Compilers Report

Spring 2021

Rania Ehsan
Abdulaziz Alghamdi

Team Work:

Given that this semester was unusual and very different from what a lot of us are used to, even though I had a partner, him and I worked on our projects on our own. The code in all mine from my project and the code on his project is entirely his. Due to the pandemic, and also with my partners roommate getting tested positive for COVID, we ended up not meeting very much at all to collab on things. The help that we had for each other is that when we were researching various aspects of the projects, we let each other know what new and helpful links we found that would help us complete the project. We had intentions of generating tests for each other, however due to time constraints and my partners roommate getting tested with COVID toward the last few weeks of the semester we ended up not being able to do that.

Design Pattern:

I used Memoization as Design Pattern for the Design Pattern. Memoization is a higher order function call that catches a different function call. What memorization can do is, it can turn slower functions into faster ones. It also saves the result of a function call after the first time in the cache. If the function is called again with same arguments, it can be found in the cache. A code snippet where I used Memoization is below. I personally used Memoization because it was taught well enough in class. I also like it due to the fact that it does have the capability to turn slower functions faster.

```
public static CatscriptType getListType(CatscriptType type) {
        return new ListType(type);
    }

    @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() + ">";
        }
    }

}
```

Introduction:

This paper documents the project that has been completed for CSCI 468. The intention of this project is for students to have a better aspect of the inner workings of a compiler. This being

a Computer Science Capstone class, there are a lot of aspects from previous classes that I used to do this project. The project walks us through the steps of creating a working compiler. I will be talking about each steps of the compiler.

What is a Compiler?

A compiler is a program that translates high level programming language to low level programming language to be executed. A compiler essentially works as a translator and along with that a lot of modern-day compilers optimizes assembly code and ensures that it runs as fast as possible on the Central Processing Unit (CPU).

There are several parts to a compiler. First there is the Scanner. What a Scanner does is, it breaks source code into tokens and runs it through a scanner generator. After that, the Tokens are parsed using a parser, the output of which is a syntax tree. We can also generate a parser using Context Free Grammar. After that Intermediate Representation is generated by doing semantic routines based on the syntax tree. This can then be run through an optimizer. And an optimizer will recognize the Intermediate Representation in a form that makes it run faster. At this point, the Intermediate Representation generates the assembly code.

What is Tokenizer?

The very first part of the project that we did was the Tokenizer. I personally found this part of the project particularly hard. What a Tokenizer does is, it converts an input code in LISP syntax into an array of Tokens. When writing a couple of tokenizers for a single token, each of the tokens will receive the code as a string and the current position is to return the length of the

token and the token as an object too with two different keys. Those keys being type and value. An example of a single character token would be as follows:

```
tokenizeCharacter = (type, value, input, current)

    (value = input[current] )? [1,{type,value}] : [0,null]

        [Function tokenizeCharacter]
```

What is a Scanner?

What a compiler does is, it breaks the input into tokens and these tokens are later taken into the parser. When this is being done, scanner also analyzes each token and evaluates if the token is in a valid programming language or not. The scanner may also try to identify variables, strings, numbers, comments, operations, or any other language construct. Matching tokens complexity is low; hence this step can be done by implementing them as a set of regular expressions. These regular expressions show what they are composed of. Regular expressions aren't enough for parsing and validating the whole program. However, they are enough to scan steps for reducing complexity.

In my code the scanner iterates through all the tokens and checks to see if the current symbol in the input will match the tokens or not. The token is then added to the token stream if they match the input. It then goes through the parser for compilation.

The hardest part I faced at this part of the project was implementing the scanSyntax() method. The scanSyntax() method took me the longest time since I had to code so that every symbol was recognized by the scanner. Because of the sheer amount of code that I had to write for this part, I kept making a lot of syntactical errors. I understood the idea behind it, but as I was

trying to rush through the steps I kept messing up the hierarchy of the symbols. I kept making a lot of syntactical errors on this portion too since the code itself looked very repetitive. A snippet of the code for this portion is as follows:

```
private void scanSyntax() {
        // TODO - implement rest of syntax scanning
        //      - implement comments
        int start = postion;
        if (matchAndConsume('+')) {
            tokenList.addToken(PLUS, "+", start, postion, line, lineOffset);
        } else if (matchAndConsume('-')) {
            tokenList.addToken(MINUS, "-", start, postion, line, lineOffset);
        } else if (matchAndConsume('*')) {
            tokenList.addToken(STAR, "*", start, postion, line, lineOffset);
        }else if (matchAndConsume('(')) {
            tokenList.addToken(LEFT_PAREN, "(", start, postion, line, lineOffset);
        }else if (matchAndConsume(')')){
            tokenList.addToken(RIGHT_PAREN, ")", start, postion, line, lineOffset);
        }else if (matchAndConsume('[')) {
            tokenList.addToken(LEFT_BRACKET, "[", start, postion, line, lineOffset);
        }else if (matchAndConsume(']')) {
            tokenList.addToken(RIGHT_BRACKET, "]", start, postion, line, lineOffset);
        }else if (matchAndConsume('{')){
            tokenList.addToken(LEFT_BRACE,"{", start, postion, line, lineOffset);
        }else if (matchAndConsume('}')){
            tokenList.addToken(RIGHT_BRACE, "}", start, postion, line, lineOffset);
        } else if (matchAndConsume('>')) {
            if (matchAndConsume('=')) {
                tokenList.addToken(GREATER_EQUAL, ">=", start, postion, line,
lineOffset);
            } else {
                tokenList.addToken(GREATER, ">", start, postion, line, lineOffset);
            }
        }else if (matchAndConsume('<')){
            if(matchAndConsume('=')) {
                tokenList.addToken(LESS_EQUAL, "<=", start, postion, line,
lineOffset);
            } else {
                tokenList.addToken(LESS, "<", start, postion, line, lineOffset);
            }
        }else if (matchAndConsume('=')){
            if (matchAndConsume('=')){
                tokenList.addToken(EQUAL_EQUAL,"==",start,postion, line, lineOffset);
            }else {
                tokenList.addToken(EQUAL, "=", start, postion, line, lineOffset);
            }
        }else if (matchAndConsume('!')) {
            if (matchAndConsume('='))
                tokenList.addToken(BANG_EQUAL, "!=", start, postion, line,
lineOffset);
        }else if (matchAndConsume('=')) {
```

```
            tokenList.addToken(EQUAL, "=", start, postion, line, lineOffset);
        }else if (matchAndConsume(',')){
            tokenList.addToken(COMMA, ",", start, postion, line, lineOffset);
        }else if (matchAndConsume('.')){
            tokenList.addToken(DOT, ".", start, postion, line, lineOffset);
        }else if(matchAndConsume(':')){
            tokenList.addToken(COLON, ":", start, postion,line,lineOffset);
//        }else if (matchAndConsume('/')){
//            tokenList.addToken(SLASH, "/", start, postion, line, lineOffset);
        } else if (matchAndConsume('/')) {
            if(matchAndConsume('/')) {
                while (peek() != '\n' && !tokenizationEnd()) {
                    takeChar();
                }
            } else {
                tokenList.addToken(SLASH, "/", start, postion, line, lineOffset);
            }
        } else {
            tokenList.addToken(ERROR, "<Unexpected Token: [" + takeChar() + "]>",
start, postion, line, lineOffset);
        }
    }
```

I used Java for doing the whole project. I personally am familiar with using Java, and our instructor had also provided us a skeleton code in Java for us to fill in the gaps.

What is a Parser?

Parser is an interpreter or a compiler that breaks data into smaller elements and easily translates into another language. What a Parser does is, it takes input as a sequence of tokens and breaks them up into interpretable components that can be used in programming. The parser checks the input data to make sure it is enough to build a Data Structure in the form of a parse tree. The parser follows a set of specific rules to do its task. During the Lexical Analysis, tokens are produced from a input string characters. This is later broken into smaller components and it forms meaningful expressions.

During Syntactical Analysis, it is checked to see if the tokens that have been generated make a meaningful expression of not. There is also Semantic Parsing where the meaning and

implications of the validated expressions are determined and whatever actions need to taken are taken. I had top-down parsing, as a result it needs to search the parse tree and find the left most derivations of the input using top-down expansion. So, they start at the start symbol that gets transformed to input symbol and it keeps parsing till all the symbols are translated and a parse tree is created. My experience from Data Structures and Algorithms specially came in handy in this part of the project. It helped me better understand parse trees.

Evaluate:

For the Evals part of the project, the evaluate() method is implemented by all Expression objects. Literal expressions literal values encoded directly in the programming language. For evaluating literals I had to return the integer value that I parsed. We had already turned string into integer value previously, at this point we just had to have it return the runtime. For parens, we had to return the values that the enclosed expression evaluates to. Some parsers don't define tree nodes for parenthesis. Instead when it parses a parenthesized expression it returns the node for the inner expression.

Symbol Table:

For this project I also had to do part of the compiler that would build a Symbol Table. For this part specifically my experience from my Data Structures and algorithms class came in handy. We earned to implement stack and we learned during that class how to push or pop items from the stack. This part was kind of difficult for me because prior to doing this, I didn't have some of the tests passing for Parser and as a result I couldn't move further unless I had fixed my Parser completely. When the Parser part of the project was fixed, I was able to identify the

grammar that I already had. Once the grammar itself is appropriately identified, a simple symbol table can be built.

What is Type Systems?

A type system is a logical system of rules that assigns a type to constructs in a programming system and enforces rules about those types. Type is an attribute of data which tells a compiler or interpreter how the programmer intends to use some data. The main goal of a type system is to reduce any chance of bugs in computer programs. It does this by defining the interfaces between different parts of a computer program. And then it checks that the parts have been connected consistently. Example of a TypeError is as follows:

```
Object x = 10;

x = “foo”;

System.out.println(x.getLength());

x = ArraysasList(1,2,3);

System.out.println(x.getLength());
```

The code above cannot be guaranteed by Java on if it is correct or not, hence it stops it from compiling. So in this case, this is cast to another type. So x is casted first to String and then length() is invoked. Then it is cast to List and size() is called. If the code were modified to put the assignment before the List before casting it to string we will get a ClassCastException at runtime. So the Java typesystem is static compile time, it also has a runtime and dynamic aspect. At runtime it has to check if object is of a given type or not. By weakening the typesystems in Java, it makes some sorts of programming a lot easier. Catscript arrays are read only, so values

cannot be added or modified in them. Keeping this constrain in mind, the list types covariant with one another. So, a type error cannot be introduced by this assignment in catscript.

What we know is that, Catscript is statically typed programming language. The types of all variables, functions and parameters are known at compile time. Catscript can also guarantee about runtime types.

Catscript:

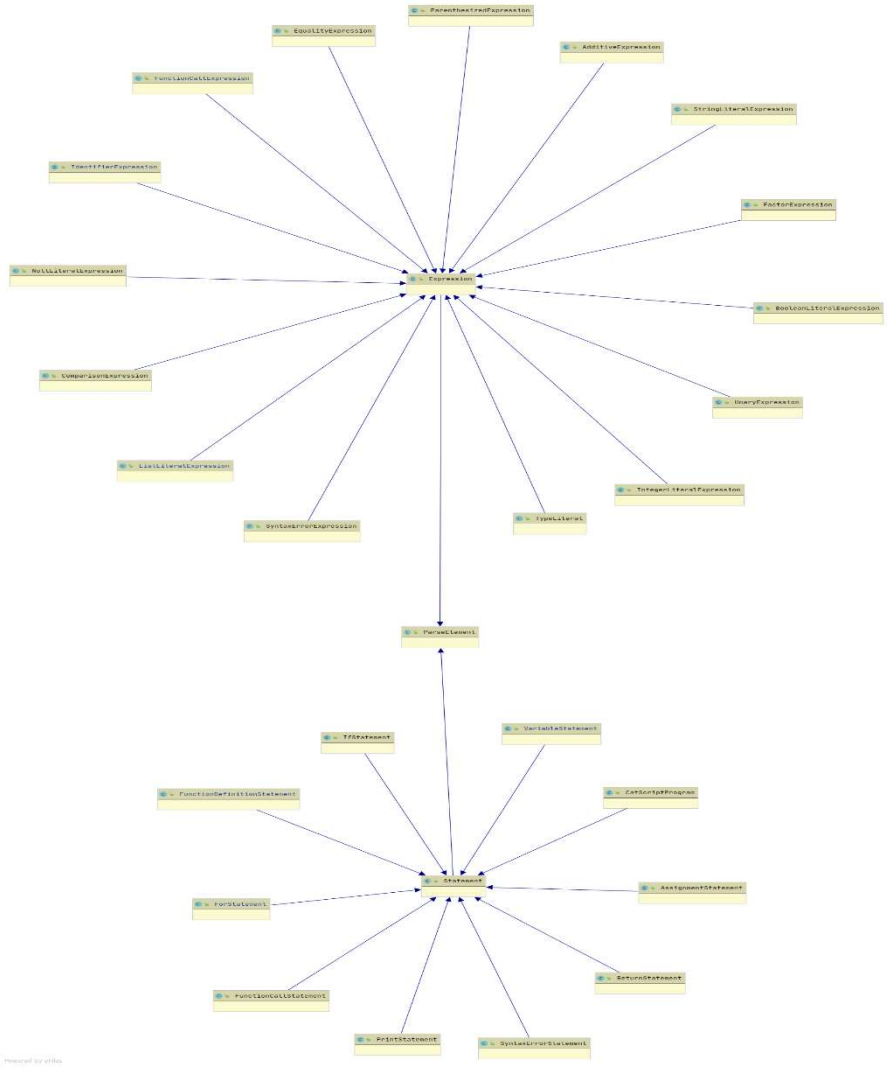
Catscript has some fun features. Those being that they are statically typed, it has list literals, local variables type interface. Java doesn't have list literals and local variable type interface. Catscript programs consist of a series of statements that gets executed sequentially. Catscript supports the standard if statement and it also supports for statement that allows iteration over list. Catscript also supports function definition. It supports some common expression types for mathematical, logical, and rational operations. Catscript has a small type system, those being int(32 bit integer), string, bool, list, null and object.

Similar to Java, catscript autobox and autounboxes the primitives. However, unlike Java there aren't corresponding reference types for primitives.

Design Trade-offs:

Generally LL parser is a lot more efficient than recursive descent. A native recursive descent will have O(k^n), n= input size in worst case scenarios. In memoization, this may be improved and the class can extend the class of grammers accepted by the parser. There is however, a trade-off. That being that LL-parsers are pretty much at most times linear.

UML:



The above UML is the one that we used.

Software Development life cycle model:

For this project the model that we used is the Test Driven Development. The TDD is a style of programming in which coding, testing and design are interwoven. We were provided with tests for the project and for us to know that our code was working was when we had the tests passing. When I had tests fail, I focused on what aspect of the code needs to be changed and

changed my code accordingly. I personally found this model to be incredibly helpful since it helped me focus on each part of the code at a time.

Acknowledgements

This is to my beloved Bobo, who stayed up with me every night that I had to stay up working on the project.

Works Cited

Sharvit, Yehonathan. “Write Your Own Compiler - Station #1: the Tokenizer.” *Klipse.tech*, 2017, blog.klipse.tech/javascript/2017/02/08/tiny-compiler-tokenizer.html.

“Parser.” *Techopedia*, www.techopedia.com/definition/3854/parser.

“Compiler Basics, Part 2: Building the Scanner.” *Visual Studio Magazine*, visualstudiomagazine.com/articles/2014/06/01/compiler-basics-part-2.aspx.

Normad, Eric. “What Is Memoization?” *LispCast*, 2019, lispcast.com/what-is-memoization/.

Blindy. “Create Symbol Table.” *Stackoverflow*, 2009, stackoverflow.com/questions/1932838/create-symbol-table.

“Difference between an LL and Recursive Descent Parser?” *Stackoverflow*, 2009, stackoverflow.com/questions/1044600/difference-between-an-ll-and-recursive-descent-parser.

“TDD.” *Agile Alligiance*, www.agilealliance.org/glossary/tdd/#q=~(infinite~false~filters~(postType~(~'page~'post~'aa_book~'aa_event_session~'aa_experience_report~'aa_glossary~'aa_research_paper~'aa_video)~tags~(~'tdd))~searchTerm~'~sort~false~sortDirection~'asc~page~1).