Catscript Documentation

Expressions

—---

Type Literal Expression:

Catscript uses type literals to represent data within its language. This data is stored in tokens and the type is recognized by the tokenizer. The parser will then store the data in a data structure to evaluate and compile at runtime.

Examples of type literals in Catscript:

```
"hello catscript!"  // string literal
123                    // int literal
true                 // boolean literal
[1,2,3]               // list literal
```

—---

Additive Expression:

The additive expression consists of both a right-hand side and a left-hand side expression. The order of evaluation is left-hand side to right-hand side followed by the operator (‘+’ or ‘-’).

Examples of additive expressions:

```
1 + 2
3 - 4
```

—---

Factor Expressions:

The factor expression is similar to additive expression since it also consists of a right-hand side and a left-hand side. Both sides are evaluated and then the operator (‘*’ or ‘/’) is applied from left to right on the expressions.

Examples of factor expressions:

```
1 * 2
3 / 4
```

—---

—---

Unary Expressions:

In Catscript the unary expression can be used to represent a negative integer or negate a boolean value. Either (‘-’ or ‘not’) is a valid input for unary expressions.

Examples of unary expressions:

```
-10
not true
```

—---

Equality and Comparison Expressions:

Equality expressions are very similar to comparison expressions except comparison expressions have a higher precedence than equality. Both of these expressions contain a left side and right side. These expressions both return a boolean value after evaluation.

```
2 == 3 // evaluates to false
2 < 3   // evaluates to true
2 <= 3  // evaluates to true
2 >= 3 // evaluates to false
2 > 3  // evaluates to false
```

—---

Identifier and Function Call Expressions:

Function call expressions are an identifier that stores a value after evaluating the invoked method following the function name. Identifiers alone will simply be stored within the expression by the parser, where the function call will parse and store the arguments as well.

```
add(1,2)      //function call
//     add is the identifier
// 1 and 2 are the arguments
```

—---

Statements

—---

Print Statement:

Print statements are recognized in Catscript as “print(‘expression’)”. The parser will evaluate the expression inside the parentheses and call an internal print command.

Examples of Print statement:

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

—---

If Statements:

Catscript recognizes If and else statements. Else-if statements are not implemented within the language. If statements take a boolean expression along with a block body that gets stored with the If statement as true statements. When the If statement evaluates to true, the list of true statements then are evaluated. Otherwise Else statements are evaluated.

Examples of If Else statements:

```
if(1 < 2){
   //  true statements
}
else{
   // false statements
}
```

—---

For Statements:

For statements in Catscript are only designed to iterate through lists using the ‘in’ keyword. The process of the for loop will continue to store values as long as there is a “has next” element in the list. Catscript can not iterate on conditions given a boolean value such as (index < list.size).

Examples of for statements:

```
var myList : list<int> = [1,2,3]
for(var x in myList){
   // do something
}
```

—---

—--

Variable and Assignment Statements:

Variables are identified by the ‘var’ keyword in Catscript. The parser will store the expression to the identifier that follows ‘var’. Scoping is also associated with assignment statements, where variables are pushed in scope and popped when there is a change in scope. Global variables are stored in a field that is maintained throughout the life of the program.

Examples of Assignment Statements:

```
var x : int = 3
x = x+1
```

—--

Return Statements:

The return statement in Catscript acts as expected. Once the parser hits the keyword ‘return’ it will halt all execution, evaluate expressions and assign the return value to its function definition. All local variables will be popped off the stack.

Examples of Return Statement:

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

—--

Function Definition Statement:

Function definition statements will begin with the keyword ‘function’ where the identifier is stored along with the argument identifiers and types followed by the statements inside the block body. Functions are stored as objects at runtime and once a return statement is executed the value of that return is stored and the function definition completes or a closing ‘}’ is matched.

Examples of Function Definition Statement:

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

—--

<u>Catscript Keywords</u>:

- **else**
- **false**
- **function**
- **for**
- **in**
- **if**
- **not**
- **null**
- **print**
- **return**
- **true**
- **var**

```
 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, '>']
```

CatScript Types

CatScript is statically typed, with a small type system as follows

- 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