

Kelby Abel

CSCI 468 - Capstone Portfolio

Section 1 – Program

See the source.zip in this directory.

Section 2 – Teamwork

My primary work on the project was to write out code for a compiler with tokenizing, expression parsing, statement parsing, evaluation and validation, execution, and compilation. Also wrote out documentation for the CatScript programming language for my team member and wrote three unit tests for their compiler.

The tokenizer included taking in input code and separating the code into tokens that were added to a token list. A Token is an object that keeps track of start point, end point, line number, line offset, string value, type, etc.. Characters such as +, -, (, “, etc. are added as tokens to a token list. Doing so creates a way to parse the input code in the parsing section.

In this example, tokens are being checked for left brace and right brace, and adds the correct token type to the token list.

```
else if(matchAndConsume('{')) {  
    tokenList.addToken(LEFT_BRACE, "{", start, postion, line, lineOffset);  
} else if(matchAndConsume('}')) {  
    tokenList.addToken(RIGHT_BRACE, "}", start, postion, line, lineOffset);  
}
```

Expression parsing uses the token list to parse multiple expression types. Tokens are checked for matching characters or words which determine the correct expression type to parse.

In this example, the token list is being checked for MINUS or NOT tokens to parse unary expressions.

```
private Expression parseUnaryExpression() {  
    if (tokens.match(MINUS, NOT)) {  
        Token token = tokens.consumeToken();
```

```

    Expression rhs = parseUnaryExpression();
    UnaryExpression unaryExpression = new UnaryExpression(token, rhs);
    unaryExpression.setStart(token);
    unaryExpression.setEnd(rhs.getEnd());
    return unaryExpression;
} else {
    return parsePrimaryExpression();
}
}

```

Statement parsing is very similar to expression parsing and uses the token list to parse multiple statements. Tokens are checked for matching keywords to match to the correct statement type.

```

private Statement parsePrintStatement() {
    if (tokens.match(PRINT)) {
        PrintStatement printStatement = new PrintStatement();
        printStatement.setStart(tokens.consumeToken());
        require(LEFT_PAREN, printStatement);
        printStatement.setExpression(parseExpression());
        printStatement.setEnd(require(RIGHT_PAREN, printStatement));
        return printStatement;
    } else {
        return null;
    }
}

```

In this example, the keyword PRINT is matched from the token list to parse print statements. Other required characters are also checked to prevent syntax errors.

Evaluation properly returns the expected result of an expression.

```

public Object evaluate(CatscriptRuntime runtime) {
    Integer lhsValue = (Integer) leftHandSide.evaluate(runtime);
    Integer rhsValue = (Integer) rightHandSide.evaluate(runtime);

    if (this.isGreater()) {
        return lhsValue > rhsValue;
    } else if (this.isGreaterThanOrEqual()) {
        return lhsValue >= rhsValue;
    } else if (this.isLessThanOrEqual()) {

```

```

        return lhsValue <= rhsValue;
    } else if(this.isLessThan()) {
        return lhsValue < rhsValue;
    }
    return null;
}

```

In this example, the left hand side and right hand side of a comparison expression is evaluated. Checks are made for the correct type of comparison. The correct value is returned for further implementation.

Validation properly sets up statements to have the correct relations.

```

public void validate(SymbolTable symbolTable) {
    expression.validate(symbolTable);
    CatscriptType symbolType = symbolTable.getSymbolType(getVariableName());
    if (symbolType == null) {
        addError(ErrorType.UNKNOWN_NAME);
    } else {
        if(!symbolType.isAssignableFrom(expression.getType())) {
            addError(ErrorType.INCOMPATIBLE_TYPES);
        }
    }
}

```

In this example, the assignment statement is validated to have the correct expression type to match the variable type, if any.

Execution uses the validated statement to generate the correct output for the statement.

```

public void execute(CatscriptRuntime runtime) {
    Object conditionalResult = expression.evaluate(runtime);
    if(Boolean.TRUE.equals(conditionalResult)) {
        for (Statement trueStatement : trueStatements) {
            trueStatement.execute(runtime);
        }
    } else {
        for (Statement elseStatement : elseStatements) {
            elseStatement.execute(runtime);
        }
    }
}

```

In this example, the true and else statements of an if statement are executed to generate the correct output.

Compilation converts the tokenized, parsed, evaluated, validated, and executed code into Java bytecode.

```
public void compile(ByteCodeGenerator code) {
    code.addVarInstruction(OpCodes.ALOAD, 0);
    expression.compile(code);
    box(code, expression.getType());
    code.addMethodInstruction(OpCodes.INVOKEVIRTUAL,
code.getProgramInternalName(),
    "print", "(Ljava/lang/Object;)V");
}
```

In this example, print statements are compiled into the correct bytecode.

My sole team member, Jacob Clostio, wrote three unit tests for my compiler as well as the documentation included in Section 4. The nature of the project means the percentage of work done by each team member was about 90% to 10%. Team member 1, myself, completed 90% of the project, and 10% was completed by team member 2.

Team member 2 created the documentation in Section 4 which does a great job at describing the general grammar definitions of CatScript as well as an overview of what CatScript is. Team member 2 also created three unit tests to test the functionality of the CatScript compiler. The three tests are:

```
@Test // Combines a for loop and an if statement.
void ifStatementWorksInsideForLoop() {
    assertEquals( expected: "3\n4\n", executeProgram( src: "for( x in [1, 2, 3, 4] ) {\n" +
        "    if(x > 2) {\n" +
        "        print(x)\n" +
        "    }\n" +
        "}\n"));
}

@Test // Ensures that after a return statement is called, nothing else in the function gets called.
void linesAfterReturnFunctionAreNotCalled() {
    assertEquals( expected: "100\n", executeProgram( src: "function foo() : int {\n" +
        "    var x = 100\n" +
        "    return x\n" +
        "    var y = 99\n" +
        "    print(y)\n" +
        "    return y\n" +
        "}\n" +
        "print(foo())\n"));
}
```

```

@Test // Combines function call and if statement (for loop) and else statement (print statement)
void forLoopAndConditionalStatementInsideFunction() {
    assertEquals( expected: "1\n2\n3\n4\n5\n6\n7\n8\n9\n10\n", executeProgram(
        src: "function foo(x : int) {\n" +
            "if(x == 10) {\n" +
            "    for( y in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] ) {\n" +
            "        print(y)" +
            "    }\n" +
            "}" +
            "else { print(\"No Luck\") }" +
            "}\n" +
            "foo(10)"
    ));
    assertEquals( expected: "No Luck\n", executeProgram(
        src: "function foo(x : int) {\n" +
            "if(x == 10) {\n" +
            "    for( y in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] ) {\n" +
            "        print(y)\n" +
            "    }\n" +
            "}" +
            "else { print(\"No Luck\") }" +
            "}\n" +
            "foo(7)"
    ));
}

```

These tests are high level and help ensure that the compiler is compiling properly though various applications. The first test confirms that if statements work inside for loops, this code is used frequently in almost all user scripts. The second test confirms that code execution stops after a return statement, without this working, return statements would be useless. The last test confirms that the correct output is used with if statements and for loops inside functions. The output needs to be correct or functions and if statements are broken.

Section 3 - Design Pattern

Memoization is the primary design pattern in CatScript. The general idea is if something has already been created, it is stored, then if a method calls for it, the reference is found in storage rather than creating a new one. Memoization in CatScript is defined with this code:

```

private static final Map<CatscriptType, CatscriptType> LIST_TYPES = new HashMap<>();
3 usages
public static CatscriptType getListType(CatscriptType type) {
    CatscriptType listType = LIST_TYPES.get(type);
    if(listType == null) {
        listType = new ListType(type);
        LIST_TYPES.put(type, listType);
    }
    return listType;
}

```

When the `getListType` function is called with a `CatscriptType INT` for the first time, it is stored inside of a `HashMap`. The second time the `getListType` function is called with the same `INT`, the `HashMap` is checked for it, and the type is returned. This prevents reinitialization of existing `CatscriptType` types.

Section 4 - Technical Writing

Catscript Documentation

Jacob Clostio

Introduction

Catscript is a statically typed simple programming language that allows for a small number of types that include:

- `int` - a 32 bit integer
- `string` - a java-style string
- `bool` - a boolean value
- `list` - a list of value with the type '`x`'
- `null` - the null type
- `object` - any type of value.

Catscript was created by defining a relatively simple set of grammar rules. With this grammar, a compiler could then be built from the ground up to tokenize, parse, and eventually return bytecode that follows the grammatical rules of the language. Although simple, the language allows for reduced complexity, and improved readability, making it a good choice for beginners to learn simple programming features.

Features

```
catscript_program = { program_statement };
```

Catscript's grammar begins with the `catscript_program`. This grammar rule defines a Catscript program as zero or more program statements. This is a common way to define the syntax of a programming language and is used to parse the input code and generate an abstract syntax tree that represents the structure of the program.

```
program_statement = statement |  
                  function_declaration;
```

The second grammar rule is defined above, This rule defines a program statement in Catscript as either a general statement or a function declaration. Again, this rule is used by the parser to build an abstract syntax tree that represents the program structure, and is the higher level syntax of the language.

```
statement = for_statement |  
            if_statement |  
            print_statement |
```

```
variable_statement |  
assignment_statement |  
function_call_statement;
```

The statements are where we see the main features of the programming language, and can take the form of any of the alterations listed above.

Statements

```
for_statement = 'for', '(', IDENTIFIER, 'in', expression ')',  
               '{', { statement }, '}'  
if_statement = 'if', '(', expression, ')', '{',  
               { statement },  
               '}' [ 'else', ( if_statement | '{', { statement }, '}'  
) ];  
print_statement = 'print', '(', expression, ')'  
variable_statement = 'var', IDENTIFIER,  
                    [ ':', type_expression, ] '=', expression;  
assignment_statement = IDENTIFIER, '=', expression;  
function_call_statement = function_call;  
function_call = IDENTIFIER, '(', argument_list, ')'  
argument_list = [ expression, { ',', expression } ]  
type_expression = 'int' | 'string' | 'bool' | 'object' | 'list' [,  
                    '<', type_expression, '>']
```

These are the grammar rules for the statements in Catscript. Beginning with the `for_statement`, which would look like:

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

This is a standard `for` loop that requires an identifier, an expression, and a statement to run correctly. Continuing on with the `if` statement:

```
if (x == 1) { print(1) }  
else if (x == 2) { print(2) }  
else { print(3) }
```

```
print(x)
```

Where an expression is needed after the 'print' keyword. The variable_statement could look either like:

```
var x : int = 43
```

Or

```
var y = "foo"
```

Where the needed components are an identifier, an optional type expression (defined above), and finally an expression. Furthermore, Catscript allows for a variable to be of any of the built-in data types without needing to specify the type in the statement when using the 'var' keyword (although a type can be specified if wanted). The assignment_statement would take the form:

```
x = 10  
y = "bar"
```

Given an identifier, and an expression, one could change the assignment of a variable with this simple syntax. Finally the function_call_statement requires a function_call,

```
myFunction()
```

which consists of an identifier and an optional argument list (which is simply expressions separated by commas).

Functions

```
function_declaration = 'function', IDENTIFIER, '(', parameter_list,  
' )' +  
                      [ ':' + type_expression ], '{', {  
function_body_statement }, '}' ;  
  
function_body_statement = statement |  
                        return_statement ;  
  
parameter_list = [ parameter, { ',' parameter } ] ;  
  
parameter = IDENTIFIER [ , ':' , type_expression ] ;  
  
return_statement = 'return' [ , expression ] ;
```

These grammar rules define how to declare and define functions in Catscript.

function_declaration begins with the keyword function followed by the function name, a list of parameters enclosed in parentheses, an optional return type, and the function body enclosed in braces. The parameter_list defines the function parameters, which may include a type expression. The function_body_statement can be any statement or a return_statement that indicates the value to return from the function. Return_statement specifies the keyword 'return' followed by an optional expression that represents the value to be returned from the function. Here's an example

of a simple function in Catscript that takes two parameters and returns their sum:

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

The `function_declaration` rule is used to declare a new function named `add`, which takes two integer parameters `x` and `y`, and returns an integer value. The `function_body_statement` consists of a single `return_statement` that adds the two parameters and returns the result.

Expressions

```
expression = equality_expression;  
  
equality_expression = comparison_expression { ("!=" | "==")  
comparison_expression };  
  
comparison_expression = additive_expression { (">" | ">=" | "<" |  
"<=" ) additive_expression };  
  
additive_expression = factor_expression { ("+" | "-" )  
factor_expression };  
  
factor_expression = unary_expression { ("/" | "*" ) unary_expression  
};  
  
unary_expression = ( "not" | "-" ) unary_expression |  
primary_expression;  
  
primary_expression = IDENTIFIER | STRING | INTEGER | "true" | "false"  
| "null" |  
list_literal | function_call | "(" , expression ,  
")"  
  
list_literal = '[' , expression , { ',' , expression } '];
```

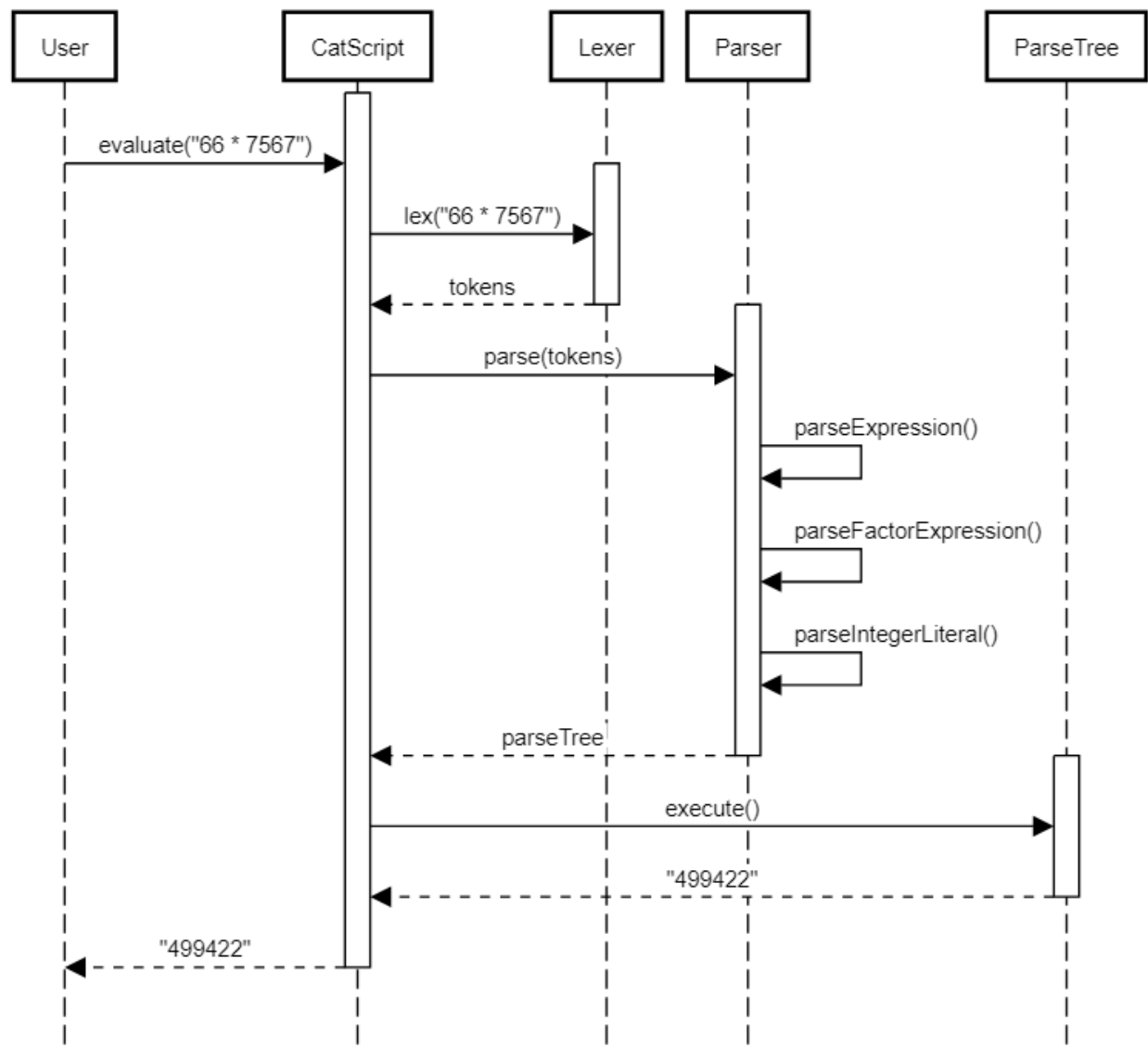
This set of grammar rules defines the basic structure of expressions in Catscript. An expression is built up from smaller expressions using a hierarchy of operations, starting with `primary_expression` at the lowest level and working up to `equality_expression` at the highest level. `Primary_expression` represents the simplest expressions, such as literal values (ints, strings, true, false, and null), identifiers (variable names) and `list_literals`. `function_call` represents a function call expression, which is used to invoke a function and pass arguments to it. `Unary_expression` represents unary operators, such as negation and logical negation. `factor_expression` represents multiplicative operators (`*` and `/`), while `additive_expression`

represents additive operators. Finally, `comparison_expression` represents relational operators (`>`, `<`, `>=`, and `<=`), and `equality_expression` represents equality operators (`==` and `!=`). Overall, this grammar defines the basic building blocks of expressions in Catscript, allowing for the creation of more complex and sophisticated expressions as needed.

In summary, Catscript is a programming language that allows developers to write code using a clear and concise syntax. Its grammar includes support for common programming features such as variables, functions, and expressions. With its intuitive design and flexibility, Catscript provides a solid foundation for beginners to learn basic programming principles.

Section 5 - UML

CatScript Multiplication Sequence Diagram



In this UML diagram, the code “66 * 7567” is evaluated through the parser. The User inputs the code “66 * 7567” which is evaluated by the CatScript program. CatScript tells the lexer to tokenize the input code and returns a list of tokens. This token list is parsed via the Parser. The Parser calls the `parseExpression()` function, which will call the `parseFactorExpression()` function because multiplication is parsed in the `parseFactorExpression()` function. Since the input was multiplication of two integers, the `parseIntegerLiteral()` function is called. This `parseTree` is returned to the CatScript program, which then calls the `execute()` function with the `parseTree`. This returns the value 499422, which is the correct result of $66 * 7567$ to the CatScript program, which returns the value to the User.

Section 6 - Design Trade-Offs

Since the parser was generated by hand rather than using a parser generator, the design quality is different. The parser created for CatScript utilized recursive descent parsing. Recursive descent allows for understanding of grammars and the recursive nature of grammars, but this form of parsing is less efficient than using a parser generator. Recursive descent parsing uses top down recursion to formulate a parse tree. Whereas a parser generator like ANTLR uses external files to generate a lexer and parser in a chosen language. The code length from generated code can also be significantly longer than recursive descent parsing. These generated lexers and parsers can be difficult to understand and debug and implementing new ideas even harder. Obscure syntax can also be generated for obvious ideas if done by hand. In CatScript the visitor design pattern cannot be used based on how it is coded. With a parser generator, the visitor pattern can be used and allows for control over the parse tree that cannot be achieved through the generated code.

Section 7 - Software Development Life Cycle Model

Test Driven Development (TDD) was the primary method used when developing the CatScript compiler. Given a test suite with specification for the language, overtime tests were fixed by coding the tokenizer, parser, and Java bytecode converter.

Starting with the tokenizer, 16 tests were used to formulate a working tokenizer. These tests included correct data type, basic syntax, basic keywords, etc. These tests were beneficial in creating a successful tokenizer used in future tests.

The parser included approximately 116 tests for expressions and statement validation. These tests included type checking, expression evaluation, and statement evaluation. Using these tests, code was written to which bytecode could be generated for.

The Java bytecode converter was the last formulated code, with 15 tests that checked for correct conversion to bytecode from CatScript. Tests included, expression compilation, statement execution, and function compilation. These tests were very beneficial in understanding how CatScript is converted to Java bytecode.

Test Driven Development was a great way to manage the progress of the CatScript compiler. Another life cycle model could have been more beneficial, but TDD is great for understanding what needs to be done, and how it can effectively be accomplished.