Montana State University
Gianforte School of Computing

CSCI 468 Compilers

Jack Tetrault

May 2022

Table of Contents

Section 1: Program……………………………………………………... 2
Section 2: Teamwork…………………………………………………… 2
Estimated Contributions
Section 3: Design Pattern……………………………………………. 3
Section 4: Technical Writing……………………………………..… 3
Catscript Typesystem…………………………………………… 3
Tokenization……………………………………………………….. 4
Parsing………………………………………………………...… 5
Evaluation………………………………………………………. 6
Section 5: UML……………………………………………………..… 7
Section 6: Design Trade-Offs……………………………………….. 8
Section 7: Senior Development Cycle Model…………………………… 8

Section 1: Program

The following file: source.zip is located in the portfolio directory.

Section 2: Teamwork

The design aspect of this project was completed by me, as I am the primary developer. The teamwork aspect of this project, however, was incorporated through the implementation of additional testing and documentation. The workload was split into the following, approximately 90% of time and effort was dedicated towards the design aspect of the program, completed by me, the primary developer, and the remaining 10% are accredited to my partner, for providing me with three additional tests as well as providing documentation for my code (Section 4). The approximation of work hours would be about 90 hours compared to about 10 hours. This division of workload also applies conversely to my partner. The following tests were provided to me by team member 2. They consist of a token, parsing, and an evaluation test.

```java
//test #1
@Test
public void tokenTests() {
   assertTokensAre("10 + 12 - 12", TokenType.INTEGER, TokenType.PLUS,
TokenType.INTEGER, TokenType.MINUS, TokenType.INTEGER, TokenType.EOF);
   assertTokensAre("return true return false", TokenType.RETURN,
TokenType.TRUE, TokenType.RETURN, TokenType.FALSE, TokenType.EOF);
   assertTokensAre("{[()]}", TokenType.LEFT_BRACE, TokenType.LEFT_BRACKET,
TokenType.LEFT_PAREN, TokenType.RIGHT_PAREN,
           TokenType.RIGHT_BRACKET, TokenType.RIGHT_BRACE, TokenType.EOF);
}
//test #2
@Test
public void parsingTest() {
   VariableStatement expr = parseStatement("var dog : string = bark");
   assertNotNull(expr);
   assertEquals("dog", expr.getVariableName());
   assertEquals(CatscriptType.BOOLEAN, expr.getExplicitType());
   assertTrue(expr.getExpression() instanceof StringLiteralExpression);
}
//test #3
@Test
public void evalTests() {
   assertEquals(false, evaluateExpression("5 > 15"));
   assertEquals(true, evaluateExpression("2 < 3"));
   assertEquals(false, evaluateExpression("true != true"));
   assertEquals(7, evaluateExpression("7"));
}
```

Section 3: Design Pattern

Memoization is a programming technique used to optimize expensive recurring function calls. Using this optimization pattern, we store the output of an expensive function call, and when the function is called again, the previous result is already stored. This prevents the program from needing to re-compute everything, and rather just uses what it already knows.

```
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;
   }
}
```

In the memoization function (shown above) the program will treat any function passed in as a CACHE. When the memoization function is called, it looks to see if the function passed in already exists in a list of functions. If the function is found in the list, then the program knows it has already done this computation and simply returns the stored result. If the function is not already in the list, the program creates a copy of the said function call and adds it to the list.

Section 4: Technical Writing

Catscript Typesystem:

- int - integers
- bool - a Boolean value
- object - any object
- null - type of null value
- void - no type
- list<int> - list of integers
- list<object> - list of objects
- list<list<int>> - list of lists holding ints

Tokenization:

Our parser, using recursive descent, starts with tokenization. This was also the first aspect that we were tasked to complete. Tokenization begins with the tokenize() function, which gets rid of unnecessary whitespace. After that is complete, the scanToken() function associates a type to the token.

```
private void tokenize() {
   consumeWhitespace();
   while (!tokenizationEnd()) {
       scanToken();
       consumeWhitespace();
   }
   tokenList.addToken(EOF, "<EOF>", postion, postion, line, lineOffset);
}

private void scanToken() {
   if(scanNumber()) {
       return;
   }
   if(scanString()) {
       return;
   }
   if(scanIdentifier()) {
       return;
   }
   scanSyntax();
}
```

This function scans the token until a matching type is returned. This is an example of how recursive descent parsing works throughout the program. Depending on the type returned, another function is called to tokenize that specific type. If the scanToken() returns scanNumber (shown below), the function will first look to verify that the token type is an integer, then it checks each integer token, if those tokens match, they are added to a list of tokens that will eventually be sent to the parser.

```
private boolean scanNumber() {
   if(isDigit(peek())) {
       int start = postion;
       while (isDigit(peek())) {
           takeChar();
       }
       tokenList.addToken(INTEGER, src.substring(start, postion), start, postion, line,
lineOffset);
       return true;
   }
   else {
       return false;
```

```
    }
}
```

Parsing:

Parsing also plays a vital role in this program. The parser essentially takes in a list fed from the tokenizer and veriefies if the given grammar fits within Catscript's. Recursive Decsent was used to construct the parser, which ultimately resulted in the structure looking very similar to the Catscript's recognizable grammar. One significant function used for parsing is parseStatement() (shown below). This function parses out the correct type and continues to descend into more specificity.

```
private Statement parseStatement() {
    Statement printStmt = parsePrintStatement();
    if (printStmt != null) {
        return printStmt;
    } else if (tokens.match(VAR)) {
        return parseVariableStatement();
    } else if (tokens.match(IF)) {
        return parseIfStatement();
    } else if (tokens.match(FOR)) {
        return parseForStatement();
    } else if (tokens.match(IDENTIFIER)) {
        Token ident = tokens.consumeToken();
        if (tokens.match(EQUAL)) {
            return parseAssignmentStatement(ident); //assignment
        } else {
            return parseFunctionCallStatement(ident); //function call
        }
    }
    if (currentFunctionDefinition != null) {
        return parseReturnStatement();
    }
    return new SyntaxErrorStatement(tokens.consumeToken());
}
```

Another very significant parsing function is parseFunction(). Similar to the type scanning that was done during tokenization, the function begins by locating the start of the function. After taking in the function token, the program sets the according function name from the type value of the function. Next, a list is created in order to store all of the function arguments, making it easier to evaluate them. After the function loops through and adds all of the statements within the list to our parameters, and we have added all the types and parameters to the function, we access the function body, which calls another function to process that said line.

Evaluation:

Literals

Literals are made up of literal values encoded into the language. Types of literals include booleans, strings, lists, and null, and their values are simply just returned when evaluated.

Parentheses

Evaluating parentheses is done by simply returning the value that the expression enclosed within '('')' evaluates to.

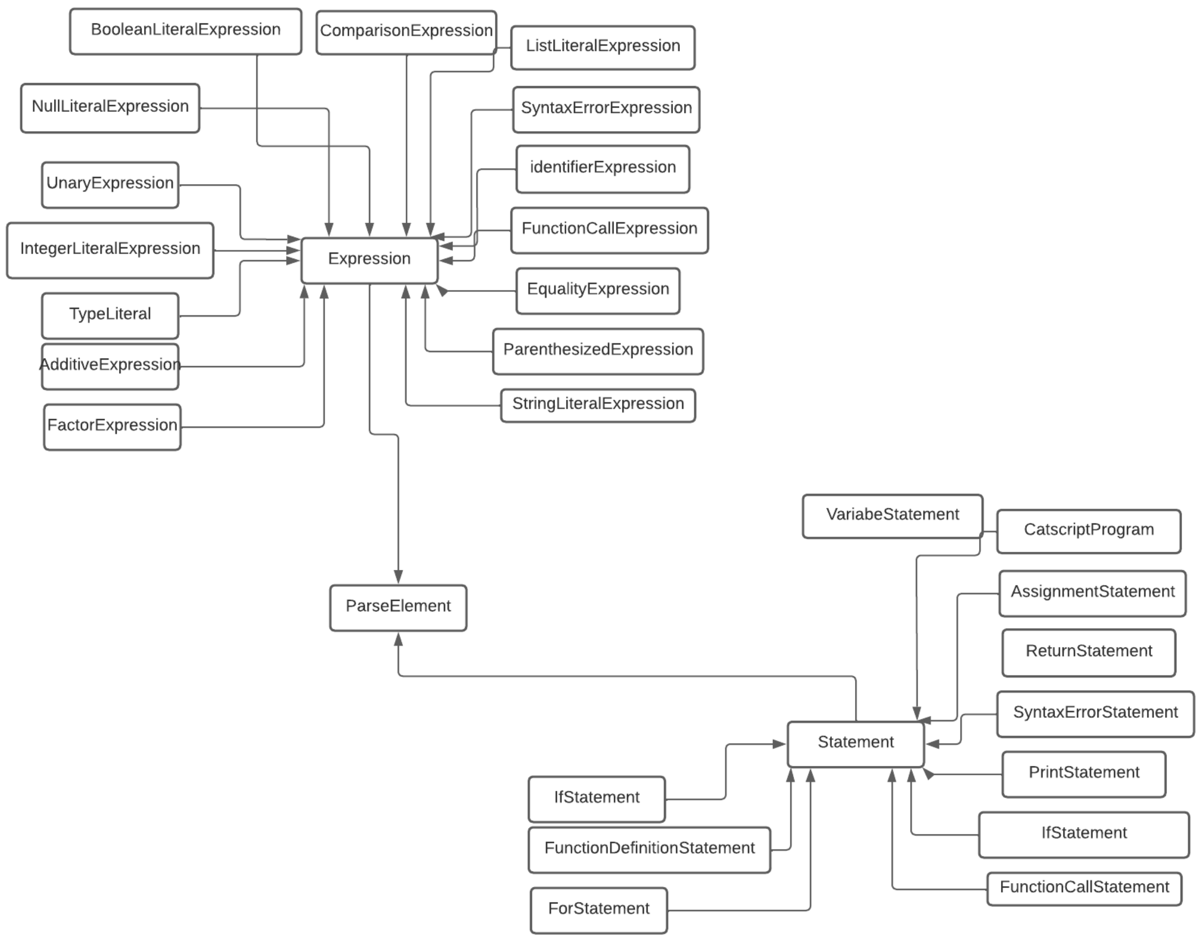
Unary

Expressions with just a single argument are evaluated by first evaluating the right-hand-side, and applying the operator to solely that value.

Binary expressions have both right-hand and a left-hand side arguments. These expressions are evaluated by first checking the type, and then concatenating the two values together to be returned.

Section 5: UML

For this project, all of the classes and overall structure of the program was created prior by the professor. As a result, a diagram complete with all the class details was not necessary. Instead, I have included the basic class diagram depicting the relationship between Expressions, Statements, and Parsing. The extression types have an “is a” relationship with Expression and the statement types have also have an “is a “ relationship with Statement. This is because the specific expression and statement types are still identified as Expressions and Statements accordingly.



Section 6: Design Trade-offs

The most significant choice regarding design trade-offs was the decision to implement "Recursive Descent" in our parsing rather than using a "Parser Generator" approach.

A **parser generator** is made up of two parts, lexical grammar as REGEX (regular expression) and a long grammar as EBNF(Extended Braccus Naur Form). This method essentially takes in a set of rules and uses this to create a parser, similar to an Abstract Syntax Tree. Although using this parsing technique can require less code and infrastructure, it can also be more difficult to understand, thus taking more time to implement.

Recursive descent, on the other hand, is a top-down approach to parsing. This parser relies on recursive procedures, which ultimately helps the programmer gain a better understanding of parsing and working with various grammars. For each output in the grammar, there also exists a method. The significance of giving each production a corresponding method is that it allows for a clearer understanding of the recursive nature of parsers. This technique is also widely used in the industry to tokenize source code.

For this project, the parsing algorithm was chosen as recursive descent. I believe the trade-off was definitely worth it, as it seemed to be much easier to understand and see how the compiler was actually working.

Section 7: Software Development Life Cycle Model

The software development life cycle model used in this program was test-driven development. Test-driven development is done by writing the test cases for your program before a full draft of the code has been completed. This style of programming essentially means the programmer codes, tests, and designs concurrently. As a result, developers have the ability to follow their progress through the evaluation of the pre-constructed tests.

This technique of a somewhat "checkpoint" approach to programming proved to be very beneficial to the personal completion of this project. Instead of creating and running various tests at once to verify certain aspects of development, individual tests with multiple aspects predetermined parameters can provide insight as to what is working, what still needs work, and what hasn't been started. This not only helps the programmer track their progress, but also gives them a better insight into how specific aspects of the program are performing.