

Portfolio:

Zip File:

<https://github.com/DrDisturbance/csci-468-spring2023-private/blob/main/capstone/portfolio/source.zip>

Teamwork: Well, I coded the project, then jacob wrote a few tests, we found a bug and I fixed it, and then he wrote the documentation and I'm finishing out this part. If we're going by time teammate number 1 spent over 110 hours total on this project, and teammate number 2 spent 2.5 hours or so creating the tests and then the documentation

Design Pattern:

The design pattern I implemented was called memoization, it basically means taking the output of a function and mapping it to a specific input so if the same input happens, it doesn't need to calculate anything again. This was done in the catscript type file, and the code below that relates to it will be highlighted in yellow.

```
package edu.montana.csci.csci468.parser;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Objects;

public class CatscriptType {

    public static final CatscriptType INT = new CatscriptType("int",
Integer.class);
    public static final CatscriptType STRING = new CatscriptType("string",
String.class);
    public static final CatscriptType BOOLEAN = new CatscriptType("bool",
Boolean.class);
    public static final CatscriptType OBJECT = new CatscriptType("object",
Object.class);
    public static final CatscriptType NULL = new CatscriptType("null",
Object.class);
    public static final CatscriptType VOID = new CatscriptType("void",
Object.class);
```

```

private static HashMap<CatscriptType, CatscriptType> mp = new
HashMap<CatscriptType, CatscriptType>();

private final String name;
private final Class javaClass;

public CatscriptType(String name, Class javaClass) {
    this.name = name;
    this.javaClass = javaClass;
}

public boolean isAssignableFrom(CatscriptType type) {
    if (type == VOID) {
        return false;
    } else if (type == NULL) {
        return true;
    } else if (this.javaClass.isAssignableFrom(type.javaClass)) {
        return true;
    }
    return false;
}

public static CatscriptType getListType(CatscriptType type) {
    if(!mp.containsKey(type)){
        return mp.get(type);
    } else{
        ListType lt = new ListType(type);
        mp.put(type, lt);
        return lt;
    }
}

}

@Override
public String toString() {
    return name;
}

```

```

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    CatscriptType that = (CatscriptType) o;
    return Objects.equals(name, that.name);
}

@Override
public int hashCode() {
    return Objects.hash(name);
}

public Class getJavaType() {
    return javaClass;
}

public static class ListType extends CatscriptType {
    private final CatscriptType componentType;
    public ListType(CatscriptType componentType) {
        super("list<" + componentType.toString() + ">", List.class);
        this.componentType = componentType;
    }

    @Override
    public boolean isAssignableFrom(CatscriptType type) {
        if (type == NULL) {
            return true;
        } else if (type instanceof ListType) {
            ListType otherList = (ListType) type;
            return
this.componentType.isAssignableFrom(otherList.componentType);
        }
        return false;
    }

    public CatscriptType getComponentType() {
        return componentType;
    }
}

```

```

        @Override
        public String toString() {
            return super.toString() + "<" + componentType.toString() +
">";
        }
    }
}

```

I chose to use memoization primarily because we were asked to do it. But the effect it has is pretty nice. Instead of running a bunch of code, it just checks with a hashmap and then answers if it can, which makes repeated simple code a lot easier to handle.

Technical writing:

Catscript Guide

Catscript is a language developed in CSCI 468 at Montana State University. Its name derives from our mascot, the Bobcat.

Introduction

Catscript is a simple scripting language. Here is an example:

```

var x = "foo"
print(x)

```

The above script will output, as expected

```

foo

```

CatScript can be much more complex, supporting functions, if statements, etc.

```

function fib(x : int) : int {
    if (x == 0) {
        return 0
    } else if (x == 1) {
        return 1
    }
}

```

```
    return fib(x-1) + fib(x-2)
}

print(fib(10))
```

which should print

55

Features

Basic Expressions

Expressions are elements of the language that simply evaluate to some sort of typed value, such as an integer, list, string, null, etc.

Additive Expression

The additive expression takes the form of `a + b` or `a - b`. If both operands are integers, the numerical value of the expression is directly computed, representing either their sum or difference. If either of the operands are a string, string concatenation will instead be used.

Factor Expression

The factor expression takes the form of `a * b` or `a / b`. Both operands must be integers, and the numerical value of the expression will be directly computed. In the case of division, regular integer division is used. Factor expressions have a higher precedence than additive expressions, so the following are equivalent

```
a + b * c
a + (b * c)
```

Unary Expressions

The unary expression takes the form of `-a` or `!a`. If a negative is used with an integer operand, the integer's value will be negated. If a bang is used with a boolean operand, the boolean's value will be negated. Unary expressions have a higher precedence than factor expressions, so the following are equivalent

```
-a + b * -c
(-a) + (b * (-c))
```

Equality Expressions

The equality expression takes the form of `a == b` or `a != b`. It compares two operands and returns either `true` or `false` depending on their value. `==` will check if the values are equivalent, whereas `!=` checks if the values are different. Equality expressions have the lowest precedence.

Comparison Expressions

The comparison expression takes the form of `a <= b`, `a >= b`, `a < b`, or `a > b`. It compares two operands and returns `true` or `false` depending on the comparison of the numerical values of the operands. Comparison expressions have a higher precedence than equality expressions, but lower than additive expressions.

Parenthesized Expressions

The parenthesized expression takes the form of `(a)`. Parenthesized expressions force their operands to be evaluated first, which can be useful for grouping statements, such as

```
a * (b + c)
```

where the value of `b + c` will be evaluated, then multiplied by `a`.

Identifier Expressions

Identifier expressions refer to a variable name and will simply evaluate to that variable's value. Variables are discussed later.

Literal Expressions

Literal expressions include integers, such as `5`, `7`, `10`; booleans, such as `true`, `false`; lists, which take the form of `[a, b, c, d]`; strings, such as `"hello"`, `"jacob"`; or null, denoted simply using `null`.

Statements

Statements are elements that have some side effect when evaluated. That is, statements perform an action.

Variable Statements

Variable statements take the form of `var x = value`, `var y : type = value`, etc. and declare a new variable with the specified value and type. Variables have static typing, which can either be inferred (such as in the first example) or explicitly stated (such as in the second example).

Assignment Statements

Assignment statements take the form of `x = value`, and modify the value at an existing variable. Since variables are statically typed, the new value must be a compatible type with the previous value.

If Statements

If statements take the form of

```
if (a) {  
    ...  
} else if (b) {  
    ...  
} ...
```

```
} else {  
    ...  
}
```

and provide a way to branch logic. The operand expression is evaluated, and when `true` the statements within the `if` clause will be run. When `false`, the statements within the `else` clause will be run, possibly branching to more `if` statements.

For Statements

For statements take the form of

```
for (x in a) {  
    ...  
}
```

and provide a way to iterate over lists. The operand expression must be some list type, and the identifier will take on the component type of the provided list. The statements within the clause will run once for each element in the list, with the identifier's value set to the corresponding element of the list.

Print Statements

Print statements take the form of `print(a)` and simply print the value of the operand expression to the console.

Function Statements

Function definition statements take the form of

```
function fnName(x : type, y : type, ...) : type {  
    ...  
}
```

and define a subroutine of logic that can be called via a function call such as `fnName(value1, value2, ...)`. Functions can take in any number of parameters, that can either explicitly specify or infer their associated types. The statements within the function clause will be run using the parameter values as local variables. Functions can also return a type using a return statement

Return Statements

Return statements must exist within a function definition statement, and take the form of `return value`. The type of value must match the return type of the function, or be empty if no value should be returned.

Types

Catscript supports the following types:

```
int
bool
string
object
list<type>
null
void
```

The `null` type is only used for the `null` literal, and can be assigned to any other type. The `void` type is only used to denote a function returns no value, and cannot be assigned from any other type.

Parsing and Compilation

Catscript can either be evaluated at runtime using a Tree-Walk Interpreter, or can be compiled to Java Bytecode to be run on the JVM. Parsing is achieved using recursive descent. The grammar of Catscript is shown below:

```
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 } ']';

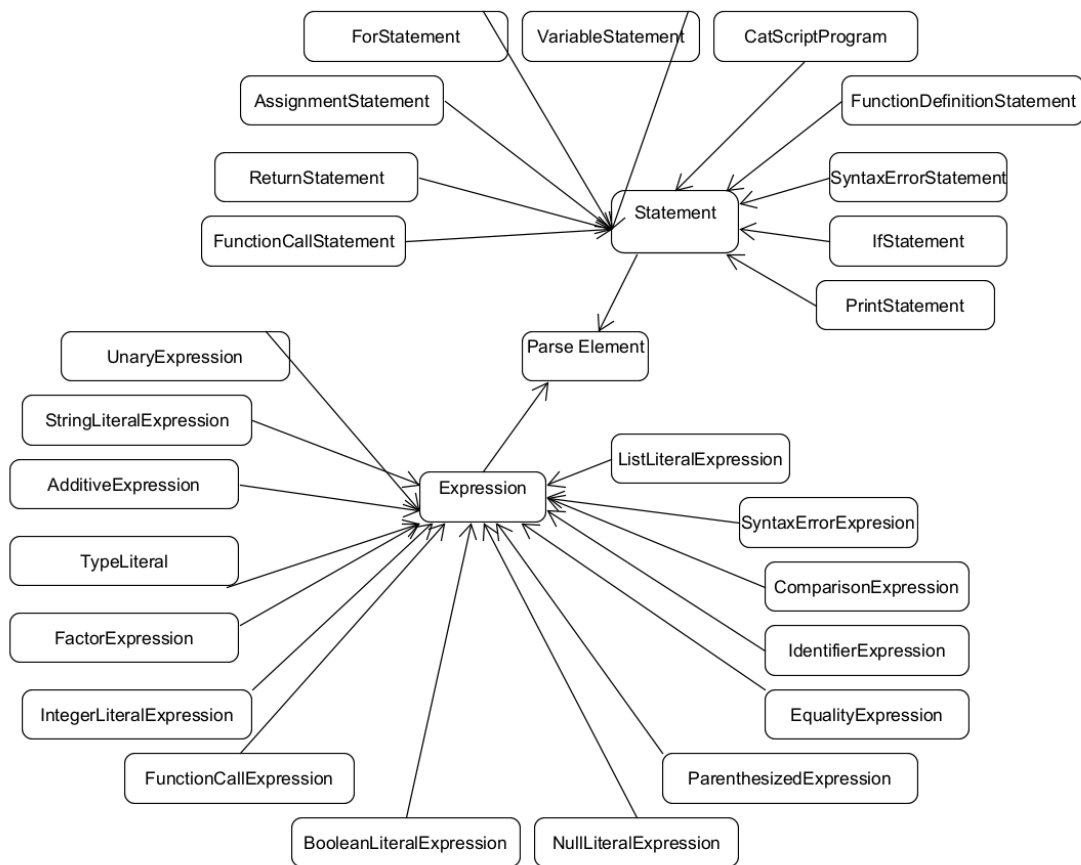
function_call = IDENTIFIER, '(', argument_list , ')'

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

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

```

UML:



Design trade offs

We used a recursive descent model. It was easy to implement, but likely will not result in code that runs as fast as possible.

Software dev life cycle

The model we used for the development was test driven development, It's a simple model that relies on getting tests to pass by set deadlines. It made meeting goals very achievable, but in testing at the end we actually found a case where the built in testing was insufficient. I guess that's a pitfall. It also lent itself to procrastination as it's easy to put off a deadline like this until it's right on you.