

CSCI 468: Compilers

Spring 2022

Maxwell Lineberger

**Partner:
Taner Rubino**

Section 1: Program

[/capstone/portfolio/source.zip](#)

Section 2: Teamwork

Team member 1 was the lead on writing the code for this project, and spent a majority of the time on the tokenizer, parser, evaluation, and bytecode. Most of this was geared towards passing the tests for each of those sections as this was a test driven project. Team member 2 provided additional tests that were aimed to further improve the code. They also supplied the Catscript documentation which can be found in section 4 under technical writing. Team member 1 spent about 95% of the time on this project and team member 2 spent about 5% of the time for this project.

Below are the tests that team member 2 provided for team member 1:

```
@Test
void nestedIfStatementWorksProperly(){
    assertEquals("1\n", executeProgram("var x = 5\n" +
        "if(x==7){ print(0) } else { if(x<7){ print(1) } else { print(-1) } }"));
    assertEquals("0\n", executeProgram("var x = 7\n" +
        "if(x==7){ print(0) } else { if(x<7){ print(1) } else { print(-1) } }"));
    assertEquals("-1\n", executeProgram("var x = 9\n" +
        "if(x==7){ print(0) } else { if(x<7){ print(1) } else { print(-1) } }"));
}

@Test
void typeInferenceWorksForListTypes(){
    VariableStatement var = parseStatement("var x = [1, 2, 3 ,4]");
    assertEquals(CatscriptType.getListType(CatscriptType.INT), var.getExplicitType());
    var = parseStatement("var x = [true, false, true, true]");
    assertEquals(CatscriptType.getListType(CatscriptType.BOOLEAN), var.getExplicitType());
    var = parseStatement("var x = [\"abc\", \"string\", \"foo\"]");
    assertEquals(CatscriptType.getListType(CatscriptType.STRING), var.getExplicitType());
}

@Test
void forStatementEnsuresSyntax(){
    assertEquals(ErrorType.UNEXPECTED_TOKEN, getParseError("for(x in [1, 2, 3] { print(x) }"));

    assertEquals(ErrorType.UNEXPECTED_TOKEN, getParseError("for (x [1, 2, 3]) { print(x) }"));
    assertEquals(ErrorType.UNEXPECTED_TOKEN, getParseError("for (x,y in [1, 2, 3]) { print(x) }"))
}
```

Section 3: Design Pattern

One design pattern used in this project was memoization, also known as flywheel. This is located in the `CatscriptType` class, and is used in the function `getListType`.

Below is the code we started with, which is functional but not efficient:

```
public static CatscriptType getListType(CatscriptType type) {  
    return new ListType(type);  
}
```

We want to avoid creating new types if they already exist, so we start by creating a hashmap which will contain each type. When we get the type, we will check to see if it already exists in the hashmap. If it does, then we will return that from the cache. If it does not, then we will create the new list type and add it to the cache. Now if the same type is called multiple times then it will only be created as a new type once.

Below is the improved code with the pattern implemented:

```
private static Map<CatscriptType, ListType> cache = new HashMap<>();  
public static CatscriptType getListType(CatscriptType type) {  
    ListType listType = cache.get(type);  
    if (listType != null){  
        return listType;  
    }  
    else{  
        ListType newListType = new ListType(type);  
        cache.put(type, newListType);  
        return newListType;  
    }  
}
```

Section 4: Technical Writing

Catscript Documentation

Catscript is a simple functional programming language that uses an easy to understand syntax and has common features of many other programming languages.

An example of a simple Catscript program to print out a list looks like:

```
function printList(myList : list<int>) {  
  for(x in myList){  
    print(x)  
  }  
}
```

```
printList([1,2,3])
```

Output:

```
1  
2  
3
```

Features

Types

The type expression is expressed in the Catscript grammar as:

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

The data types that exist within Catscript as shown above are:

- int: Stores integer values such as 1 or 25 or -10
- string: Stores words or sentences such as "hello" or "this is a string"
- bool: Stores true or false values
- object: A generic type, can store the values for any other type
- list: Stores a collection of values of a single type, can be given a type using <> such as list<int>, lists are *immutable*

Expressions

The various types of expressions are represented in the grammar as:

```
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, ")"
```

Expressions in Catscript are used as the basis of the program logic, which are evaluated at runtime to produce a value.

Expressions can be used to represent simple arithmetic operations such as addition, multiplication, or negation as shown above within *additive_expression*, *factor_expression*, and *unary_expression*.

- The additive expression is used to add integers as well as to concatenate strings together.
- The factor expression is used to multiply two integers
- The unary expression is used to take the opposite of an int using '-' or a boolean using 'not'

Expressions can also be used to compare two values together as shown in *equality_expression* and *comparison_expression* which will evaluate to a boolean value

- The equality expression is used to check if any two values are equivalent or not
- The comparison expression is used to compare integers to each other

Expressions can also be a single value. These expressions are considered primary expressions and will evaluate to the single value within them.

Expression examples:

- additive expressions: `2 + 5` , `"hi " + "there"`
- factor expressions: `3 * 4` , `25 * (9 * 3)`
- unary expressions: `-5` , `not true`
- equality expressions: `5 == 5` , `10 != 6` , `true == true` , `null != false`
- comparison expressions: `1 < 3` , `6 >= 5`
- primary expressions: `1` , `true` , `"hello"` , `null`

Variables

Defining Variables

The variable statement is expressed in the Catscript grammar as

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

As shown in the grammar, a variable is defined using the *var* keyword followed by the name of the variable represented as *IDENTIFIER* above

A variable's type can be declared using a colon : after the name of the variable. If no type is given, then the parser will determine the type at runtime.

The variable will be assigned to the value that the *expression* on the right side of the '=' evaluates to

Example of defining a variable in Catscript with an explicit type:

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

Example without an explicit type (string type will be assumed by the parser):

```
var x = "foo"
```

Assigning Variables

The assignment statement is expressed in the grammar as:

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

The assignment statement is used to reset the value of a variable that is first defined as shown in the *Defining Variables* section above.

The variable defined under the name represented as *IDENTIFIER* above will be set to the value that the expression on the right side of the '=' evaluates to

Example of assigning a variable in Catscript:

```
var x = "foo"  
x = "new string"
```

Printing

The print statment is defined in the grammar simply as

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

The print statement is used to display a value to the program output.

A print statement is defined using the *print* keyword followed by the expression to be printed within parenthesis()

The expression will be evaluated and the value will be displayed as program output

An example of a print statement:

```
print(2 + 2)
```

Output:

4

For loops

The for statement is expressed in the grammar as:

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

The for statement in Catscript is used exclusively to iterate over lists.

The for statement is defined with the *for* keyword followed by a variable name, the *in* keyword, and an expression representing a list within all parenthesis()

On each iteration of the loop, the variable defined as *IDENTIFIER* will be set as the next element in the list defined in 'expression' above

Statements within the body of the loop (designated by brackets {}) are executed in order repeatedly based on the length of the list

As the statements within the body of the statement are executed, the *IDENTIFIER* variable shown above will behave as a local variable each iteration through the loop.

A simple implementation of a for loop:

```
var alist = [1,2,3]
```

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

Output:

1
2
3

If Statements

The if statement is expressed in the grammar as:

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

The if Statement in Catscript is used to test conditional statements, and is defined using the *if* keyword

If the expression defined as *expression* above evaluates to true then the statements contained within brackets {} after the expression will be executed in order

If the expression evaluates to false then the statements contained within the brackets after the *else* will be executed.

If statements can be contained within themselves, when executed they will evaluate the outermost if statement first and then work inwards.

A simple example of a Catscript If statement:

```
x=5  
  
if (x > 10){  
    print("Greater than 10")  
}else{  
    print("Less than 10")  
}
```

Output:
Less than 10

Functions

Function Declaration

The function declaration statement is expressed in the Catscript grammar as

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

As shown in the grammar, a function is defined using the *function* keyword followed by the name of the function represented above as *IDENTIFIER*

The parameters of the function are declared within parenthesis, with each parameter separated by a comma and optionally using a colon to declare its type as shown in the example below.

The return type of the function is declared using a colon after the parameter list followed by the type the function should return. If no return type is specified, then the return type *void* is assumed.

The body of the function containing the statements to be executed upon function call is defined within brackets{}

If the keyword *return* is used at the beginning of a statement within the function body, the function will stop executing and if an expression is given it will return the result of the expression.

An example of a function declaration in Catscript:

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

An example of a similar function declaration with no return type or parameter types:

```
function printAdd(x,y,z){
    print(x + y + z)
}
```

Function Calls

A function call is represented in the Catscript grammar as

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

A function can be executed by using its name followed by a list of arguments to be used as parameters within parenthesis().

Each statement listed in the function's body as shown in the *function declaration* section above will be executed

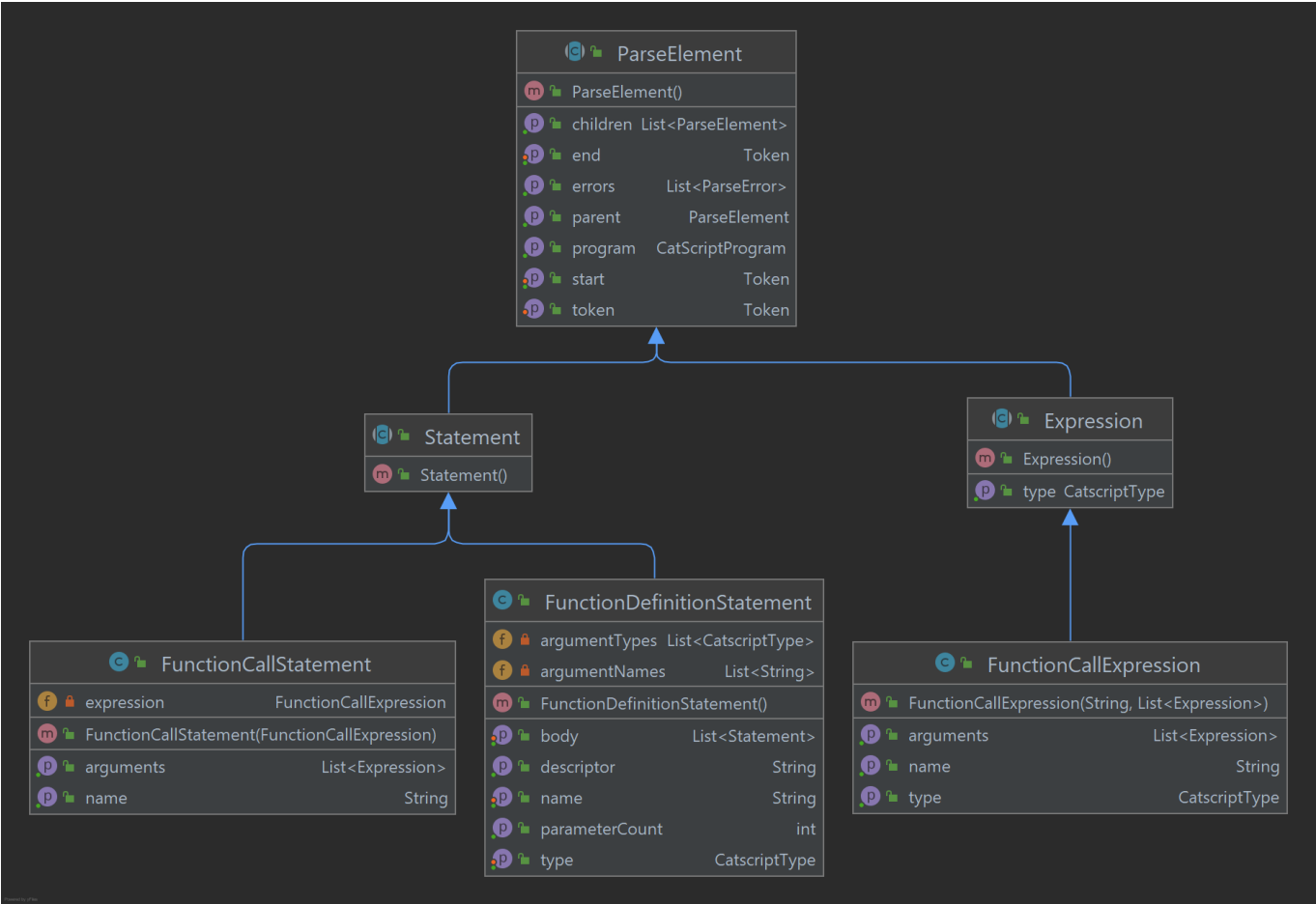
An example of a function call, using the function declaration for *add* from above

```
num = add(1,2,3)
print(num)
```

Output:
6

Section 5: UML

In the UML diagram below, we can see a few of the many relationships within the parser of Catscript. Although there are many elements of *Statement* and *Expression*, the focus of this diagram is on functions in Catscript. The diagram clearly shows that *Statement* and *Expression* are the two kinds of *ParseElement*, and that *FunctionCallExpression* is an *Expression*. *FunctionDefinitionStatement* and *FunctionCallStatement* are both statements as well.



Section 6: Design Trade-Offs

The major design trade off for this project was using recursive descent over a parser generator. The main reason for this choice is that it is simpler to write and gives a better understanding of how a compiler works. Parser generators rely on trickier code and how it works is much less intuitive. Although it is a bit slower and requires more code, recursive descent was a better choice for this project. It is much easier to understand the natural progression of the grammar through a parser generator because of the top-down approach. Each expression and statement call each other, and this makes it easier to debug as well. It was very obvious when one of the pieces was not working correctly.

Section 7: Software Development Life Cycle Model

This project used Test Driven Development, which means we used tests to make sure each section of our code was written correctly. We used tests for the tokenizer, parser, evaluation, and the byte code. Each of those sections performs a different task so new tests were needed to make sure they were all working correctly. We relied heavily on these tests, and it saved a lot of time over manually testing each section. It helped with the structure of the coding process, because it was very clear which tests were passing and which were not. By having tests before we start coding, it made it easier for our team to focus on the more important pieces of this large compiler puzzle. It was always very clear which part we needed to work on next, and when we were finished with a section of the compiler. Sometimes the tests were not perfect, and after completing one section the next section would have issues. We had to go back and change some previous code, but since the old tests were still there we could tell if we did it correctly. Overall, Test Driven Development is a great approach to a project such as this, and we will surely be implementing it in future projects.