Lukas Bernard

Nathan Parnell (Main Author)

CSCI 468 – Compilers
Capstone Portfolio Document

Section 1: Program

See included source.zip directory.

Section 2: Teamwork

Team Member 1: 95% -

Team Member 1 did the implementation and coding for the project, starting with the tokenizer, then the parser, and finally compiling into bytecode.

The tokenizer was written early on, and chunked written statements into lexical objects, and sorted them into tokens that were usable by the compiler, such as identifying the “type” of each token and the operations attached to that token type. Such as the “if” mapping to a token type that tells the compiler that an if statement is coming up, or identifier tokens mapping to variables or function calls.

Using recursive descent, the parser would identify the type of statement or expression, and generate a parse tree. This had to be done in a specific order because of operator precedence, This would generate certain expressions or statements using the left and right side of each, and the matching token type to determine what kind of expression / statement was appropriate.

Catscript is then compiled using JVM bytecode. This was hand-written to provide specific instructions to the virtual machine, which allowed the language to run and produce output. Without this step, Catscript would simply be generating parse trees.

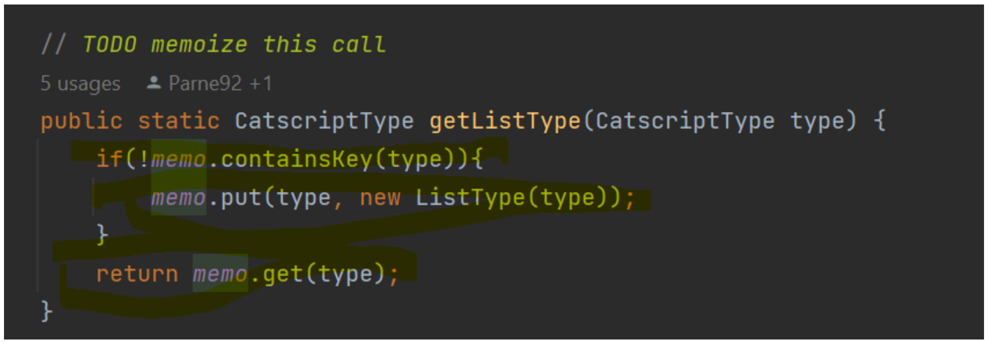
Team Member 2: 5%

Partner two wrote the documentation and provided testing for Catscript. The documentation is provided in the technical documentation section below. The provided tests checked if:

1. Functions could take in variables from for loops as valid arguments
2. If statements worked as expected within the body of for statements
3. Variable scoping worked as expected when referencing them outside of the declared scope (checking if error occurred)

Section 3: Design pattern

Memoization was the chosen design pattern. It is in CatscriptType.java, at line 37. This design pattern was chosen to avoid returning a bunch of list type objects,



```
// TODO memoize this call
5 usages   Parne92 +1
public static CatscriptType getListType(CatscriptType type) {
    if(!memo.containsKey(type)){
        memo.put(type, new ListType(type));
    }
    return memo.get(type);
}
```

The way this design pattern object works is that memo is a hashmap, and each time getListType is called, it checks to see if this listType object already exists, and returns it if it does, and adds it to it the hashmap if it does not yet exist. This prevents bogging down the compiler with a bunch of duplicate objects.

Section 4: Technical writing. Include the technical document that accompanied your capstone project.

CatScript Guide

Introduction

Catscript is a simple scripting language. Here is an example:

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

Output:
foo

Additive Expression

Description

Allows for the addition and subtraction of integers, as well as the concatenation of strings.

Example

1 + 2;		2-1;		“String” + “House”;

Parameters

Operator (Token), leftHandSide (Expression), rightHandSide (Expression)

Returns the product of addition, subtraction or concatenation.

Boolean Literal

Description

Allows for the compilation of a boolean value.

Example

test = true;		if(test) {*execute when test is true*}

Parameters

Boolean value

Returns a boolean literal value (‘true’ or ‘false’)

Comparison Expression

Description

Allows for the comparison between two integers.

Example

2 <= 2;   4 < 5   5 > 2

Parameters

Operator (Token), leftHandSide (Expression), rightHandSide (Expression)

Returns True if comparison is valid, False if comparison is not valid.

Equality Expression

Description

Allows for the evaluation of if two expressions are equal/not equal

Example

1 == 1

Parameters

Operator (Token), leftHandSide (Expression), rightHandSide (Expression)

Returns True if “==“ operand and objects match, false if they don’t match.

Factor Expression

Description

Allows for the multiplication and division of expressions

Example

5 * 5 6 / 3 2 * 2

Parameters

Operator (Token), leftHandSide (Expression), rightHandSide (Expression)

Returns the product of leftHandSide (* or /) rightHandSide

Function Call Expression

Description

Implements the ability to call functions

Example

function test(x) {print(x);}

Parameters

functionName (String), arguments (List<Expression>)

Returns N/A

Identifier Expression

Description

Handles the name of variables

Example

var x = 2

Parameters

value (String)

Returns N/A

Integer Literal Expression

Description

Implementation for the creation of integer values

Example

var x = 2 Parameters

value (String) Returns

N/A

List Literal Expression

Description

Implementation for list types in CatScript

Example

[1, 2, 3, 4]

Parameters

values (List<Expressions>)

Returns N/A

Null Literal Expression

Description

Implementation for the handling of null values

Example

var test = null

Parameters

N/A

Returns N/A

Parenthesized Expression

Description

Allows for parenthesis to modify order of operations in expressions.

Example

(1 + 2) * 2

Parameters

expression

Returns the value of the internal expression

String Literal Expression

Description

Implementation for the handling of Strings

Example

var test = “testString”

Parameters value (String)

Returns stringValue (String)

Syntax Error Expression

Description

Implementation of Syntax Error

Example

throw new IllegalStateError(“Bad token : “ + getStart());

Parameters

consumeToken (Token)

Returns N/A

Type Literal Expression

Description

Allows for the declaration and retrieval of Catscript Types

Example

Boolean, Integer, Object

Parameters type

(CatscriptType)

Returns N/A

Unary Expression

Description

Implementation for negative and negation on values

Example

```
!false;                -2;
```

Parameters

operator (Token), rightHandSide (Expression)

Returns negative (int) or negation (boolean) of value

For Statement

Description

Implementation for looping through code iteratively

Example

```
for (index in [“a”, “b”, “c”]) {print(index)}
```

Parameters

expression (Expression), variableName (String), body (List<Statements>)

Returns N/A

Function Call Statement

Description

Implementation for calling functions
Example
test(a)
Parameters
parseExpression (FunctionCallExpression)
Returns N/A

Function Definition Statement

Description
Implementation for defining functions
Example
Function test() {}
Parameters
N/A
Returns N/A

If Statement

Description
Allows for the execution of code under certain conditions.
Example

if(x==true) {print(“Hello”);}

Parameters

N/A

Returns N/A Print

Statement

Description

Allows for the output of an expression

Example

print(“Test”);

Parameters

N/A

Returns N/A

Return Statement

Description

Allows for the returning of values in functions

Example

return x;

Parameters

parseExpression (Expression)

Returns expression (Expression)

Syntax Error Statement

Description

Allows for the creation of Syntax Errors

Example

Return new SyntaxErrorStatement(tokens.consumeToken());

Parameters

tokens.consumeToken()

Returns Statement

Variable Statement

Description

Allows for the creation of variables, and the memory slot where they are stored

Example

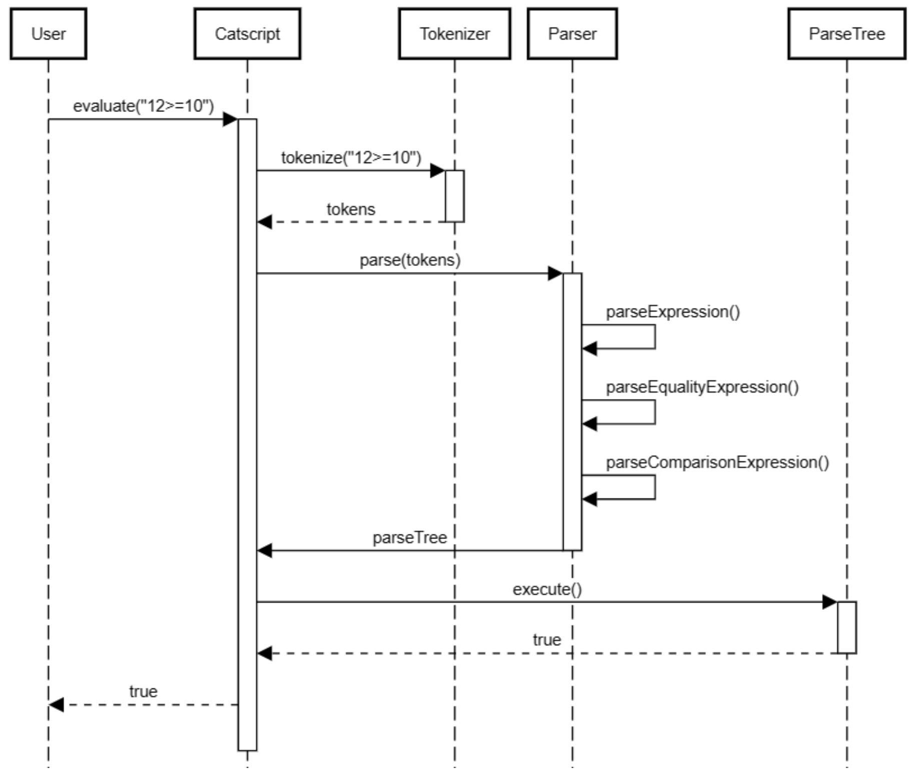
var test = 17

Parameters

variableName (String), type(Catscript Type), expression(Expression)

Returns

Section 5: UML.



This sequence diagram shows the process that happens when you parse a comparison expression. When the user sends the instruction to evaluate 12 >= 10, the Catscript program will send this to the tokenizer, and receive back the individual tokens. The Parser then uses the recursive descent to parse this, going through parseExpression, parseEqualityExpression, and then landing on parseComparisonExpression, where the tokens will match, and it will generate a parseTree and return it to the Catscript program. From here, the execute command will be run, which will find that this specific comparison expression is true, and that true value will be passed all the way back to the user.

Section 6: Design trade-offs

2-3 Paragraphs, code generator vs recursive descent

The design decision we made to go for recursive descent instead of code generation, this is because recursive descent was easy to do by hand, whilst code generation is hard to comprehend, and muddies each step of the process.

Recursive descent also provides more control over how each step of process is working, which in turn provides more knowledge on each step of the process. If code generation was used, we would have to deal with learning a new step of the process of reading through the generated code.

Section 7: Software development life cycle model

Using Test Driven Development (TDD) for the project was helpful because it allowed us to verify each step was working properly as we were writing, without having to just write more parsing statements under the assumption that we have written correct code for prior parsing, for example.

This model also allowed us to check the tests to get a general example the direction to start with. If a test failed, it also provided useful errors to debug with, where it started to break down was with bytecode tests, by the nature of writing bytecode, the tests were rather hard to debug because I hadn't worked with this area of the JVM before.