

Montana State University Computer Science Department

Senior Portfolio

Section 1: Program

[CSCI 468 source directory .zip](#)

Program specifications

```
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, '>']

```

Section 2: Teamwork

Describe how your team worked on this capstone project. List each team member's primary contributions and estimate the percentage of time that was spent by each team member on the project. Identify team members generically as team member 1, team member 2, etc.

For this project, the primary work, involving deriving the starting code base provided and implementing all missing features outlined by the grammar above, was done solely by me.

For the teamwork portion of this project, I selected a partner who provided documentation for my code, as well as QA services in the form of three additional tests to add to my program's test suite. In return, I provided the same items (documentation and three additional tests) for my partner's capstone portfolio.

The documentation provided by my partner can be found under Section 4, and the three additional tests provided by my partner can be found below.

```
@Test
public void testFunctionCallWithFunctionCallAndForLoop() {
    String source = "function sumToFive(): int { var sum = 0\n for (x in [1, 2,
3, 4, 5]) {\n" +
        " sum = sum + 1\n } return sum }\n" +
        " function sumToTen(): int { return sumToFive() + sumToFive() }\n"
+
        " function sumToTwenty(): int { return sumToTen() + sumToTen() }\n"
+
        " print(sumToTwenty())";
    Object result = executeProgram(source);
    assertEquals("20\n", result);
}
```

```
@Test
public void testFunctionCallSumList() {
    String source = "function sumList(x: list<int>): int { var sum = 0\n for (e
in x) {\n" +
        " sum = sum + e\n } return sum }\n" +
        " print(sumList([1,2,3,4,5]))";
    Object result = executeProgram(source);
    assertEquals("15\n", result);
}
```

```
@Test
public void testIsOddFunction() {
    String source = "function isOdd(x: int): bool { if ((x / 2) * 2 == x) {
return false } else { return true }} " +
        " print(\"1 is odd:\") print(isOdd(1))" +
        " print(\"2 is odd:\") print(isOdd(2))";
    Object result = executeProgram(source);
    assertEquals("1 is odd:\ntrue\n2 is odd:\nfalse\n", result);
}
}
```

Section 3: Design pattern

Identify one design pattern that was used in your capstone project and describe exactly where in the code it is located. Highlight the design pattern in yellow. Explain why you used the pattern and didn't just code directly.

One design pattern used in this project was the Memoization Pattern, which can be found within the `getListType()` method (located at line 37) within `src > main > java > edu > montana > csci > csci468 > parser > CatscriptType.java`.

This pattern was used because initially, whenever a CatScript list was created, a new type was also created to serve as the list's associated type, meaning that creating several integer lists would cause the repeated initialization of the `list<int>` type. While this technique wasn't particularly expensive, the method could be improved upon with the Memoization Pattern.

By memoizing type access in this way, a list type can be cached and used multiple times without needing to be reinitialized, thus saving computation time.

Section 4: Technical writing

Catscript Guide

Introduction

Catscript is a simple, statically-typed scripting language that can be executed directly by the Catscript application or compiled to JVM bytecode. The Catscript execution environment is designed to port easily to the JVM, so features like scoping operate similarly to Java. Catscript files are parsed using a recursive descent parser, which makes debugging syntax or parsing errors much easier than other types of parsers.

Catscript's list type is also fairly unique in that all lists are static and cannot be changed after they are created. Strings operate similarly to Java lists.

For this introductory guide to Catscript, we will start with the basics, going from printing simple expressions, to some Catscript type examples, to function declarations and calls.

The Basics

Here is an example of some print statements:

```
print("Hello World\n")
var x = "Hello"
var y: string = " World\n"
print(x)
print(y)
print(x + y)
```

```
> Hello World Hello World Hello World
```

Some strings and ints

```
print(2 * 3)
print("2" + 3)
print(2 * 3 + "4")

> 6 23 64
```

The print statement takes only one argument and evaluates similarly to Java. It outputs to standard out.

Types

The basic Catscript types are:

- int - a 32-bit integer
- string - a Java-style string
- bool - a boolean value
- null - the null type
- object - any type of value
- void - what functions return when they don't return anything

All of these types can be encompassed in a list. List types are parsed recursively, so lists of lists can be nested arbitrarily deep. The other two non-types are the null and void types.

An example of creating a variable with each given type:

```
var x: int = 1
var x_2 = 1
var y: string = "example"
var z: bool = false
var a: object = "example object"
var b: list<int> = [1,2,3]
```

Catscript supports type inference when declaring variables, so the type information is unnecessary during assignment. Type annotations are, however, required for declaring parameters for functions but not for return types.

Type casting

Some types in Catscript can be assigned to other types, or "cast".

The rules are:

- No types are assignable from void.
- Every type is assignable from null.
- Otherwise, we follow Java's assignability rules

Lists in Catscript are immutable, which means that all list types are covariant. This means that assignability for lists works the exact same way as the underlying type!

Operators

Catscript supports these basic math symbols:

- +
- -
- *
- /

For comparisons, Catscript includes:

- ==
- !=
- <=
- >=
- <
- >

If, Else If, Else Statements

If, else if, and else statements are very javascript-like, though they don't require semicolons.

```
var x = 1
if (x < 1) {
  print("x is less than 1")
} else if (x == 1) {
  print("x is 1")
} else {
  print("x is greater than 1")
}

> x is 1
```

For loops

For loops in Catscript are also javascript-like, but the iteration part is less flexible. The form of the iteration expression in Catscript is always "(x in list_type)".

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

Control Flow

Functions

Functions in Catscript are the main way to control the flow of the program. A module system will not be implemented to encourage the use of simpler programs. However, the user is invited to make the main Catscript file as long as they want.

```
function foo() : int {  
    var x = 100  
    return x  
}  
print(foo())  
  
> 100
```

Variables created in a function are local to that function. Variables created outside of a function are global, and assigning to them within a function will apply to the global variable. There is no variable shadowing because a variable cannot be created with a name already in use.

```
var y = 1  
function foo() : int {  
    y = y + 1  
    return y  
}  
print(y)  
print(foo())  
print(y)  
print(foo())  
print(y)
```

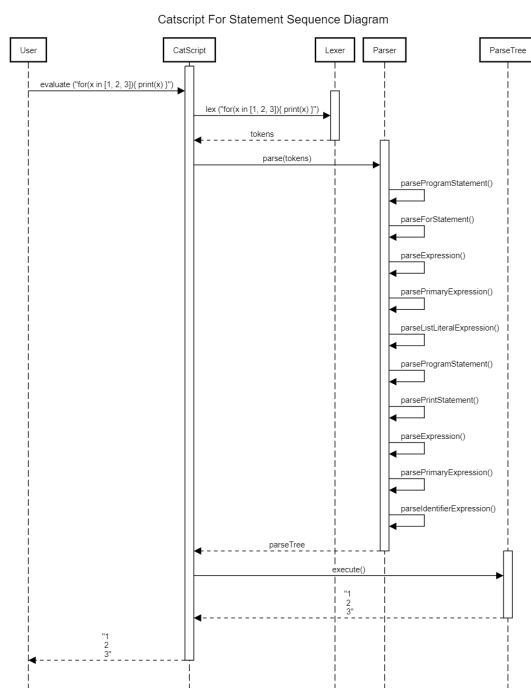
Comments

Line comments are supported in Catscript and are implemented by the double slash

```
// Comments!!  
print("Not a comment")  
  
> Not a comment
```

Section 5: UML

Click to view the full image.



Section 6: Design trade-offs

Describe a design trade-off decision (e.g. execution time vs. space requirements or compile time) in your capstone project and justify the design decisions that you made.

A major design trade-off that was made for this project was the decision to write a full parser by hand rather than using a parser generator such as Lex or YACC. While it would ultimately be faster to use a generated parser for this project, it was decided that it would be much more intuitive and educational to create one using a recursive descent technique. In doing so, I was able to gain a much deeper understanding of how compilers work, as well as how to create a Java-based programming language completely from scratch.

Section 7: Software development life cycle model

Describe the model that you used to develop your capstone project. How did this model help and/or hinder your team?

To develop my capstone project, I followed a test driven development model. Having been provided a comprehensive test suite, work was completed over a period of several “checkpoints” wherein a subset of tests within the suite were required to pass in order for portions of the code to be considered complete/correct.

Having worked with an agile development model in the past, I vastly preferred working in a test driven model as it was much easier to meet goals on time and achieve code correctness when all required tasks were defined in such a concrete way.

However, there were some challenges associated with this model, most notably the fact that although tests were for the most part independent, some tasks were tightly coupled with others, requiring that more code be complete on average in order to pass one test, which was not always immediately obvious.