

Ike Wessel

CSCI 486 – Compilers Capstone Documentation

Section 1: Program. The source file for the code can be found in the /capstone/portfolio folder under the name “source.zip”

Section 2: Teamwork. Our team was comprised of two members. Team member 1 took on the role of product engineer, and team member 2 took on the dual roles of documentation engineer and testing engineer. Team member 2 would design a suite of tests to drive team member 1’s solutions. Team member 1 would then implement the various stages of the CatScript parser, checking their solutions against the tests provided by team member 2. When team member 1’s solutions passed the tests, team member 2 would then document them in detail. At the end of the project, team member 2 then designed three major tests to verify the functionality of the finished product. Approximately 60% of the total team spent on project development was used by team member 1, while the remaining 40% was used by team member 2. The tests can be found in the “NewEvalTests” folder under the “Eval” folder in the test area. Here are the final tests produced by team member 2:

```
DarkKnightBZN
@Test
void functionCallWithForAndIf() {
    //assertEquals(false, evaluateExpression("1 > 2"));

    assertEquals( expected: "Ending Number:6\n", executeProgram( src: "var x = [1,2,3,4,5]\n" +
        "var counter = 0\n" +
        "function counterIncrement(){\n" +
        "if(counter==0){\n" +
        "    for(i in x){\n" +
        "        if(i==4){\n" +
        "            counter=counter+2\n" +
        "        }else{\n" +
        "            counter = counter + 1\n" +
        "        }\n" +
        "        \n" +
        "    }\n" +
        "}\n" +
        "counterIncrement()\n" +
        "print(\"Ending Number:\"+counter)\n"));
}
```

```

DarkKnightBZN
@Test
void weirdMath() {
    assertEquals( expected: "59618\n", executeProgram( src: "function recursiveSolution(number:int){\n" +
        "    if(number<10000){\n" +
        "        number=(number*2)/3+(number*7+2)\n" +
        "        recursiveSolution(number)\n" +
        "    }else{\n" +
        "        return(number)\n" +
        "    }\n" +
        "}" +
        "\n" +
        "var number = 0\n" +
        "recursiveSolution(number)\n" +
        "print(number)"));
}

new *
@Test
void printingTypesWithForLoop() {
    assertEquals( expected: "1atrueanull\n", executeProgram(
        src: "    var stringType = \"a\"\n" +
        "    var integerType = 1\n" +
        "    var booleanType = true\n" +
        "    var listType = [\"a\"]\n" +
        "    var nullType = null\n" +
        "    var printStatement = integerType + stringType + booleanType\n" +
        "    for (i in listType){\n" +
        "        printStatement = printStatement+ i\n" +
        "    }\n" +
        "    printStatement = printStatement + nullType\n" +
        "    print(printStatement)"));
}

```

Section 3: Design pattern. The one real design pattern that we implemented into CatScript was the Memo pattern, which is used in the CatscriptType class in the CatScript compiler, within the “getListType()” method. We used a memo pattern to essentially cache the values of CatScript Types that have been previously queried. We chose to do that instead of just grabbing the type directly to save us time when this function is called multiple times over the course of several compilations. It is worth mentioning that the current solution is not necessarily thread-safe. Here is a screenshot of the method with the Memo pattern implemented:

```
// DONE: memoize this call
2 usages
private static final Map<CatscriptType, CatscriptType> LIST_TYPES = new HashMap<>();
4 usages  DarkKnightBZN +1
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;
}
```

Section 4: Technical writing. Here is the technical documentation written by team member 2:

CatScript Documentation

Created by: Jadeyn Fincher
April 20th, 2023

Introduction:

CatScript is a language developed in CSCI 468 as a capstone compilers class at MSU. This language is basic but has all the tools necessary to be a Turing complete language. The following documentation will cover everything that the language covers, from the grammar to how to use the expressions and statements covered in the grammar. The language is compiled into java byte code during compilation and currently, there is no transpilation support for the language, although javascript implementations of this should be straightforward.

Control flow:

The control flow statements that you have access to in CatScript are if and else statements for branching based on truthiness and for loops for iterations. Both are covered under the statements part of this document and you can find the syntax and example use cases there.

Grammar:

The following is the grammar defined for CatScript.

```
catscript_program = { program_statement };

program_statement = statement |
                    function_declaration;

statement = for_statement |
            if_statement |
            print_statement |
            variable_statement |
            assignment_statement |
            function_call_statement;

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;

function_call_statement = function_call;

assignment_statement = IDENTIFIER, '=', expression;

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 ];

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 } ']';

function_call = IDENTIFIER, '(', argument_list , ')'

argument_list = [ expression , { ',', expression } ]

type_expression = 'int' | 'string' | 'bool' | 'object' | 'list' [ '<', type_expression, '>' ]
```

Statements:

FOR STATEMENT:

Use Case: The for loop allows for iteration based on the input parameters along with a reference identifier to access the current item the loop is accessing. This is the only iteration control flow structure cat script has defined.

Syntax:

KeyWord-Required, UserDefined-Required, Optional

```
for ( Identifier in Expression ){  
    Statements(0 or more)  
}
```

Example Uses:

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

```
for( l in x){  
    print(x)  
    print(l)  
}
```

IF STATEMENT:

Use Case: The if statement is a piece of the control structure to allow branching based on the truthiness of the value being evaluated. If the evaluation is true it will evaluate statements contained in the brackets of the if statement, else if there is an else attached it will evaluate those statements

Syntax:

KeyWord-Required, UserDefined-Required, Optional

```
if ( expression ){  
    Statements(0 or more)  
}
```

OR

```
if ( expression ){  
    Statements(0 or more)  
}else{  
    Statements(0 or more)
```

```
}
```

Example Uses:

```
if(1==1){  
    print("true")  
}else{  
    print("false")  
}
```

ASSIGNMENT STATEMENT:

Use Case:

The assignment statement is used to assign variables/identifier expressions to values after they have already been initialized with the variable statement. The inferred or defined typing defined within the variable statement must be followed to avoid compilation errors.

Syntax:

Keyword-Required, UserDefined-Required, Optional

Identifier = Value

Example Uses:

```
A = 2  
B = "Hello world"
```

PRINT STATEMENT:

Use Case:

The print statement is used to evaluate an expression and print the value back out onto the system. The value in the local server will display prints below the field allowed to create your program.

Syntax:

Keyword-Required, UserDefined-Required, Optional

print(expression)

Example Uses:

```
print("hello world")  
print(1+2)
```

```
Var a = 3+2  
print(a)
```

VARIABLE STATEMENT:

Use Case:

The variable statement is used to define new variables and their values. They also support set typings if you do not want to use an inferred type system for these. This is also if you want to implement a strongly typed system. The value of the variable statement can later be changed with the assignment statement, but the typing of the variable cannot be changed.

Syntax:

Keyword-Required, UserDefined-Required, Optional

```
var Identifier : type_expression = expression
```

OR

```
var Identifier = expression
```

Example Uses:

```
var a = 2  
var:int a =2
```

```
var b = "hello"
```

FUNCTION CALL STATEMENT:

Use Case:

The function call statement is used to call an already-defined function to evaluate it.

Syntax:

Keyword-Required, UserDefined-Required, Optional

```
functionName(arguments(0 or more))
```

```
var a = calculator(2,2)
```

```
print(calculator(2,2))
```

FUNCTION DECLARATION STATEMENT:

Use Case:

The function declaration statement is used to create callable functions that perform certain user-defined functions with given or no parameters. These functions can return values and have side effects that can be used to manipulate data as need be.

Syntax:

Keyword-Required, UserDefined-Required, Optional

```
function identifier ( parameter:type_expression(0 or more) ) : type_expression {  
    Statements(0 or more)  
}
```

OR

```
function identifier ( parameter:type_expression(0 or more) ){  
    Statements(0 or more)  
}
```

Example Uses:

```
function calculator (firstItem:int,secondItem:int):int{  
    return(firstItem+secondItem)  
}
```

RETURN STATEMENT:

Use Case:

The Return statement is used to evaluate an expression and return the value to whatever is calling it. A major use-case of this is within function calls to return a value after the instructions have been completed within the body of the function.

Syntax:

Keyword-Required, UserDefined-Required, Optional

```
return expression
```

Example Uses:

```
function calculator (firstItem:int,secondItem:int):int{  
    return(firstItem+secondItem)
```


}

Expressions:

EQUALITY EXPRESSION:

Use Case: The equality expression is used to evaluate whether two expressions are equal depending on the operand and return a boolean value of the result.

Operands:

!= -> If both expressions are not equal to each other this will evaluate to true

== -> If both expressions are equal to each other this will evaluate to true

Syntax:

Expression1 **Operand** Expression2

Example Uses:

1!=2 -> Evaluates to true

1==2 -> Evaluates to false

COMPARISON EXPRESSION:

Use Case: The comparison expression is used to evaluate the values of the left-hand side expression and the right-hand side expression and compare the values to return a boolean value based on the result.

Operands:

> -> If the left expression is greater than the right expression this will return true else it will return false

>= -> If the left expression is greater or equal to the right expression this will return true else it will return false

< -> If the left expression is less than the right expression this will return true else it will return false

<= -> If the left expression is less than or equal to the right expression this will return true else it will return false

Syntax:

Expression1 **Operand** Expression2

Example Uses:

1>2 -> Evaluates to false

2>2 -> Evaluates to false

2>=2 -> Evaluates to true

2<3 -> Evaluates to true

ADDITIVE EXPRESSION:

Use Case: The additive expression is used to add or subtract the value of the right-hand expression and the left-hand expression and returns the calculated value based on the result.

Operands:

- + -> Returns the left and righthand expressions values added together
- -> Returns the value of the left-hand expression minus the value of the right-hand expression

Syntax:

Expression1 **Operand** Expression2

Example Uses:

1+2 -> Evaluates to 3
1-2 -> Evaluates to -1

FACTOR EXPRESSION:

Use Case: The factor expression is used to handle division and multiplication between the right-hand expression and the left-hand expression and returns the calculated value based on the result.

Operands:

- * -> Returns the right-hand and left-hand values multiplied together as a value
- / -> Returns the left-hand value divided by the right-hand value as a value

Syntax:

Expression1 **Operand** Expression2

Example Uses:

10*2 -> Evaluates to 20
10/2 -> Evaluates to 5

UNARY EXPRESSION:

Use Case: The unary expression is to take a value of a boolean or integer and apply a negative or switch the boolean value depending on the

Operands:

- not** -> Applies a boolean switch to booleans and returns the switched boolean value
- -> Applies a negative switch to any value and returns that value

Syntax:

Operand Expression

Example Uses:

not true -> Evaluates to false
- 10 -> Evaluates to -10

PRIMARY EXPRESSION:

Use Case: The primary expression is used to evaluate literals, expressions, and variables down to their core values.

Syntax:

Expression

Example Uses:

1 -> Evaluates to 1

false -> Evaluates to false

[1,2,3] -> Evaluates to [1,2,3]

Var x = 10

x -> Evaluates to 10

TYPE EXPRESSION:

Use Case: The type expression is used to set a defined type to whatever is being called

Syntax:

typing

Supported Typings:

There Are 5 main types supported within CatScript: Integer, String, Boolean, Object, and List types.

INTEGER TYPE:

They can support all whole numbers such as 1,2,100000, 28818.... No support for floating points within this type.

Formatting: CatScript will take the value without formatting

STRING TYPE:

This type can support strings similar to Java's strings such as "Hello world", "T", and "test'problem".

Formatting: Surround the value with double quotes -> " "

BOOLEAN TYPE:

This type supports two values: true and false. Values work similarly to Java boolean values

Formatting: CatScript will take the value without formatting

OBJECT TYPE:

Object typings work in the way that they unbox a value from a primitive. Certain operations will not become available if an object type is set such as value addition. String concatenation is still available.

Formatting: CatScript will take the value without formatting

LIST TYPE:

The list type is a bit more complicated as you can define a type of the list within it. Lists support iterations being called on them, so they can be used inside of for-loops for easy iterations

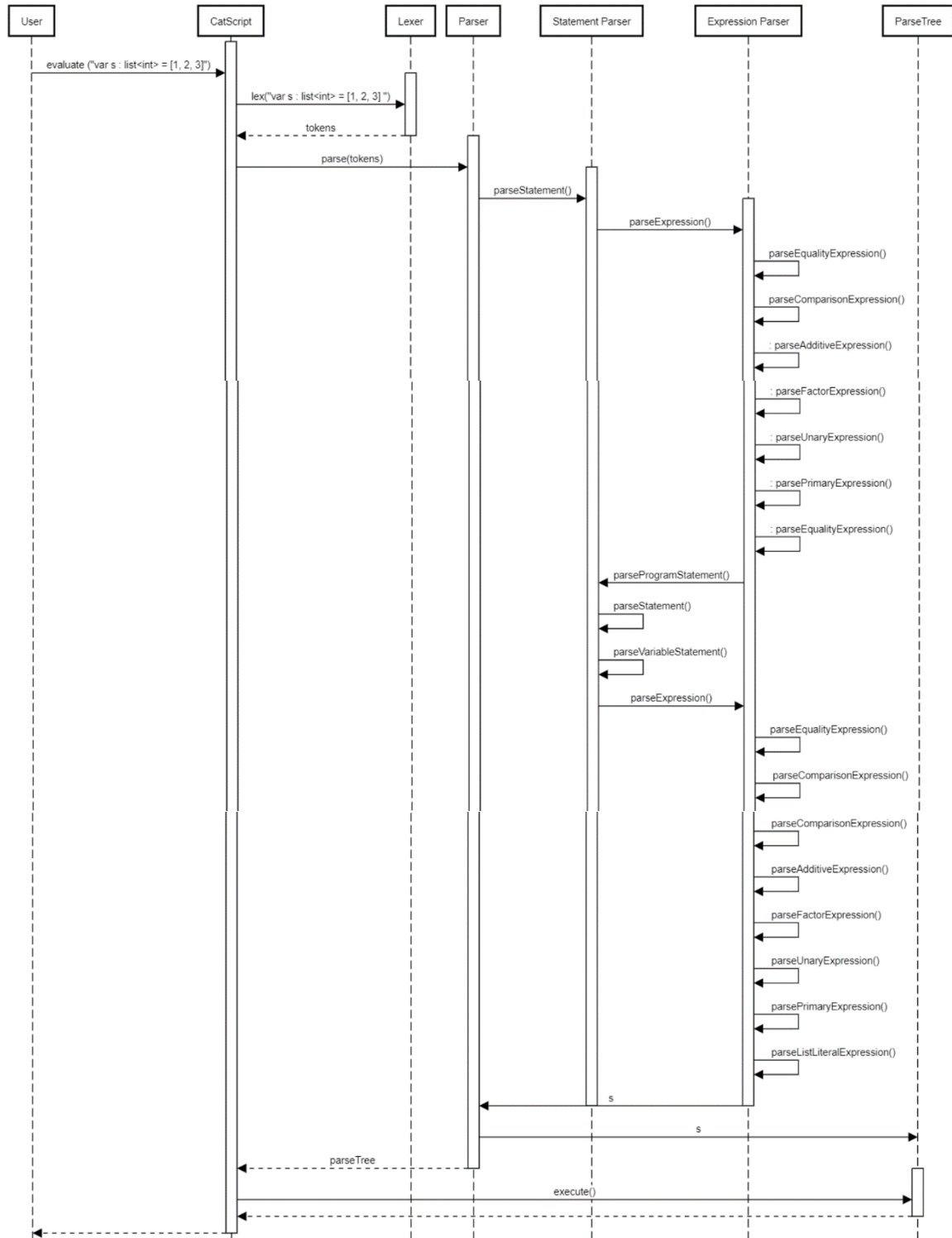
Keyword-Required, UserDefined-Required, Optional

Formatting: **list** OR **list<type_expression>**

Section 5: UML. Here is a UML diagram that illustrates a standard compilation cycle in the CatScript compiler:

SEE NEXT PAGE

Catscript Var Statement Diagram



Section 6: Design trade-offs. There was one main trade-off that we had to make over the course of designing this compiler. Many first-time compiler developers choose to use a tool such as Lux to GENERATE a compiler based off of a carefully crafted set of regular expressions. Our team chose to take another route, and built the whole compiler by hand, using the Recursive-Descent strategy, instead of using a tool to build it for us with regex.

There are a couple drawbacks to this approach. For one, it takes significantly more time to design and write, by hand, all of the overhead classes and methods in order to support a larger application like a compiler. This overhead only gets larger as you move on to more complicated phases like transpilation farther down the development line. Also, it requires a much, MUCH more mature understanding of the compilation process to attempt this method of implementation that many developers simply do not have.

However, the advantages of the by-hand, Recursive-Descent approach far outweigh any drawbacks. Because the compiler is not hidden inside the black box of an auto-generating application like Lux, debugging is a thousand times easier, because all of the code is written by a human developer, which means that variable names, method names, logical flow, etc., should, in theory, be easier to read, easier to understand, and easier to track throughout the debugging process. The Recursive-Descent method of building a compiler also results in a codebase that is often 10+ times smaller than the generated code of the regex method and more efficient computationally. And finally, the by-hand method forces the developer to gain a much deeper understanding of the language that they are designing, as well as the internals of any languages that they intend to compile to. This knowledge will only benefit them as they work in the languages that are familiar to them.

Section 7: Software development life cycle model.

We used a test-driven development (TDD) model for our development. Everything started with the requirements for what our final language had to be able to do. After those were established, team member 1 wrote a suite of tests that would theoretically guarantee the satisfaction of those parameters. Then team member 2 wrote the tokenizer, parser, evaluator, and verifier, and the final code through the tests that team member one had provided. This worked extremely well for our team because the nature of the application we were trying to build lent itself very well to the TDD approach. As soon as we knew what the language needed to be able to do, then it was relatively simple to work off of a test suite without worrying that some needed future functionality might be missed, since the language was already well defined before we even started coding.