



Compiler Capstone Portfolio

PREPARED FOR

Carson Gross

Montana State School of Computing

PREPARED BY

Ryan Krauss

-01975661

1. Program

A copy of the compiler program is included in the /capstone/portfolio repository on Github.

2. Teamwork

John and I teamed up to provide each other with technical documentation as well as tests to check the functionality of our compilers.

3. Design Pattern

One of the design patterns we used is called memoization. Memoization is a technique used to speed up compiling by caching the results of an expensive function call and is common in recursive descent parsing. Memoization uses a hashmap to keep track of the inputs and outputs of functions to reference instead of performing the function again. In our case we are using it to track the List types we have already created. I have included the relevant code below.

```
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;
}
```

4. Technical Documentation

Catscript Guide

Introduction

Catscript is a statically typed scripting language. Here is an example:

```
var x = "Hello World"  
print(x)
```

Output : Hello World

Features

Identifier Expressions

Objects of any type can be assigned to identifier variables:

```
var x = 1  
print(x)
```

Integer Literal Expressions

Basic integer type

```
int x = 1
```

String Literal Expressions

Basic string type

```
string x = "Hello"
```

Boolean Literal Expressions

Basic boolean type

```
bool x = true
bool y = false
```

List Literal Expressions

Lists are static and can contain objects of any type:

```
list x = [1, 2, 3]
list y = ["hi", "there"]
list z = [true, false]
```

Null Literal Expressions

Basic null type:

```
var x = null
```

Additive Expressions

Integers can be added and subtracted, and Strings can be concatenated:

```
var x = 1 + 1
var y = 1 - 1
string s = "Hello" + " World"
```

Factor Expressions

Integers have multiplication and division:

```
var x = 1 * 1
var y = 1 / 1
```

Comparison Expressions

Integers values can be compared:

```
bool x = 1 > 2
bool y = 1 >= 1
bool z = 1 < 2
```

```
bool a = 1 <= 1
```

Equality Expressions

All types can be compared for equality:

```
bool x = (1 == 1)
bool y = ("Hi" == "Hi")
```

Unary Expressions

Integers can be negative

```
int x = -1
```

Parenthesized Expressions

Any expression can be parenthesized:

```
int x = (-1 + 3) / 2
```

Function Call Expressions

Defined functions can be called with parameters:

```
function add(x, y) {
    return x + y
}
var x = add(1, 1)
```

Assignment Statement

Values can be assigned to variables

```
var x = 1
```

For Statement

For statements enable iteration through lists:

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

Functional Call, Function Definition and Return Statements

Functions can be defined and then called with parameters. Return type and parameter type can be optionally defined:

```
function add(x, y) {  
    return x + y  
}  
var z = add(1, 1)  
  
function subtract(x : int, y : int) : int {  
    return x - y  
}  
int z = subtract(1, 1)
```

If Statement

If/else statements allow branching based on boolean expression value:

```
int x = 5  
if(x > 0){  
    print("x is positive")  
} else {  
    print("x is negative")  
}
```

Print Statement

Print statements display expression in console

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

Output:

```
1  
2  
3
```

Variable Statement

Variable statements assign type based on implied type:

```
var x = 1
```

```

package edu.montana.csci.csci468.eval;

import edu.montana.csci.csci468.CatscriptTestBase;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class PartnerTests extends CatscriptTestBase {
    @Test
    void intToStringConcatenation() {
        assertEquals("10ten\n", executeProgram("var x = 10\n" +
            "var y = \"ten\" +\n" +
            "print(x+y)"));
    }

    @Test
    void functionReturnsnull() {
        assertEquals("null\n", executeProgram("var x = 10\n" +
            "function foo() {return null}\n" +
            "foo()\n" +
            "print(foo())"));
    }

    @Test
    void stringsCanContainKeywords() {
        assertEquals("print\n", executeProgram("var x = \"print\"\n" +
            "print(x)"));
        assertEquals("function\n", executeProgram("var x =\n" +
            "\"function\"\n" +
            "print(x)"));
        assertEquals("int\n", executeProgram("var x = \"int\"\n" +
            "print(x)"));
    }
}

```

5. UML Diagrams

Included in the portfolio directory is “expressionClassDiagram.pdf”, the UML diagram of the different expression classes, the parent expression class, as well as the ParseElement class. This diagram gives a detailed look of what is contained with each class and a glimpse into how the Catscript compiler is ordered. In it you see each class contains their own execute, transpile, and compile methods to override the parent functions. Part of the reason I chose this diagram was to highlight the design choice used to not use a visitor pattern, but a quasi factory pattern, letting each type of expression class decide what to do with these methods.

6. Design Trade Offs

Our main design trade off was the use of recursive descent parsing instead of something like a parser generator. The advantages of using recursive descent parsing are the process being simpler as well as gleaning a better understanding of parsing and code generation. The main disadvantage is it takes more code to implement.

Another design trade off is our lack of using the visitor pattern, or alternatively the compartmentalization of every expression and statement to contain their own execute, compile, and transpile methods. This unusual method was implemented to give us a simpler understanding of parse trees and recursive descent.

7. Software Development Lifecycle

For our lifecycle model, we used test driven development. We were given many tests at the beginning and coded against those tests in order to get them to pass. I enjoyed this method of development because it gave us a clear cut goal and an obvious show of success. Having the tests first allowed me to at least have a starting off point when I wasn't sure where to go.