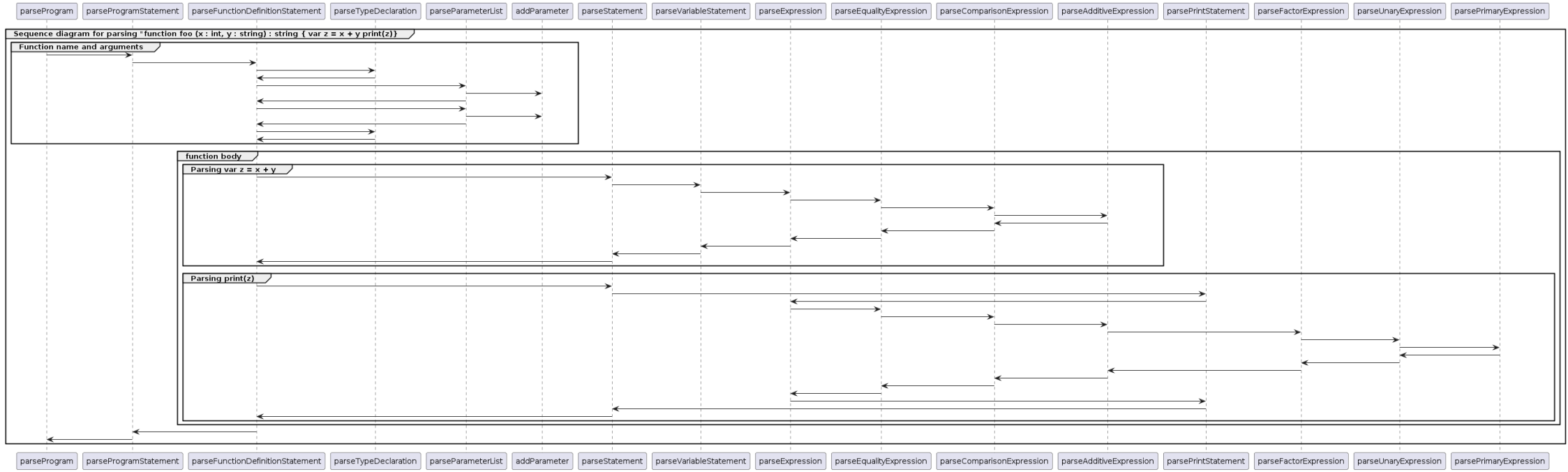
Return Statement

The return statement is what ends a function and program, returning what the output of a function would be just ending the program.
The syntax for the return statement is defined as:

```
return_statement = 'return' [, expression];
```

Example return statement:

```
function foo (x : int, y : string) : string{
    var z = y + x
    return(z)
}
```


parseProgram
parseProgramStatement
parseFunctionDefinitionStatement
parseTypeDeclaration
parseParameterList
addParameter
parseStatement
parseVariableStatement
parseExpression
parseEqualityExpression
parseComparisonExpression
parseAdditiveExpression
parsePrintStatement
parseFactorExpression
parseUnaryExpression
parsePrimaryExpression
Sequence diagram for parsing "function foo (x : int, y : string) : string { var z = x + y print(z)}
Function name and arguments
function body
Parsing var z = x + y
Parsing print(z)

Section 6: Design Trade-Offs

May 2024

For this project, we had the choice of creating a parser with two different methods. method one was to build the parser by using a parser generator such as Antler. The other method was to create a parser by using a recursive decent algorithm. for my project, we opted to use a recursive descent parser. The biggest reason why I believed that a recursive descent algorithm was the correct choice was its ease in debugging. In this class, we used Test Driven Development (TDD) to develop the parser. this meant that a lot of time was spent creating tests and debugging code. The process of debugging code is much easier when you are the one writing the code. another issue with debugging and testing in regards to a parser generator is the clarity of the code produced. Looking at the code that is generally produced by a generator, it was clear that a large amount of time would need to be spent in order to understand what the code was doing. many generators employ advanced techniques that are specific to the language of your parser that need to be understood if one wants to effectively work with the code that is generated.

Another major reason why a recursive descent algorithm was chosen is how it aligns with the grammar of a language. when constructing the parser, it became clear how recursive descent intuitively aligned with the structure of our grammar. We were able to construct the parse_element and parse_Statement trees in a way that very closely mirrored the grammar that was laid out in our EBNF document for our grammar. This made conceptualizing and implementing code much faster and easier than if we had opted to use a generator.

Section 7: Software Development Life Cycle Model

May 2024

The model that we used to develop our capstone project was a test-driven development(TDD) model. Test Driven Development (TDD) is a software development approach where tests are written before the actual code is implemented. It follows a cycle of writing tests, implementing code, and then refactoring.

Writing Test Cases

to being using this model, Test cases are written to determine the behaviors that we want to see from the program.

Run Tests

The next step in the process is to run the tests that we made. The first time we took this step at the beginning of development all of the tests failed, as we hadn't implemented code yet, but this does confirm that the tests are working properly and accurately reflect the desired behavior.

Implement Code

The next step in the process was to implement code to begin passing tests.

Running Tests

After the code has been implemented, tests need to be run again to see if the solutions that we had implemented were working. when we did this step for our project, we implemented the code and ran one test at a time. The main reason why we couldn't attempt multiple tests at a time was due to dependency issues. For example, if we wanted to test to make sure that parsing a function expression was working properly, we had to first ensure that our parser was able to successfully parse anything that could be placed into a function. This meant that throughout the development of our parser, we needed to establish a hierarchy of dependency for our tests and work our way up from the bottom. The benefit of this was that as we worked our way up to more complex processes in development, it always came after we were able to fully understand all the other parts of the parser that came before.

Refactor Code

The next step in the TTD model is to refactor code to make it more readable, manageable, and efficient. A good example of this step for us was the parseTypeDeclaration function found in the CatScriptParser file. we had noticed that we were going to need to parse Type information for both function and variable definitions, so we decide to make this its own method in order to improve readability. This kind of refactoring made a big difference over the course of development in making the whole project far more manageable.

We continually repeated this process as new requirements for our parser were introduced over the semester. In retrospect, we believe that this was the best way that we could approach this project. It doesn't seem to us that there were any real downsides to using the model. The model paired nicely with the escalation from tokenizing and parsing, to execution and byte code generation.