

CSCI 468 Compilers Portfolio

SPRING 2022

Emilia Bourgeois | Trey Popp

Section 1: Program

A zip file of the source code is included in this directory in [source.zip](#).

Section 2: Teamwork

For the capstone project, our team separately used test driven development to develop separate recursive descent parsers. Both partners also wrote the technical documentation for the other partner's project. We each wrote tests to assist each other in writing a fully complete parser. My partner's tests are provided in [src/test/java/edu/montana/csci/csci468/demo/PartnerTests.java](#).

Section 3: Design pattern

The Memoization Pattern is used to memoize calls to [CatScriptType#getListType\(\)](#). If we are creating a list we don't want to store the type of each element with the element itself more than we have to, so instead we use our design pattern. A hash map is used as a cache to store different variants of list types. This frees up memory and sufficiently implements the Memoization Pattern

```
static HashMap<CatScriptType, ListType> cache = new HashMap<>();
public static CatScriptType getListType(CatScriptType type) {
    ListType listType = cache.get(type);
    if(listType == null){
        listType = new ListType(type);
        cache.put(type,listType);
    }
    return listType;
}
```

Section 4: 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
}
```

Catscript Keywords:

- 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, '>']

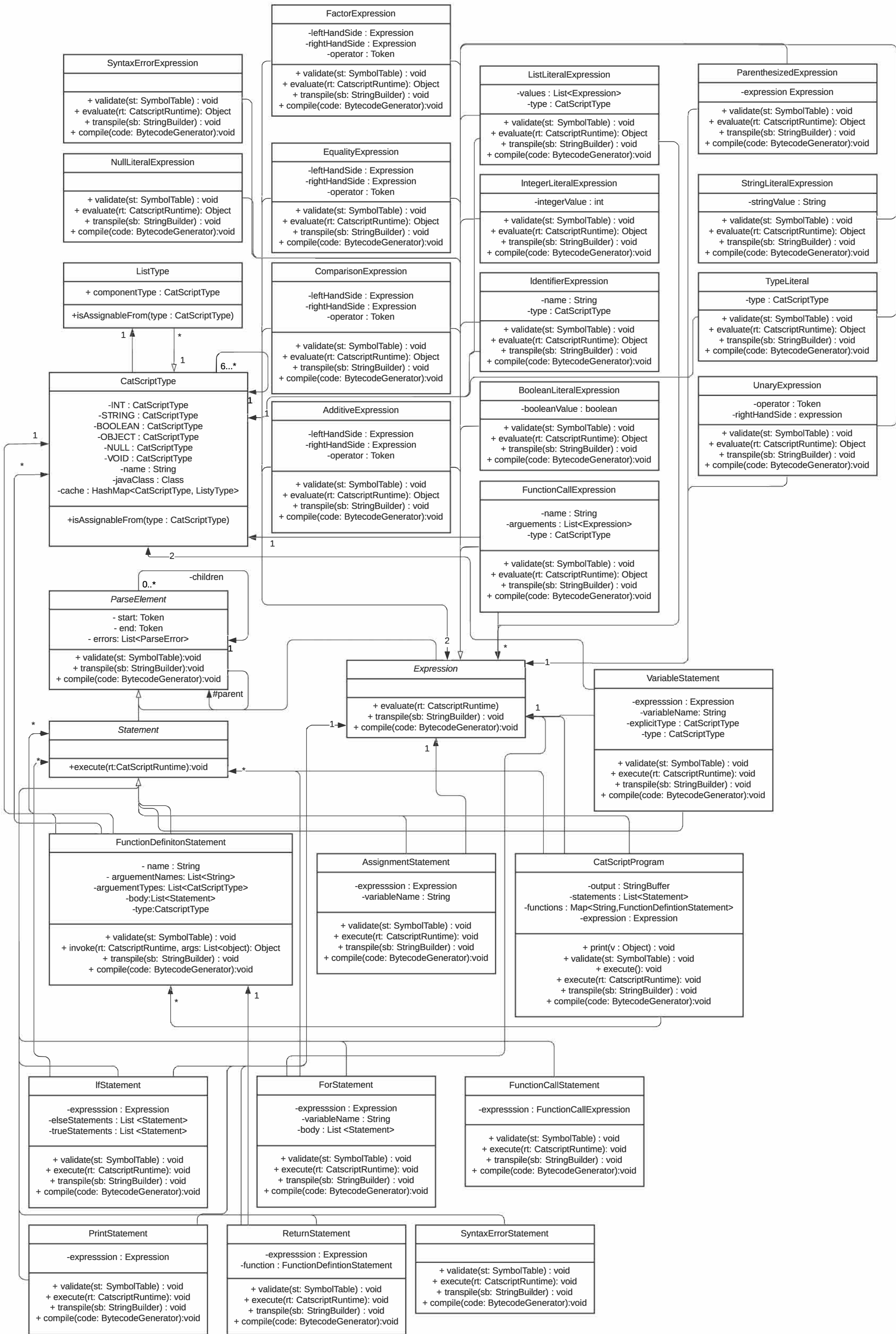
```


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

Section 5: UML



Section 6: Design trade-offs

We used a handwritten recursive descent instead of a parser generator to develop our catscript parser. Parser generators are the typical method for creating a parser in a compilers course. All they require is an input grammar, and most of the course would have been dedicated to developing that. It's simple to write a specification if the input format is close to normal. The end result is typically easily maintainable and understood. These parsers are also faster than hand written ones, but not in the case of handwritten recursive descent parsers. However, parser generators can sometimes reject grammars so research into the specific generator must be done before hand.

A handwritten recursive descent parser comes with many advantages. The programmer gains a deeper understanding of all components, the parser is faster because it begins at the start symbol of the program with no back tracking, the output creates a parse tree, a incredibly useful and fast data structure. However, all the code had to be handwritten, and the complexity of the project was high. A tokenizer, parser, evaluator, and bytecode generator all had to be implemented which took significant effort. This method is also less space efficient than other methods due to the large amounts of function calls and recursion.

Section 7: Software development life cycle model

We are using Test Driven Development (TDD) for this project. TDD is a software development approach where test cases are created to specify and validate every small functionality of the program. Tests are tested first, and if they fail new code is written to fulfill them. At the beginning of the project having so many tests to pass was daunting. Even finding where to start was troublesome. However, after the first checkpoint this is our new favorite life cycle model. What was unclear and insurmountable and the beginning became a task list of little check boxes. Easily allowing us to isolate a specific piece of code to implement correctly before tackling the rest of the project. Had the development model been more traditional to university courses having to write the thousands of lines of code required would have been impossible.