Logan Ladd's Capstone Portfolio

CSCI 468 : Compilers
Spring 2021

Logan Ladd | Nick Zumpano

Section 1 - Program

Please see source.zip.

Section 2 - Teamwork

The team breakdown for our project is relatively straightforward. As we are a duo, partner 1 was primarily responsible for the main design and implementation of the compiler. This was the majority of the work for the project, as it required many hours of coding and debugging.

Partner 2 was responsible for two things. First, partner 2 wrote the following tests such as these to test the functionality of partner 1's implementation:

```
@Test
public void parseForIf() {
   assertEquals("1\n2\n4\n3\n", executeProgram("for (x in [1, 2, 3]) {\n" +
         "if (x == 3){\n" +
         "   print(4)\n" +
         "}\n" +
         "print(x)\n" +
         "}"
   ));
}
@Test
public void recursiveFunctionWorksProperly() {
      assertEquals("0\n1\n2\n3\n4\n5\n6\n7\n8\n9\n10\n", executeProgram(
            "function foo(x : int) {\n" +
                  "print(x)" +
                  "if(x < 10) {" +
                  "  foo(x + 1)" +
                  "}" +
                  "}\n" +
                  "foo(0)"
      ));
}
@Test
public void parseNestedUnaryExpression() {
   UnaryExpression expr = parseExpression("not not not true");
   assertEquals(true, expr.isNot());
   assertTrue(expr.getRightHandSide() instanceof UnaryExpression);
}
```

}

These tests were then applied to partner 1's implementation of Catscript, and they tested the functionality of the statement parsing, execution, and expression parsing respectively. This helped partner 1 ensure that his implementation of Catscript would not run into advanced issues, such as if an if statement inside of a for statement executes properly, which could be considered a relatively complex parsing. After partner 1 would run the tests, he would attempt to fix the implementation of the recursive descent parser if errors were present, otherwise the tests would pass and no action would need to be taken. Please see section 7 on testing methodology to understand the purpose of the testing more in-depth.

The second thing that partner 2 was responsible for was writing the documentation for the Catscript language in section 4 of this portfolio. This creates a pleasant higher-level explanation of Catscript similar to that of the documentation of popular coding languages such as Python so that people that have never touched Catscript before but have a general knowledge of coding can use Catscript. Please see this contribution below. This is the breakdown of each partner's contribution to the project.

Section 3 - Design Pattern

One design pattern that we used is called memoization. Memoization is defined as a technique to optimize computer programs by storing the results of expensive function calls and returning a cached result when a similar input occurs again.

This makes it so that when we call a function in Catscript, let's say foo(2), and we call the function a few different times, foo(3), foo(5), and then come back and call foo(2) again, it will not be executed again, the result is already stored in the cache. This obviously frees up execution time and therefore makes the program more efficient. Let's look at a simple implementation in Catscript:

```
static Map<CatscriptType, CatscriptType> CACHE = new
HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
   CatscriptType potentialMatch = CACHE.get(type);
   if (potentialMatch != null){
       return potentialMatch;
   } else {
       ListType listType = new ListType(type);
       CACHE.put(type, listType);
       return listType;
   }
}
```

First, a map is declared to store the catscript type in getListType. Then when getListType is called, the CatscriptType checks against the cache to see if the type is already in the cache. If it is, the match is returned and no further execution is required. If the list type is not in the cache, then the new listType is created and returned.

This design pattern is helpful as it brings down the execution time of Catscript, and this sort of pattern is something that can be implemented whenever a function is called that can apply variables and types to a map. This is obviously extremely useful as variable type assignment and declaration is an integral part of a compiler.

Section 4 - Technical Writing

The following was contributed by Nick Zumpano.

Catscript

Catscript is a statically typed programming language that has many features reminiscent of a modern programming language. Catscript programs consist of a series of statements and expressions that will be executed sequentially using a recursive descent language.

Type System

Catscript contains 4 basic data type structures as well as lists and objects

```
int: refers to 32 bit integers
string: refers to a string type that is stored dynamically and is terminated whit ""
bool: refers to data that is either true or false
null: the empty null type
list: - a list type that can store multiple data points of the same type.
object: - an object type that can store anything. Objects can be cast to
        different data types based upon CatScripts hierarchy
```

The data hierarchy has null type at the base, then string, int and bool above it, then finally object above those.

Statements

Catscript contains 8 Statements. Statements change the flow of the program but do not return a value of their own.

Assignment Statement

IDENTIFIER, '=', expression;

```
x = 10
```

For Statement

for_statement = 'for', '(', IDENTIFIER, 'in', expression ')', '{', { statement }, '}';

```
for (x in ["a","b","c"]){
    print(x)
}
```

Function Call Statement

function_call_statement = function_cal

```
foo(x)
```

Function Definition Statement

function_declaration = 'function', IDENTIFIER, '(', parameter_list, ')' + [':' + type_expression] + "{" + { function_body_statement } + "}";

```
function foo (x:int, y){
    z = x + y;
    print(y);
    return z;
}
```

If Statement

if_statement = 'if', '(', expression, ')', '{', { statement }, '}' ['else', (if_statement | '{', { statement }, '}')];

```
if(x = 21){
    print("Excellent, Twenty One!");
}else{
    print("Other numbers just aren't that good...");
}
```

Print Statement

print_statement = 'print', '(', expression, ')'

```
print("Hello World");
```

Return Statement

return_statement = 'return' [, expression];

```
return x;
```

Variable Statement

variable_statement = 'var', IDENTIFIER, [':', type_expression,] '=', expression;

```
var x : string = "This is a variable, isn't it?"
```

Expressions

In CatScript there are 6 expressions. Expressions differ from the statement in the fact that they will always return a value CatScript uses recursive descent to parse each expression, so there is a ladder for what kind of expression an expression will be. This is the order of expression hierarchy:

1. Equality Expression
2. Comparison Expression
3. Additive Expression
4. Factor Expression
5. Unary Expression
6. Primary Expression

Equality Expression

equality_expression = comparison_expression { ("!=" | "==") comparison_expression };

```
1 == 1;
```

Comparison Expression

comparison_expression = additive_expression { (">" | ">=" | "<" | "<=") additive_expression };

```
4 >= 3;
```

Additive Expression

additive_expression = factor_expression { ("+" | "-") factor_expression };

```
2 + 9;
```

Factor Expression

factor_expression = unary_expression { ("/" | "*") unary_expression };

```
3/4;
```

Unary Expression

unary_expression = ("not" | "-") unary_expression | primary_expression;

```
-1
```

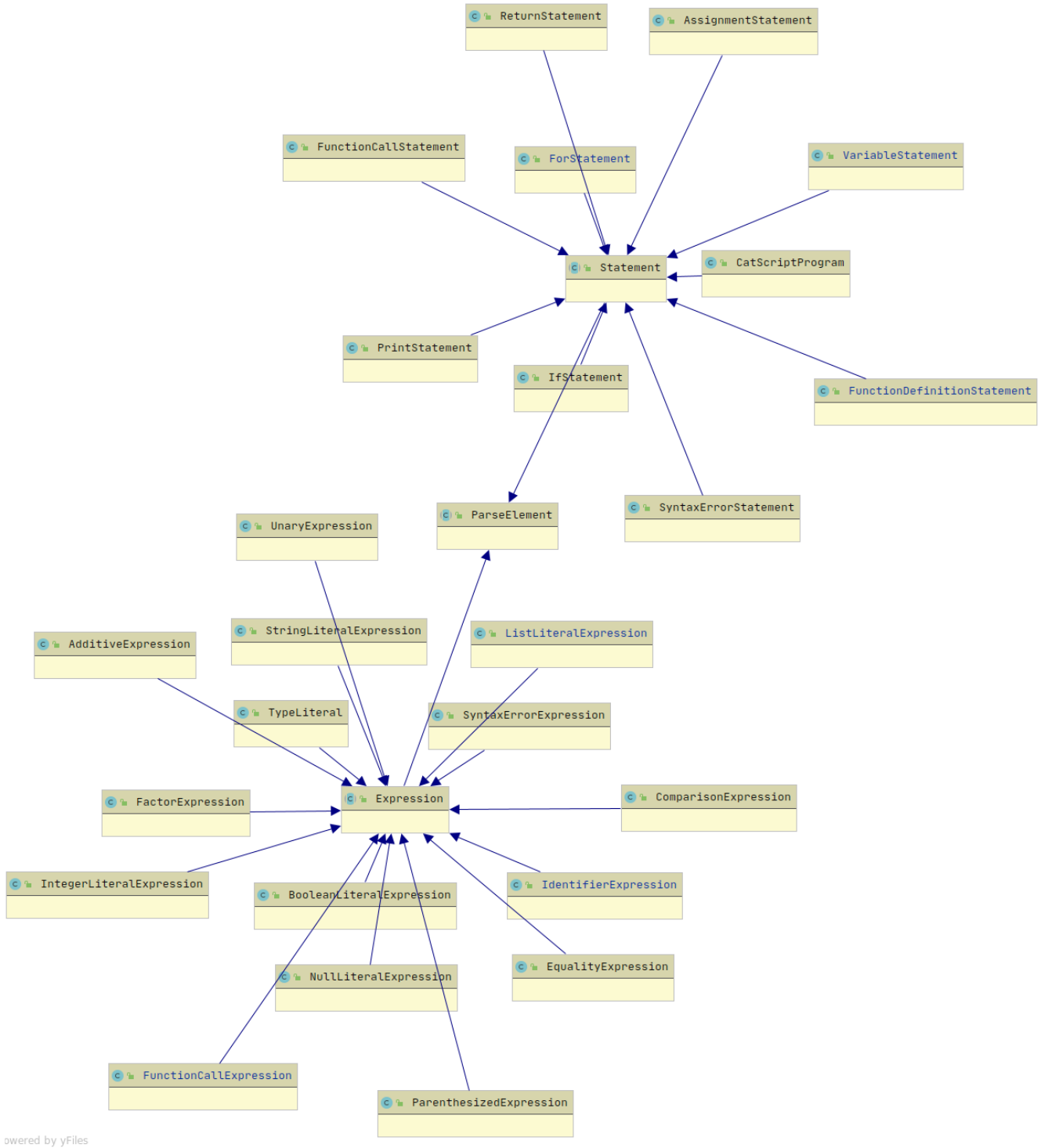
Primary Expression

primary_expression = IDENTIFIER | STRING | INTEGER | "true" | "false" | "null"| list_literal | function_call | "(", expression, ")"

```
"Hello World";
```

Section 5 - UML

Observe the following UML diagram:



This UML diagram is a diagram of the parsing of either an expression or statement in Catscript grammar. This UML diagram was created to outline the design of the written recursive descent parser and to more easily understand what different types of statements and expressions belong where parser. Take a look at the following line:

return x +1;

This is a combination of a statement and expression that needs to be parsed separately. If we break this down in accordance to the UML diagram, it is a ReturnStatement, that requires a 'return' token and expression afterward, which in this case is an AdditiveExpression. As can be seen, these two underlying objects are specified to be in the statement section and expression section of the UML diagram respectively, and an important distinction of the two different types would now be parsed properly so that the return statement can be executed. This UML diagram was extremely important to creating a successful recursive descent parser which is the largest part of our CatScript implementation.

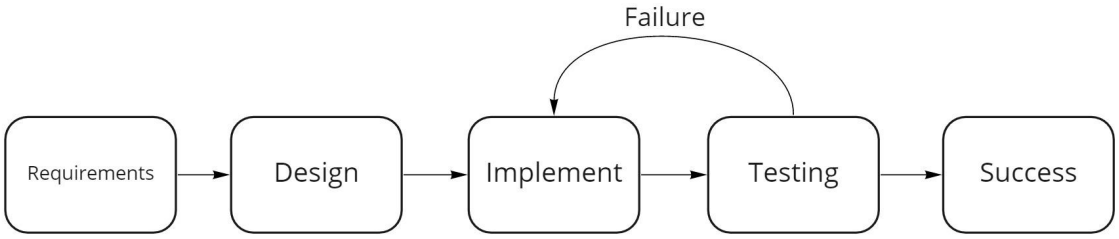
Section 6 - Design Trade-offs

The design trade-off decision I will outline in this section is the decision to use a recursive descent parser rather than a parser generator or other type of parser. First, there are a couple advantages to using a recursive descent parser. The first advantage of using a recursive descent parser is that it is always the easiest form of parser you can create using a base language's standard library and tools. In our case, this was Java, and we were easily able to implement recursive descent in java using the standard library and tools. Without recursive descent you are likely required to install a parser generator tool that does the job for you, which is the main tradeoff that will be discussed in a moment.

There are a few different tradeoffs now to be discussed. First, a pre-built parser generator tool would likely result in a higher quality compiler if the time and effort is put into learning one of these tools such as JLex or JFlex, however the parser can not be written all in a single language and requires external tools. The tradeoff also comes from the time consumed to learn a new tool in a new language, rather than just directly coding the recursive descent.

Another trade-off is control we have over our compiler with recursive descent. Certain decisions about scope, syntax, and other certainties can be completely controlled with our own written recursive descent parser. It may take more time to go deeper and code these certainties of our parser, but the control trade-off received was beneficial for our testing and compiling purposes. Lastly, recursive descent perhaps may not be as efficient in time complexity as a parser generator tool, however, more research would have to be conducted to see such differences in a variety of tests.

Section 7 - Software Development Life Cycle Model



First, **requirements** were set through the successful ideal functionality of CatScript. This is relatively straightforward as the goal of the project was to create a successful compiler. At each respective step of tokenizing, parsing, evaluating, etc., We built requirements based upon the methodology of building a compiler.

Next, The compiler itself was **designed** so that Catscript could be executed in a way that is logical and efficient. The design is largely a framework that has numerous java classes for data types, evaluating, executing, tokenizing, parsing, and so on. Most classes have helper functions as well to successfully execute the different steps of catscript. This is the design step of our life cycle model.

After requirements and the design is set, it is time to **implement** the compiler. This was the majority of the work for Nick and I, and it was done through the use of the IntelliJ Java IDE. We were responsible for implementing the actual tokenizing, parsing, evaluating, and execution of the code within the built design so that Catscript would run successfully. Observe the following implementation of number tokenization:

```
private boolean scanNumber() {
   if(isDigit(peek())) {
       int start = position;
       while (isDigit(peek())) {
```

```
            takeChar();
        }
        tokenList.addToken(INTEGER, src.substring(start,
position), start, position, line, lineOffset);
        return true;
    } else {
        return false;
    }
}
```

This is a relatively simple example of implementation, it should be apparent that the tokenizer checks the current token with peek(), takes the digits until there are no more integers, then adds the token to the tokenizer and returns true. There are over 1000 lines of implemented code in the compiler.

Our project was primarily developed through the use of **tests**. The tests were produced by the JUnit test framework and written to ensure the functionality of each step of the compiler. Observe the following test as an example;

```
@Test
public void unterminatedStrings(){
   assertTokensAre("\"asdf", ERROR, EOF);
   assertTokensAre("\"asdf\"\"asdf", STRING, ERROR, EOF);
   assertTokensAre("\"asdf \"asdf\"", STRING, IDENTIFIER, ERROR,
EOF);
}
```

This particular test ensures that when the compiler is breaking down strings into tokens, the strings have an end quote to terminate the string. Many of the tests are like this to ensure the functionality of the compiler. If a test failed, we were required to go back to the implementation step to fix the error. In this way we were able to develop the software and ensure the functionality of the compiler. This was the most helpful step in our model.

Overall, these four steps in our software development life cycle model worked well, the model helped us to successfully create a compiler through high level abstracting down to the most particular tests. The model did not hinder our ability to create the compiler, however at times it was difficult to understand the errors in the implementation when a test fails. This is our Software Design Life Cycle Model.