RECURSIVE DESCENT PARSER – A COURSEWARE

A Project

Presented to the faculty of the Department of Computer Science
California State University, Sacramento

Submitted in partial satisfaction of
the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

by

Drashti Dilipkumar Kothari

FALL
2013
Student: Drashti Dilipkumar Kothari

I certify that this student has met the requirements for format contained in the University format manual, and that this project is suitable for shelving in the Library and credit is to be awarded for the project.

__________________________, Graduate Coordinator
Dr. Nikrouz Faroughi

Department of Computer Science
Abstract

of

RECURSIVE DESCENT PARSER – A COURSEWARE

by

Drashti Dilipkumar Kothari

Recursive Descent Parser is a top-down parsing approach built from set of recursive procedures for each non-terminal in grammar production rule. Recursive Descent Parser recognizes whether lexemes are in proper order or not.

The objective of the project is to provide web interactive parser tutorial and make it available to other users to improve their learning and understanding in recursive descent parsing method. Additionally the courseware will be a platform for students to find other related and useful material.

A web based parser courseware is specially focuses on recursive descent parsing tutorial and web interactive implementations of recursive descent parser. The courseware is effective and efficient method to provide useful material to learn key concept of parsing methodology. Along with learning material, it brings out web interactive implementation of recursive descent parsing for starters. It includes implementation support documents. It presents generalized implementation steps to build parser using recursive descent parsing technique.
As a part of courseware, five different implementation examples using functional programming language are included. The courseware is designed to be integrated with CSc135 (Computing Theory and Programming Languages) course website.

_______________________, Committee Chair
Dr. Meiliu Lu

_______________________
Date
ACKNOWLEDGEMENTS

I do take the opportunity to acknowledge and honor the contribution of my project advisor, Dr. Meiliu Lu for her support and guidance throughout the project. I thank her for providing benevolent direction, valuable suggestions and time worthy interactions. Her elaborated comments were beneficial throughout the project design and development.

My sincere thanks to Dr. Chung-E Wang for his detailed review and productive remarks on the project report. I also thank Dr. Nikrouz Faroughi for his review and advice for the successful completion of this project.

Finally my special thanks and appreciation to my family and friends, without whose love and support I could not have completed this difficult endeavor.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Acknowledgements</th>
<th>vii</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>viii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>x</td>
</tr>
<tr>
<td>Chapter</td>
<td></td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2. BACKGROUND</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Purpose and Scope</td>
<td>4</td>
</tr>
<tr>
<td>2.2 Project Design and Development Overview</td>
<td>4</td>
</tr>
<tr>
<td>3. RESEARCH AND ANALYSIS</td>
<td>6</td>
</tr>
<tr>
<td>4. COURSEWARE DESIGN AND DEVELOPMENT</td>
<td>8</td>
</tr>
<tr>
<td>5. WEB INTERACTIVE RECURSIVE DESCENT PARSER IMPLEMENTATION</td>
<td>16</td>
</tr>
<tr>
<td>5.1 Choice of Language</td>
<td>16</td>
</tr>
<tr>
<td>5.2 Recursive Descent Parsing Process</td>
<td>18</td>
</tr>
<tr>
<td>5.3 Web Interactive Implementation</td>
<td>20</td>
</tr>
<tr>
<td>5.4 Authentication Feature for Code Download</td>
<td>25</td>
</tr>
<tr>
<td>6. ILLUSTRATION</td>
<td>29</td>
</tr>
<tr>
<td>7. CONCLUSION</td>
<td>49</td>
</tr>
<tr>
<td>Appendix A. Source Code for Grammar Classes</td>
<td>50</td>
</tr>
</tbody>
</table>
Appendix B. Source Code for Web User Interface .......................................................... 68
Bibliography ...................................................................................................................... 73
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figures</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Figure 1 Courseware Sitemap</td>
<td>9</td>
</tr>
<tr>
<td>2. Figure 2 Courseware Home Page</td>
<td>10</td>
</tr>
<tr>
<td>3. Figure 3 Major Compiler Operations</td>
<td>11</td>
</tr>
<tr>
<td>4. Figure 4 Recognizer Process</td>
<td>11</td>
</tr>
<tr>
<td>5. Figure 5 Courseware Tutorial Page</td>
<td>12</td>
</tr>
<tr>
<td>6. Figure 6 Courseware Recursive Descent Parser – Basics</td>
<td>13</td>
</tr>
<tr>
<td>7. Figure 7 Courseware Example Page</td>
<td>14</td>
</tr>
<tr>
<td>8. Figure 8 Example Grammar Rules</td>
<td>17</td>
</tr>
<tr>
<td>9. Figure 9 Exercise Grammar Rules</td>
<td>18</td>
</tr>
<tr>
<td>10. Figure 10 Recursive Descent Parser Implementation – Steps</td>
<td>20</td>
</tr>
<tr>
<td>11. Figure 11 Recursive Descent Parser Implementation – Class Diagram</td>
<td>21</td>
</tr>
<tr>
<td>12. Figure 12 Recursive Descent Parser Implementation – Input page for Arithmetic Expression</td>
<td>23</td>
</tr>
<tr>
<td>13. Figure 13 Recursive Descent Parser Implementation – Output page for Arithmetic Expression</td>
<td>24</td>
</tr>
<tr>
<td>14. Figure 14 Source Code Download Link</td>
<td>26</td>
</tr>
<tr>
<td>15. Figure 15 HTTP Authentication Request Page</td>
<td>27</td>
</tr>
<tr>
<td>16. Figure 16 Source Code Download Prompt</td>
<td>28</td>
</tr>
<tr>
<td>17. Figure 17 Arithmetic Expression Execution Screenshot</td>
<td>48</td>
</tr>
</tbody>
</table>
Chapter 1

INTRODUCTION

Languages are a potentially infinite set of strings [1]. Each sentence in language is combination of tokens ordered in specific structure. Grammar is set of rules describing a language [1]. Lexical and syntactical analysis is a process, which takes source code as input, and returns syntax errors, parse tree and data structures. Compilers are important part of computer science, and even if you do not write a compiler program, it is necessary to have good knowledge in this area for various different reasons. Many different processes that we see regularly uses compiler as a concept. Concept of compiler is in interpreter, smart editors, program debugger, AI etc. Compilation process is multiphase process in which input is source code and output is object code. Different phases involved in compilation process are, Lexical Analysis, Parsing (Syntactic Analysis), Semantic Analysis, Optimization and Code Generation.

Syntactic analysis or parsing is required to determine if the input string (or sentence) is in right form or not. However, it is true that not all syntactically valid sentences are meaningful, semantic analysis is been applied for that. A parser receives a string of tokens from lexical analyzer and constructs a parse tree if the string of tokens could generated by the grammar of the source language; otherwise, it reports the syntax errors present in the source string. The generated parse tree is passed to the next phase of the compiler. Parsing is a process of analyzing a text, which contains a sequence of tokens, to determine its grammatical structure with respect to a given grammar.
Recursive Descent Parsing is one of the most aboveboard way of parsing. As the name indicates, one needs a construct recursive functions to implement parser program. Recursive Descent Parsing follows top-down parsing approach in which parser program tries to verify if the given input stream is syntactically correct as it reads from left to right [2]. In other words, recursive descent parsing is one of the parsing techniques, which takes input and identifies if it belongs to the given language or not based on generated grammar rules.

The Widespread use of computer courseware in numerous fields and domains has given quite an impact on our education system all around the world. Courseware is effective and efficient learning platform for students. This approach of study provides students with possibility of learning at their own pace. It provides with learning material and implementation examples to follow. During the learning of computing theory, this courseware is specially focused in recursive descent parsing.

In this project, we have developed a web interactive courseware as a learning platform for parser related tutorial. Along with that, it also contains recursive descent parser implementations guideline and samples for web interactive implementation of parser. Using provided guideline one can construct interactive parser using any programming language which supports concept of recursion.
Chapter 2

BACKGROUND

Recognizer is like a scanner, which also determines whether lexemes are in proper order or not. Recursive descent parsing is the top-down parsing [10] approach, which is used to build recognizer for your language. Recursive descent parser is a recursive program to recognize sentences in the language. It consists of a set of methods, one for each nonterminal symbol in grammar definition. You can write recursive descent parser in any programming language, which supports recursion.

Once you have prepared a grammar for your language, next step is to construct the recognizer for your language. Recursive descent is the most quick and easy way to build your recognizer. Basic steps involved in construction of a parser using recursive descent technique are,

1. Reading character from input stream from left to right and one at a time
2. Matching input character with grammar that describes the syntax of your language.

The implementation process description will be provided in more elaborated steps later in the report.

I have introduced recursive descent parsing and basics on courseware in first chapter. I will state purpose of the project and overview on what has been done in this chapter.
2.1 Purpose and Scope

Motivation for this project came from student comments and suggestion provided by students in computing theory and programming languages class assignment. Assignment was to implement web interactive recursive descent parser. According to student’s suggestions, some help was required in implementation of web interactive recursive descent parser.

The main purpose of the project is to provide web interactive parser tutorial and make it available to audience who is interested in learning concepts of recursive descent parsing. Additionally it serves as a platform for learning parser related concepts as well. Major focus of the courseware is recursive descent parsing methodology, and a set of implementation examples.

2.2 Project Design and Development Process Overview

The complete process of design and development has carried out in two phases. First phase focuses on courseware website design and development, and the second phase focuses on five different web interactive parser implementations using recursive descent parsing technique.

A courseware website contains learning materials related to compiler process, compilation phases, parsing, parsing techniques and recursive descent parsing. Along with above-mentioned tutorials, it contains learning material related to recursive descent parsing in-depth. Implementation steps of recursive descent parsing, which can work with
any programming language implementation supporting recursion. It also describes web interactive implementations of parser using recursive descent technique. Additional tutorials for related terms and concepts like Predictive Parsing, BNF to EBNF, First and Follow sets [11] etc. are also included into courseware web site.

The second phase of project, that of contains implementation, is done separately. As mentioned previously, five different set of grammar rules are implemented using programming language. I have carried out my implementation using PHP, which allows web interactive development of parser. I have integrated all the five implementations of recursive descent parsers with courseware for demonstration purpose. Demonstration pages for each implementation contains details for grammar rules and acceptable language details and some sample input and expected output.

Both courseware and sample implementations will be discussed in later chapters of this report. In next chapter, we will see what research has done during project design and development. Additionally I have presented courseware coverage details in next chapter.
Chapter 3

RESEARCH AND ANALYSIS

Research and analysis are the most important part of any project. Main importance of research is to produce knowledge. Research in the field of recursive descent parsing has carried out first, followed by all the related terms and concepts to put into courseware. Additionally research on how courseware can build and what should be included in this type of educational courseware had performed. Research on different grammar became necessary for the implementation purpose. All the research done at start of the project provided me with lots of information and depth understanding of recursive descent parsing.

I started with the basic compilation process, in which parsing is included. As parsing or syntactic analysis is the part of compilation process, providing knowledge regarding compilation process is a good start. Next, I learned more about parsing and parsing techniques. The most important part in research process was to learn about recursive descent parsing. Studying recursive descent parsing concept in depth helped me come up with basic steps to implement parser using recursive descent methodology.

I learn what kind of grammars will be suitable for recursive descent implementation during my research, and that helped me pickup proper grammars or converting my grammar to match format suitable for recursive descent parsing.

Along with that, I picked list of related terms and concepts to write some essays to provide reading materials all on the same platform. Some of the most important terms I
have picked up for my courseware are, Predictive Parsing [15], First and Follow sets [11], BNF-EBNF [6] and Compiler-Compilers [13] [14].

My research gave me path towards my design and development phase. I concluded my research with list of topics required to include in courseware and list of grammar rules to work on for implementation purpose. Next task is to come with courseware design and implementation steps.
Chapter 4

COURSEWARE DESIGN AND DEVELOPMENT

Courseware is a software program, which contains educational material as a tutorial for students and supplementary teaching tool for teachers. It is effective and efficient platform for providing learning materials to students. This kind of material sharing platform guaranties same knowledge distribution among readers. Idea behind courseware is to provide student with course material in easily understandable format. It helps reduce learning time for students and gives students some privilege to learn on their own pace.

Idea here is to come up with something, which helps student learn concept of parsing, specially focused in recursive descent technique. Along with theoretical concept description of recursive descent parsing, some practical implementation using programming language is useful. Considering this, I have included some examples with open code and some exercises without code as a part of this courseware.

A web interface of this courseware was constructed using PHP5, HTML, JavaScript and CSS mainly. Courseware website gives tutorial on background concepts like compiler process, phases of compiler process, parsing basics and parsing techniques. Recursive descent is one of the parsing techniques, which relates concepts given in background with the recursive descent-parsing page. Moreover, steps to implement recursive descent parser using programming language are part of this page as well. Along with these tutorials, I have included few more related concepts as extra readings under tutorial page. Last in the series of web pages, we have FAQ and references pages. FAQ
contains few frequently asked questions in this area of study. References page includes list of references used to construct this courseware contents and some other useful online and paperback book readings. Figure 1 shows the sitemap of courseware website, which provides basic content organization idea.

![Courseware Sitemap](image)

Figure 1 Courseware Sitemap

Figure 2 bellow contains a partial screen shot of courseware home page. Header menu lists major web pages for the site. Figure 5 and Figure 6 shows tutorial page and recursive descent parsing page respectively. It shows content arrangement for recursive descent parsing and related tutorials. Example and Exercise pages are demo pages, which redirects users to specific demo page, as shown in Figure 7.
As you can see in Figure 2, this courseware contains details on compiler process and different phases in compilation process [6]. Figure 3 below shows major operations involved in compilation and where exactly parser or syntactic analysis fits in.

Figure 2 Courseware Home Page
As you can see in figure 3, parser comes into picture after lexical analysis and before optimization. Output of parser is a parse tree, and before parse tree generation, parser recognizes input for syntactical validity. Figure 4 bellow shows how recognizer works. Recognizer takes result of lexical analyzer, which is set of tokens and checks those tokens against grammar rules of source program language. Output of recognizer is success in case when given input matches grammar and failure otherwise.

Figure 3 Major Compiler Operations

Figure 4 Recognizer Process
Figure 5 shows tutorial page, which includes related concepts and their brief explanations along with detailed reading reference links. It includes tutorials on predictive parsing, first and follow sets, compiler-compiler (compiler generators) and list of some widely used compiler generators.
Figure 4 shows recursive descent parsing basics and implementation steps. We will see details on implementation steps in next chapter, which discusses the implementation steps in details.

Figure 6 Courseware Recursive Descent Parser - Basics

Layouts of example pages and exercise pages are similar. All of the grammars presented in both pages are implemented using web programming language PHP5. I have done implementations using web interactive interfaces. Following screen shots (Figure 7) shows arrangement of demo pages. Link given under each grammar rules will lead user to specific a demo page.
Exercise page looks exactly same as above but includes some other grammar rules and demo links. Purpose of this page is to check students’ knowledge and assign some practical work to implement. Idea is to ask students to follow steps presented for implementation and implement their own parser for grammars given on exercise page.

Figure 7 Courseware Example Page
Domo for grammars in exercise page will be available to students, for sample purpose. They can run demo to understand execution and expected outputs.

Some authentication implantation has done using HTTP authentication mechanism to protect code. Download instruction for open code is given in digital document, which will be presented as support document. Web based parser implementation description is given in next chapter. It bring out details on design and development process of different grammars. All the implementations are web based.
Chapter 5

WEB INTERACTIVE RECURSIVE DESCENT PARSER IMPLEMENTATION

Recursive descent parsing is the most straightforward way of implementing parser. It is a top-down parsing approach, in which parser attempts to verify syntax of given input based on grammar rules generated for some language. What a recursive descent parser actually does is it perform a depth-first search of the derivation tree for the string given for parsing. This provides the 'descent' portion of the name. The 'recursive' portion comes from the parser's form, a collection of recursive procedures [2]. Rest of this chapter is divided in to 4 sub-sections, which shows implementation approach taken for the web interactive implementation of parser using recursive descent technique.

5.1 Choice of Language

This section describes grammar rules or languages chosen to work with. I have implemented five different set of grammar rules while working on this project. Following figures (figure 8 and 9) shows all five set of grammar rules. Figure 8 shows grammar for integer number where input can be any positive or negative integer. Input string will be any integer, signed or unsigned. I have used this grammar to run through generic steps proposed for recursive decent parsing. Next is S-Expression, In computing, s-expressions, sexprs or sexps (for "symbolic expression") are a notation for nested list (tree-structured) data, invented for and popularized by the programming language Lisp, which uses them for source code as well as data [3]. In the usual parenthesized syntax of Lisp, we classically define an s-expression as,
1. an atom, or

2. an expression of the form \((a, a)\) where ‘a’ is s-expression.

List of grammar rules implanted as exercise are as in Figure 9. These implementations are not open code, as they are part of exercise, which is to test knowledge gained from courseware. It includes implementation of arithmetic expression, regular expression and decimal number grammars.

First grammar in exercise series is for arithmetic expressions. In this case valid input will be combination of symbol ‘+’, ‘-‘, ‘*’, ‘/’, ‘(’ and ‘)’. Along with symbols mentioned it would take any digit between ‘0’ and ‘9’. Next is grammar for regular expression. It takes terminals like a, b, empty string ’e’, ‘{’ and ‘*’. Last is grammar for decimal number, valid input could be any combination of one or multi-digit numbers either integer or decimal. You can pass either signed or unsigned decimal number.
A recursive descent parser consists of a collection of subprograms, many of which are recursive, and it produces a parse tree in top-down (descending) order [4]. This recursion is a reflection of the nature of programming languages, which include several different recursive structures. Recursive descent parsing comes under top-down and predictive parsing techniques. Recursive descent parsers are quite versatile and appropriate for a hand-written parser.

Figure 9 Exercise Grammar Rules

5.2 Recursive Descent Parsing Process
EBNF is ideally suitable for recursive descent parsing and that is when our first step comes in. First step in our process involves converting BNF to EBNF [5]. Other than that, we will make sure that grammar is suitable for recursive descent parsing. Not all grammars are suitable to recursive descent parsing. The grammar containing left recursion directly or indirectly is not suitable for recursive descent parsing implementation, it can cause infinite recursive loop, so one should transform it into right recursion. Once after you eliminate all left recursions, there is a straightforward transformation from a grammar rule to a function, which implements the grammar rules. Before we go into implementation, deciding on implementation language becomes necessary; in our case, implementation language chosen to work with is PHP5.

A next step is to work on syntax diagram and prepare pseudo code based on grammar generated. Each left hand side of rule (non-terminals) will become procedures. Body of each function will be series of if statements, which will choose which production to use from right hand side of the grammar rule (terminals), and for all non-terminals, we will calls related recursive functions. Next task is to prepare test cases and trace pseudo code against those test cases.

After pseudo code, one can write program closely matching pseudo code. Depending on language you choose to build your recognizer, you need to strategize your program structure. Finally run your program against test cases you created in earlier step. Figure 10 bellow gives overall steps of implementation.
5.3 Web Interactive Implementation

Code implementation of recursive descent parser is straightforward once you have grammar and pseudo code ready. I have picked PHP5, which is a server side-scripting language designed for web development. PHP5 supports object oriented programming features, and that is added advantage. It allows features like reusability, abstraction and inheritance. Figure 11 shows class diagram of my implementation.
Grammar class is the main or top-level class for parsing string and matching it to given grammar. It is abstract class. It contains common attributes like,

- **inputString** which contains input string entered by end user
- **pinterInString** which represents current point position in input string
- **resultString** is a Boolean variable which contains true or false based on parsing result
- **endOfString** checks for end of string delimiter in input string

Major operations included in Grammar class are,

- **abstract function exp()** – this function is the starting point of each parser implementation.

Figure 11 Recursive Descent Parser Implementation - Class Diagram
- **match()** – match function basically matches character token with current pointer character, if matches it advances the pointer and returns true, otherwise it will return false.

- **endOfInput()** – this function checks for end of input or empty string, if any of these is true it will sets global variable isDone to true.

- **isresultString()** – this function is the last function call in implementation. It will return result as true or false which leads to the valid or invalid output.
Grammar rules implementation is in separate class, which extends basic grammar class. Operations of child class depends on grammar rules, but as mentioned previously it will closely represents all non-terminals as one operation. Therefore, as shown in Figure 11 – class diagram, arithmetic expression parser requires four different functions or operation implementations, each representing one grammar rule or non-terminal. Figure
12 and Figure 13 below represents screenshots of input and output for arithmetic expression implementation respectively.

Figure 13 Recursive Descent Parser Implementation – Output Page for Arithmetic Expression

As you can see in Figure 12, it will provide user with grammar details and input box to enter input string to check against grammar rules presented. Along with grammar details and input box, this page represents few sample inputs and whether it is valid or
invalid. Once user enters input string and hits submit button, this file calls class representing arithmetic expression grammar rules implementation. Entry point will be function `exp()` which is defined in grammar class. Once complete input string is processed, it will call function `isresultString()` to print output final result. Figure 13 shows screenshot of one valid execution for arithmetic-expression parser implementation. In case when input string is not valid for given, grammar output page will look almost same except for the message between grammar rules and sample inputs. For invalid input output message will say input string is invalid.

All the five implementation looks almost the same except for individual child class, which extends grammar class. Each child class will have different set of operations based on grammar rules. All the common function defined in grammar class are shared functions for child classes, which provides code reusability and readability.

5.4 Authentication Feature for Code Download

As a part of courseware, I am providing source code of all my implementations to students for educational review. Source code download is available with proper authentication. Instructor will provide students with proper authentication details as and when required. I am using `PHP_AUTH_DIGEST` [7] for authentication page, which provides HTTP authentication. HTTP authentication function call sends authentication-required message to user and prompts user with username and password. Properly entered credentials will allow user to download source code. Download link is provided
at the end of implementation page which leads user to authentication page. Such authentication will allow restricted distribution of source code.

![Recursive Descent Parser Implementation](image)

Figure 14 Source Code Download Link

Figure 14 shows download link provided at the bottom of implementation page. Figure 15 shows HTTP authentication request page and Figure 16 shows download option once after proving correct credentials. Once user has provided authentication code,
download code option will be available. User can click on download code and pdf will get download containing source code.

Figure 15 HTTP Authentication Request Page
Web interactive recursive-descent parser implementation is relatively simple if all the concepts are clear and prior knowledge of recursion through one programming language. This courseware is been designed in a way that by following implementation steps you can easily create your own parser using recursive descent technique. As all the related terms are available on a single platform, user can concentrate more on understanding contents rather than searching for details here and there.

Following chapter provides systematic execution example for arithmetic-expression grammar rules. This will provide user with practical implementation guideline with illustration.
Chapter 6

ILLUSTRATION

In this chapter, we will take one of the implementations and match it with the steps given in previous chapter. We will start with one of grammars given in exercise grammar list, arithmetic expression. I am including grammar rules bellow for quick reference.

BNF Grammar for Arithmetic Expression,

\[
\begin{align*}
\text{EXP} & ::= \text{EXP} + \text{TERM} \mid \text{EXP} - \text{TERM} \mid \text{TERM} \\
\text{TERM} & ::= \text{TERM} \ast \text{FACTOR} \mid \text{TERM} / \text{FACTOR} \mid \text{FACTOR} \\
\text{FACTOR} & ::= (\text{EXP}) \mid \text{DIGIT} \\
\text{DIGIT} & ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*}
\]

Step 1 – Convert BNF to EBNF. Make sure grammar is suitable for recursive descent parsing and decide on programming language for implementation

- EBNF for the grammar give for arithmetic expression would be as follow,

\[
\begin{align*}
\text{EXP} & ::= \text{TERM} \{ ( + \mid - ) \text{TERM} \} \\
\text{TERM} & ::= \text{FACTOR} \{ ( \ast \mid / ) \text{FACTOR} \} \\
\text{FACTOR} & ::= (\text{EXP}) \mid \text{DIGIT} \\
\text{DIGIT} & ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*}
\]

- Next is to check if grammar is suitable for recursive descent parsing or not. This will require elimination of left recursion as it might end up in infinite recursive calls. In our case, there is no left recursion so we will skip this step.
• Other important thing is to check left factoring especially if you are planning to implement your parser in language like prolog. Left factorization is helpful when first part of two or more production is the same and the rest of the similar productions are different. Left factorization help improve efficiency in non-imperative language like prolog. We will skip this step, as our grammar does not require any left factorization.

Step 2 – Syntax diagram or Pseudo code generation along with common function specification.

We will take pseudo code approach; it will make our programming implementation easy, as it will be close to what we will put in pseudo code.

-- Take input from user, with string delimiter ‘$’

-- Call constructor function,

Which will contain variable initialization and first non-terminal function call

```plaintext
__constructor() { 
Assign user input to string variable inputString;
Initialize pointer (pointerInString) pointing to current character in input string;
Initialize resultString with true; // false incase when user input character at any point does not match to expected character
Initialize delimiter character; // in this case it will be ‘$’
Make call to function for first non-terminal in rule set; // it will be exp() in our case
}
```
-- We need to define some common functions as well, before we start with specific case.

Function Match() {

// compare current character with expected character and returns true if matches and advances the pointer

In case when no match found it will return false

}

Function endOfInput{

// checks if we reach to end of input string or not.
// it also checks for blank input

}

Function isresultString() {

// last function to call which will return final result … if given input is valid according to grammar rules or not

}

-- Once all these common functions are ready, it is time to write function for each rule.

As mentioned previously, we will have to write one function for non-terminals each, body of those functions will be right hand side of production rules.

function exp() {

Call term()

Check if we are done or not – call endOfInput;

while(!done)
if(inputString == + || inputString == -)
{
    match(pointerInString)
    Call term()
}

elseif(inputString == "\)" || endOfString)
    done = true;
else
    resultString = false;
}

function term() {
    call factor();
    while(!done)
    {
        if(inputString == * || inputString == /)
        {
            match(pointerInString);
            call factor();
        }
        elseif(inputString == + || inputString == - || inputString == \) || endOfString)
            done = true;
    }
else
    resultString = false;
}
}

function factor() {
    if(endOfInput())
        $this->resultString = false;
    if(resultString)
        {
        ifpointerInString == '('
        {
            match(pointerInString);
            exp();
            resultString = match(')');
        }
        else
            call digit();
        }
}

function digit() {
    digitArray = array('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');
    if(endOfInput)
resultString = false;

elseif(search in array(pointerInString, digitArray) !== False)
    resultString = resultString && match(pointerInString);
else
    resultString = false;
}

As you see in pseudo code, we have function for each non-terminal and we are checking all the production rules with conditioning and branching. If you will follow these simple steps, you can write parser for any given grammar using recursive descent technique.

**Step 3 – Tracing pseudo code against some valid and invalid test cases**

Starting to test your product even before you, start development is helpful. It is one way to look through your requirements and validate it even before you write your functional code.

I did my test with some valid and invalid inputs to make sure my pseudo code work the way it should be. In this example, I took total six test cases based on general definition of arithmetic expression language. Language allows string with symbols like ‘+’, ‘-‘, ‘*’, ‘/’, ‘(’ and ‘)’. Along with these symbols, you can include numbers between 0 and 9 as operands. You operand can be another expression as well. Valid inputs for the given grammar are, ‘1-2*2$’, ‘(1-2)*3$’, ‘2/3$’. Invalid inputs will be something like, ‘22-3$’, ‘(1-4)*3$’, ‘2/$’. My pseudo code passes all tests with expected output.
Step 4 – Parser implementation in functional programming language.

Now it is time to transform your pseudo code into actual program using any programming language that supports recursion. As mentioned previously, I choose PHP5 for my web interactive parser implementation. My input and output page is written using HTML, PHP5, JavaScript and CSS, whereas my parser is written in PHP5. Following code represents both input/output file and parser.

```
<?php

/********************************************
 * Grammar class
 * The Grammar class is a top-level class for parsing a string
 * Grammar rules will be implemented in to another class, which will extend this class
 */

abstract class Grammar {

    // User input - string.
    protected $inputString;

    //Pointer pointing to current position in input string
    protected $pointerInString;

    // boolean variable which will return true or false based on parsing result
    protected $resultString;

    //end of string variable '$' - in this case'.
```
protected $endOfString;

/**
 * Recursive Descent Parser
 * This function will get overridden by child classes
 */
abstract protected function exp();

function __construct($input, $delimiter = '$') {
    $this->inputString = $input; // user input string taken from input page
    $this->pointerInString = 0; // initial pointer value will be 0 - pointer pointing to first
    // character in input string
    $this->resultString = true; // it will be set to false if program cannot match string to the
    // expected at any point in time while execution
    $this->endOfString = $delimiter;
    $this->exp(); // starting point for each parsing
    if(!$this->endOfInput())
        $this->resultString = false; // this means the string contains some unparsable character
    }

/*
 * True if expression is resultString else False
 */

function isresultString() {
    return $this->resultString;
protected function endOfInput() {

// check for end of the string
$isDone = ($this->pointerInString >= strlen($this->inputString)) || (strlen($this->inputString) == 0);
if($this->pointerInString == (strlen($this->inputString) - 1))
  if($this->inputString[$this->pointerInString] == $this->endOfString)
    $isDone = true;
return $isDone;
}

/*
* match function basically matches character with current pointer character
* if matches, it will advance pointer to next character and return true.
*/
protected function match($myToken) {
  if(($this->pointerInString < strlen($this->inputString)) &&
     ($this->inputString[$this->pointerInString] == $myToken))
  {
    $this->pointerInString += 1;
    return true;
  }
else
return false;

}

/**
 * Grammar for RDR1 is:
 * EXP ::= EXP + TERM | EXP - TERM | TERM
 * TERM ::= TERM * FACTOR | TERM / FACTOR | FACTOR
 * FACTOR ::= (EXP) | DIGIT
 * DIGIT ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
 * Assume the input ends with '$'.
 */

class RDR1 extends Grammar {
    function exp() {
        if($this->resultString)
            $this->term();
        $done = $this->endOfInput(); // if there's nothing left to process, we're done
        while($this->resultString &amp;&amp; !$done)
        {
            if(($this->inputString[$this->pointerInString] == '+')
                || ($this->inputString[$this->pointerInString] == '-'))
            {
                $this->match($this->inputString[$this->pointerInString]);
            }
function term() {
    if($this->resultString)
        $this->factor();
    $done = $this->endOfInput(); // if there's nothing left to process, we're done
    while($this->resultString && !$done)
    {
        if(($this->inputString[$this->pointerInString] == '*') || ($this->inputString[$this->pointerInString] == '/'))
            $this->resultString = false;
    }
}

/*
 * handle processing for the term rule in the grammar
 */

elseif(($this->inputString[$this->pointerInString] == ')') || ($this->inputString[$this->pointerInString] == $this->endOfString))
{
    $done = true;
}
else
    $this->resultString = false;
}
```php
{$this->match($this->inputString[$this->pointerInString]);
$this->factor();
}
elseif(($this->inputString[$this->pointerInString] == '+' ||
($this->inputString[$this->pointerInString] == '-') ||
($this->inputString[$this->pointerInString] == ')') ||
($this->inputString[$this->pointerInString] == $this->endOfString))
{
$done = true;
}
else
    $this->resultString = false;
}

/*
* handle processing for the factor
*/
function factor() {
if($this->endOfInput())
{$this->resultString = false;
if($this->resultString)
```
if($this->inputString[$this->pointerInString] == '(')
{
    $this->match($this->inputString[$this->pointerInString]);
    $this->exp();
    $this->resultString = $this->resultString && $this->match(')');
}
else
{
    $this->digit();
}
/*
 * If the character at the current position is in [0..3]
 * advance the position pointer else change resultString to false.
 */
function digit() {
    $digitArray = array('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');
    if($this->endOfInput())
        $this->resultString = false;
    elseif(array_search($this->inputString[$this->pointerInString], $digitArray) !== False)
{
$this->resultString = $this->resultString && $this->match($this->inputString[$this->pointerInString]);
}
else
$this->resultString = false;
}
Recursive Descent Parser Implementation

**Grammar for Arithmatic Expression**

\[
\begin{align*}
\text{EXP} & \::= \text{EXP} + \text{TERM} \mid \text{EXP} - \text{TERM} \mid \text{TERM} \\
\text{TERM} & \::= \text{TERM} \times \text{FACTOR} \mid \text{TERM} / \text{FACTOR} \mid \text{FACTOR} \\
\text{FACTOR} & \::= (\text{EXP}) \mid \text{DIGIT}
\end{align*}
\]
DIGIT ::= 0|1|2|3|4|5|6|7|8|9 <br>
</p>
<?php
if(!isset($_POST['exp']))
{
    // ask user for initial input
    ?>

    <?php
}
else
{
    // handle parsing and display result
    $userInput = $_POST['exp'];
    require_once 'grammarClass.php';
    $rdr1 = new RDR1($userInput);
    ?>

    <?php
    if($rdr1->isresultString())
    {
        ?>
        <div><p><font color="green">Input String <php echo $userInput; ?> is
        <strong>Valid</strong>.</font></p></div>
Example of some valid strings: <br>
<code>1-2*2$</code> <br>
<code>(1-2)*3$</code> <br>
<code>2/3$</code> <br>
</p>
<p>Example of some invalid strings: <br>
<code>22-3$</code> <br>
<code>(1-4)*3$</code> <br>

Enter string using the numbers 0 to 9, and the symbols +, -, *, /, (, and ).

Your end of string variable will be dollar sign ($).

When you click on 'submit', it will check whether entered string is valid or not for a given grammar.

Input String: 

```
<form action="rdr1.php" method="post">

Enter string using the numbers 0 to 9, and the symbols +, -, *, /, (, and ).

Your end of string variable will be dollar sign ($).

When you click on 'submit', it will check whether entered string is valid or not for a given grammar.

Input String: 

```

<!DOCTYPE html>
<html>
<body>

<form action="rdr1.php" method="post">

Enter string using the numbers 0 to 9, and the symbols +, -, *, /, (, and ).

Your end of string variable will be dollar sign ($).

When you click on 'submit', it will check whether entered string is valid or not for a given grammar.

Input String: 

```

<!DOCTYPE html>
<html>
<body>

</body>
</html>
As you can see, actual code is fairly close to the pseudo code.

Step 5 Test your parser against the test cases used previously to test pseudo code

In this step, you will run your program with same test cases that you used in pseudo code testing. Your parser should return expected results as valid or invalid. Figure 17 shows one execution of valid input.

This illustration is the way of presenting sample execution of implementation. It represents how you can perform recursive-descent parser implementation using steps presented in previous chapter. This shows integration of theoretical details of courseware and practical integration of web interactive parser using programming language. This is just one illustration out of five different implementations done, rest are attached in Appendix A. Appendix B contains input and output file for one more implementation for readers reference. Input and output file for each execution will look almost the same except for we will create object of corresponding class, and that is why Appendix B contains just one out of five input and output files.
Recursive Descent Parser Implementation

Grammar for Arithmetic Expression

\[ EXP := EXP + TERM \mid EXP - TERM \mid TERM \]
\[ TERM := TERM * FACTOR \mid TERM / FACTOR \mid FACTOR \]
\[ FACTOR := (EXP) \mid DIGIT \]
\[ DIGIT := 0|1|2|3|4|5|6|7|8|9 \]

Input String \((1-2)*3\) is Valid.

Example of some valid strings:

\[ 1 - 2 * 2 \]
\[ (1 - 2) * 3 \]
\[ 2 / 3 \]

Example of some invalid strings:

\[ 22 - 3 \]
\[ (1-4) * 3 \]
\[ 2 / \]

Enter string using the numbers 0 to 9, and the symbols +, -, *, /, (, and ).
Your end of string variable will be dollar sign ($). When you click on 'submit', it will check whether entered string is valid or not for a given grammar.

Input String: [Input Field]  [Submit]

Figure 17 Arithmetic Expression Execution Screenshot
Chapter 7

CONCLUSION

The purpose of a courseware is to deliver information related to specific topic and improve readers learning. In this project, we have seen how this courseware will be effective for the students learning concepts of computing theory and programming languages. Main objective of the project was to provide courseware on recursive descent parsing along with web interactive parser implementation tutorial.

Therefore, as a conclusion to this project report, we can say that we have achieved primary goal defined at the initiation. This courseware provides theoretical concepts related to parsing and implementation phase provides details on how to implements recursive descent parser in practice. Related reading materials and tutorials will allow user to dig deep into concept for more knowledge gain in the field.

Suggested future work would be, modify implementation code so that it produces parser tree along with syntactical analysis. Other than that, some more information related to compiler-compiler [13] or parser generator [13] tools should be included into courseware. I learned about tools like Antlr, JavaCC, CppCC, YACC, COCO [14] during my research and that is something interesting to include into this courseware. Moreover is some quizzes can be added as a knowledge checker.
APPENDIX A

SOURCE CODE FOR GRAMMAR CLASSES

Source code for grammar classes includes six classes. Main class is Grammar, which is parent class and each child class extends this class. Following list shows which grammar is represented under particular class.

1) Class RDR1 – Grammar for Arithmetic Expression
2) Class RDR2 – Grammar for S-Expression
3) Class RDR3 – Grammar for Regular Expression
4) Class RDR4 – Grammar for Integer Number
5) Class RDR5 – Grammar for Decimal Number

<?php

/** The Grammar class is a main/top level class for parsing a string and matching it to a grammar. Grammar rules will be implemented in to another class which will extend this basic class */

abstract class Grammar {

protected $inputString; // User input - string.
protected $pointerInString; //Pointer pointing to current position in input string
protected $resultString; // boolean variable, returns true or false based on parsing result
protected $endOfString; //end of string variable '$ - in this case'.
abstract protected function exp(); //entry point for each implementation
function __construct($input, $delimiter = '$') {

    $this->inputString = $input; // user input string taken from input page
$this->pointerInString = 0; // pointer pointing to first character in input string

$this->resultString = true; // it will be set to false if program cannot match string
to the expected at any point in time while execution

$this->endOfString = $delimiter;

$this->exp(); // starting point for each parsing

if(!$this->endOfInput())

    $this->resultString = false; // this means the string contains some
    unparsable character

}

function isresultString() { /* True if expression is resultString else False */
    return $this->resultString;
}

protected function endOfInput() {
    $isDone = ($this->pointerInString >= strlen($this->inputString)) || (strlen($this->inputString) == 0);
    if($this->pointerInString == (strlen($this->inputString) - 1))
        if($this->inputString[$this->pointerInString] == $this->endOfString)
            $isDone = true;
    return $isDone;
}

/* * match function basically matches character with current pointer character
  * if matches, it will advance pointer to next character and return true. */
protected function match($myToken) {
    if(($this->pointerInString < strlen($this->inputString)) &&
        ($this->inputString[$this->pointerInString] == $myToken))
    {
        $this->pointerInString += 1;
        return true;
    }
    else
    {
        return false;
    }
}

/* Grammar for RDR1 is:

* EXP ::= EXP + TERM | EXP - TERM | TERM
* TERM ::= TERM * FACTOR | TERM / FACTOR | FACTOR
* FACTOR ::= (EXP) | DIGIT
* DIGIT ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

* Assume the input ends with '$'. */

class RDR1 extends Grammar {

    function exp() {
        if($this->resultString)
            $this->term();

        $done = $this->endOfInput(); // if there's nothing left to process, we're done
while($this->resultString && !$done)
{
    if(($this->inputString[$this->pointerInString] == '+' ||
        ($this->inputString[$this->pointerInString] == '-')))
        {
            $this->match($this->inputString[$this->pointerInString]);
            $this->term();
        }
    elseif(($this->inputString[$this->pointerInString] == ')') ||
        ($this->inputString[$this->pointerInString] == $this->endOfString))
        $done = true;
else
    {
        $this->resultString = false;
    }
}
/* handle processing for the term rule in the grammar */
function term() {
    if($this->resultString)
        $this->factor();
    $done = $this->endOfInput(); // if there's nothing left to process, we're done
    while($this->resultString && !$done)
    {
    }
if(($this->inputString[$this->pointerInString] == '*') || ($this->inputString[$this->pointerInString] == '/'))
{
    $this->match($this->inputString[$this->pointerInString]);
    $this->factor();
}
elseif(($this->inputString[$this->pointerInString] == '+') ||
    ($this->inputString[$this->pointerInString] == '-') ||
    ($this->inputString[$this->pointerInString] == ')') ||
    ($this->inputString[$this->pointerInString] == $this->endOfString))
    $done = true;
else
    $this->resultString = false;
}
/* handle processing for the factor */
function factor() {
    if($this->endOfInput())
        $this->resultString = false;
    if($this->resultString)
    {
        if($this->inputString[$this->pointerInString] == '(')
{ 
    $this->match($this->inputString[$this->pointerInString]);
    $this->exp();
    $this->resultString = $this->resultString && $this->match(')');
} 
else  
    $this->digit();
} 
/* * If the character at the current position is in [0..9], advance the position pointer else 
change resultString to false. */
function digit() {
    $digitArray = array('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');
    if($this->endOfInput())
        $this->resultString = false;
    elseif(array_search($this->inputString[$this->pointerInString], $digitArray) !== False)
    {
        $this->resultString = $this->resultString && $this->match($this->inputString[$this->pointerInString]);
    } 
    else
        $this->resultString = false;
```php
/** Grammar for RDR2:
* EXP ::= ( LIST ) | a
* LIST ::= LIST , EXP | EXP
* Assume the input ends with '$'. */
class RDR2 extends Grammar {
    function exp() {
        if($this->endOfInput())
            $this->resultString = false;
        if($this->resultString)
            
            if($this->inputString[$this->pointerInString] == 'a')
                
                $this->match($this->inputString[$this->pointerInString]);
            };
            elseif($this->inputString[$this->pointerInString] == '(')
                
                $this->match($this->inputString[$this->pointerInString]);
                $this->ecList();
            };
            if($this->endOfInput())
                $this->resultString = false;
    }
}
if($this->resultString)

    $this->match('}');

}
else

    $this->resultString = false;

}
}

function ecList() {

    $this->exp();
    $done = false;

    while(!$done && $this->resultString && !$this->endOfInput())

    {
        if($this->inputString[$this->pointerInString] == ',')

        {
            $this->match($this->inputString[$this->pointerInString]);
            $this->exp();
        }
        elseif($this->endOfInput() || $this->inputString[$this->pointerInString] == ')')

            $done = true;
        else

            $this->resultString = false;
        }
    }
class RDR3 extends Grammar {

function exp() {

if($this->resultString)

$this->term();

$done = $this->endOfInput(); // if there's nothing left to process, we're done

while($this->resultString && !$done)

{

if(($this->inputString[$this->pointerInString] == '+'))

{

$this->match($this->inputString[$this->pointerInString]);

$this->term();

}
elseif(($this->inputString[$this->pointerInString] == ')') || ($this->inputString[$this->pointerInString] == $this->endOfString))
    $done = true;
else
    $this->resultString = false;
}

/* * handle processing for the term rule in the grammar */

function term() {
    if($this->resultString)
        $this->factor();
    $done = $this->endOfInput(); // if there's nothing left to process, we're done
    while($this->resultString && !$done)
    {
        if(($this->inputString[$this->pointerInString] == '.))
        {
            $this->match($this->inputString[$this->pointerInString]);
            $this->factor();
        }
        elseif(($this->inputString[$this->pointerInString] == '+') || ($this->inputString[$this->pointerInString] == ')') || ($this->inputString[$this->pointerInString] == $this->endOfString))
        {
            $this->factor();
        }
        elseif(($this->inputString[$this->pointerInString] == '+') || ($this->inputString[$this->pointerInString] == ')') || ($this->inputString[$this->pointerInString] == $this->endOfString))
        {
            $this->match($this->inputString[$this->pointerInString]);
            $this->factor();
        }
    }
$done = true;

else

    $this->resultString = false;

}

} /* handle processing for the factor */

function factor() {

    if($this->endOfInput())

        $this->resultString = false;

    if($this->resultString)

        $this->id();

    $done = $this->endOfInput(); // if there's nothing left to process, we're done

   while(($this->resultString && !$done))

    {

        if(($this->inputString[$this->pointerInString] == '*'))

            {

                $this->match($this->inputString[$this->pointerInString]);

                $done = true;

            }

        else if(($this->inputString[$this->pointerInString] == $this->endOfString) || ($this->inputString[$this->pointerInString] == '+') || ($this->inputString[$this->pointerInString] == ')') || ($this->inputString[$this->pointerInString] == '.'))
$done = true;

else
{
    $this->match($this->inputString[$this->pointerInString]);
    $this->factor();
}

} */ * If the character at the current position is in [0..3]
* advance the position pointer else change resultString to false. */

function id() {
    $idArray = array('a', 'b', 'e');
    if($this->endOfInput())
        $this->resultString = false;
    elseif(array_search($this->inputString[$this->pointerInString], $idArray) !== False) //for a|b|" " //
    {
        $this->resultString = $this->resultString && $this->match($this->inputString[$this->pointerInString]);
    }
    elseif($this->inputString[$this->pointerInString] == '(') //for empty set (E) //
    {
    
}
$this->match($this->inputString[$this->pointerInString]);

$this->exp();

$this->resultString = $this->resultString && $this->match(')');
}

elseif($this->inputString[$this->pointerInString] == '{')  //for empty set {} //
{
    $this->match($this->inputString[$this->pointerInString]);
    $this->resultString = $this->resultString && $this->match('}');
}
else

    $this->resultString = false;
}
}

/** Grammar for RDR4 is:

* EXP ::= + NUM | -NUM | NUM
* NUM ::= NUM DIGIT | DIGIT
* DIGIT ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
* Assume the input ends with '$'. */

class RDR4 extends Grammar {
    function exp() {
        if($this->endOfInput())
            $this->resultString = false;
else
{
    if($this->resultString)
    {
        if(($this->inputString[$this->pointerInString] == '+' || ($this->inputString[$this->pointerInString] == '-')))
            $this->match($this->inputString[$this->pointerInString]);
        $this->num();
    }
}

/* handle processing for the term rule in the grammar */
function num() {
    $this->digit();
    while($this->resultString && !$this->endOfInput())
        $this->digit();
}

/* * If the character at the current position is in [0..9]
 * advance the position pointer else change resultString to false. */
function digit() {
    $digitArray = array('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');
    if($this->endOfInput())
function exp() {
    if($this->endOfInput())
        $this->resultString = false;
    else
        {
            $this->resultString = $this->resultString && $this->match($this->inputString[$this->pointerInString]);
        }
    $this->resultString = false;
}

/** Grammar for RDR4 is:

* EXP ::= + NUM | -NUM | NUM
* NUM ::= NUM DIGITS | DIGITS
* DIGITS ::= DIGIT | DIGIT . DIGIT
* DIGIT ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
* Assume the input ends with '$'. */

class RDR5 extends Grammar {

    function exp() {
        if($this->endOfInput())
            $this->resultString = false;
        else
            {
                // Code...
            }
}
if($this->resultString)
{
    if(($this->inputString[$this->pointerInString] == '+') || ($this->inputString[$this->pointerInString] == '-'))
        $this->match($this->inputString[$this->pointerInString]);
    $this->num();
}

/* * handle processing for the term rule in the grammar */

function num()
{
    if($this->endOfInput())
        $this->resultString = false;
    else
    {
        if($this->resultString)
        {
            $this->digits();
            if($this->inputString[$this->pointerInString] == '.')
                {
                    $this->match($this->inputString[$this->pointerInString]);
                }
$this->digits();

}

}

/* * handle processing for the digits */

function digits() {

$this->digit();

$done = false;

while($this->resultString && !$this->endOfInput() && !$done)
{

if($this->inputString[$this->pointerInString] == '.

        $done = true;

    else

        $this->digit();

    }

}

}

function digit() {

$digitArray = array('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');

if(!$this->endOfInput())

$this->resultString = false;

elseif(array_search($this->inputString[$this->pointerInString], $digitArray) !== False)
{  $this->resultString = $this->resultString && $this->match($this->inputString[$this->pointerInString]);
}

else

    $this->resultString = false;

}
Source code for web user interface for one of the implementation is included in this section. User interface is input and output page for each demo. Source code for S-expression grammar implementation is included bellow for reference.

```html
<!DOCTYPE html>
<html>
<head>
<title>Recursive Descent Parser</title>
<style type = 'text/css'>
body {
    background-image:url('sitebg.jpg');
    background-repeat:repeat;
}
hr {
    background-color: #000;
    border: 1px solid #000;
    height: 1px;
    width='949px';
}
</style>
</head>

</html>
```
<div id="content-text" background-color:#ffffff; style="margin: 0px 40px 0px 40px; text-align: left;">  
<h3 style="text-align: center">Recursive Descent Parser Implementation</h3>  
<p> <b> Grammar for S-expression - LISP [parenthesized syntax]</b><br>
EXP ::= ( LIST ) | a <br>
LIST ::= LIST , EXP | EXP <br> </p>  
</div>  
<?php  
if(!isset($_POST['exp']))  
{  
// ask user for initial input  
?>
<?php
}
else
{

// handle parsing and display result
$userInput = $_POST['exp'];
require_once 'grammarClass.php';
$rdr2 = new RDR2($userInput);
?>
<?php
if($rdr2->isResultString())
{

<?php
<font color="green">Input String <?php echo $userInput; ?> is <strong>Valid</strong>.</font>
</p></div>

<?php

</p></div>

<?php
}
else
{

<?php
<font color="red">Input String <?php echo $userInput; ?> is <strong>Invalid</strong>.</font>
</p></div>
```php
?>
<br>
```

```php
?>
<br>
```

```php

</p>

<p>Example of some invalid strings: <br>
<code>(a,aa)$</code> <br>
<code>a(a,a)$</code> <br>
<code>(a,aa)</code> <br>
<code>(a,aa)</code> <br>
</p>

<form action="rdr2.php" method="post">
<p>Enter s-expression (nested list) invented for and popularized by programming language LISP <br>
Parenthesized syntax of Lisp, an s-expression is classically defined inductively as <br>
an atom, or <br>
an expression of the form (a, a) where a is s-expressions. <br>
</p>
Input String: <input type="text" name="exp">
<input type ="submit" value ="Submit">
<br>
<br>
<p align = 'right'> <a href = "auth.php"> Download Code </a> </p>
<form>
<br><br>
</div>
<div id="pageFooter" align = "center" width='949px'>
<br>
<br>
</div>
<img src="pageFooter.jpg" align="center" width='949px'>
</div>
</div>
</body>
</html>
BIBLIOGRAPHY

[1] Lexical and Syntax Analysis of Programming Language, Notes by Chris Northwood
[Online]*

[2] Compiling and Translation, Essays by F. D. Lewis
[Online]*


[5] CSc135 notes by Dr. Meiliu Lu, Department of Computer Science, CSUS


[7] HTTP Authentication with PHP [Online]*

[8] Parsing Regular Expression using Recursive Descent Parsing [Online]*

Available: http://www.cs.nott.ac.uk/~txa/g51mal/notes-6x.pdf

[10] Top-down Parsing [Online]*

[12] Parsing Expression Grammar as a Primitive Recursive-Descent Parser with Backtracking by Roman R. Redziejowski [Online]*

[13] Compiler-Compiler or Compiler Generator [Online]*

[14] Several Compiler-compilers [Online]*

http://en.wikipedia.org/wiki/Recursive_descent_parser

[* for all online resources, available link is given as of November 2013]