SAVITRIBAI PHULE PUNE UNIVERSITY

A PRELIMINARY PROJECT REPORT ON

Implementation of UI for LL(k) parser generator

SUBMITTED TOWARDS THE
PARTIAL FULFILLMENT OF THE REQUIREMENTS OF

BACHELOR OF ENGINEERING (Computer Engineering)

BY

Alok Singh	B120224205
Pankaj Kumar	B120224237
Rohit Rawat	B120224247
Yogesh Irale	B120224225

Under The Guidance of

Prof. S. R. Dhore

DEPARTMENT OF COMPUTER ENGINEERING
ARMY INSTITUE OF TECHNOLOGY
Digh Hills,Alandi Road
Pune-411015

ARMY INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER ENGINEERING

CERTIFICATE

This is to certify that the Project Entitled

Implementation of UI for LL(k) parser generator

Submitted by

Alok Singh	B120224205
Pankaj Kumar	B120224237
Rohit Rawat	B120224247
Yogesh Irale	B120224225

is a bonafide work carried out by Students under the supervision of Prof. S. R. Dhore and it is submitted towards the partial fulfillment of the requirement of Bachelor of Engineering (Computer Engineering) Project.

Prof. S. R. Dhore
Internal Guide
Dept. of Computer Engg.

Prof. S. R. Dhore
H.O.D
Dept. of Computer Engg.

Abstract

The visual parser generator is a tool which is developing parsers without the use of any text based grammar,script or code. The parser rules are representaion by using different symbols in the form of tree structure and each node represent a rule specified in user defined grammar. Anybody with little knowledge of parser rules and concepts can be trained to delevop their own set of grammer rules.

Users need tools for understanding and solving their problems in efficient and flexible manner. They likes to automate their tasks and generate output that can be easily applied to other applications. This application provides facilities of generation of abstract syntax tree. Along with user friendly interface this application allows the user to store their grammer as back in an XML file that can be later used for reviewing, testing, or modification of grammer.

This application accepts context-free grammars from the user for parser generation. It allows users to develop, edit and understand the working and flow of grammer languages and also facilitate them to test their own grammar that is user can made their own grammer and test on this application. This application is pretty user friendly and provide a handy graphical user interface environmnet in generating parse trees and action code generation.

The application API is quite simple to operate and can be easily understood with real examples. Some highlighted features of this includes:

- It accepts a context-free grammar.
- It accepts a regular expression as terminal symbols.
- It allows the programmers to interact with the grammar without an explicit code generation step.
- It provides an API that enables the programmers to write tree-walking code quickly and easily.

In addition, the API also allows the user to developing parsers without code or script of any kind. The grammer tree can be represented with distinct icons for various elements.

Acknowledgments

It gives us great pleasure in presenting the preliminary project report on ***'Implementation of UI for LL(k) parser generator'****.*

I would like to take this opportunity to thank my internal guide ***Prof. S. R. Dhore*** *for giving me all the help and guidance I needed. I am really grateful to them fore their kind support. Their valuable suggestions were very helpful.*

In the end our special thanks to ***Prof. Sagar Rane*** *for providing various resources such as laboratory with all needed software platforms, continuous Internet connection, for Our Project.*

Alok Singh
Pankaj Kumar
Rohit Rawat
Yogesh Irale
(B.E. Computer Engg.)

Contents

List of Figures

List of Tables

Chapter 1

Synopsis

1.1 Project Title

Implementation of UI for LL(k) parser generator.

1.2 Project Option

Internal project

1.3 Internal Guide

Prof. S. R. Dhore

1.4 Sponsorship and External Guide

NA

1.5 Technical Keywords

1. Regular Expression
2. Parser
3. Abstract Syntac Tree
4. Action Code
5. Context free grammar

6. BNF(Bottom up parsing)
7. PEG(parsing expression grammer
8. LL(k) grammar
9. First and Follow
10. Production rules

1.6 Problem Statement

To develop a visual parser generator UI as a learning tool to develop parsers without having any textual grammar specification ,code or script.

1.7 Abstract

A visual parser-generator is an Integrated Development Environment used for the development of parsers without using text based grammar, script or code. On contrast with other parser generators, it can present user given rules in the form of trees which gives a graphical view to parsel rules. It uses various icons in order to represents nodes in the tree where each node represent one rule. These generated trees can represent Parsing Expression Grammer and can work with LL(k) grammars. The executable of these code can be done by a single click of the button.
Users need tools for understanding and solving their problems in effiecient and flexible manner. They want to automate their tasks and generate output that can be easily applied into their application. This aplication provides facilities for automatic abstract syntax tree construction.
This application accepts context-free grammars for parser generation. It allows users to develop, edit, understand and test agrammar in an interactive userfriendly environment. This GUI visualizes the operations in generating parse trees and action code generation.

1.8 Goals and Objectives

To learn to process text input using both regular expressions and parser generators.

To create a learning tool which implements the processing of a regular expression using LL(k) parser generator.

The parser generator consists of user defined grammar tree i.e a collection of tokens and expressions with user defined rules,action code, .this will help out the users to learn about the parsing techniques easily.

1.9 Names of conferences/Journals where papers can be published

- STOC - *ACM Symposium on Theory of Computing*
- ICALP -*International Colloquium on Automata, Languages and Programming*
- WoLLIC -*Workshop on Logic, Language, Information and Computation*
- CIAA -*International Conference on Implementation and Application of Automata*
- CGO -*ACM SIGPLAN International Symposium on Code Generation and Optimization*

1.10 Review Of Conference/JournalL Papers Supporting Project Idea

- Incremental LL(1) Parsing in Language-Based Editors – John J. Shilling This parsing techniques used in language-based editors like making correct structuralization after changes have been by made by user. It is known as Incremental Parser because it provides facility to parse input after a certain interval so it is easy for user to make changes according to their application. Basically it is work on editor which is different from top-down parsing because a parser are made then it can never make reverse decision,for this reason in Incrmental Parser, decision are never made and this algorithm also supports real time feedback for shortest interval then according to feedback user can perform operation along with their application. It uses operations like Incremental Computation,Incremental Parsing, and Language based Editors.

- ANTLR: A Predicate LL(k) Parser Generator–T. J. P ARR University of Minnesota, AHPCRC, 1100 Washington Ave S Ste 101, Minneapolis ANTLR is useful for PCCTS.It has Many features.It generates Recusive Descent Parsers by integrating Lexical Analyzer and Abstract Syntax Analyzer in c or c++ which is easy to compaitable with another Applications.Our work on ANTLR continues,We are currently developing GUI,that might highlights conflicting Syntax Paths for Invalid Grammer.

- The Foundation of the ANTLR Parser Generator
  Parsing is not a solved problem because adding Parser speculation to Normal LL LR parser can lead to unexpected beahavior which lead to parse time increment, error handling and step by step debugging and many side effects within grammar action. It introduces a LL(*) strategy and an algorithm that makes LL(*) parsing decisions for ANTLR grammars.It also reduces the Unexpected behavior of parsing time as well handles error with Unrestricted grammar actions.

- Visualization of Syntax Trees for Language Processing Courses Francisco J. Almeida-Mart nez, Jaime Urquiza-Fuentes J. Angel Velazquez-Iturbide
  VAST provides an Aplication Programming Interface that representing the visualiztion og abstract sytnax tree as completely independant of the parser generator and it also providesan advance interface for huge trees.

1.11 Plan of Project Execution

The planning for the project will be divided in the following subtasks:

- Project title selection
- Discussion and proposal
- Proposal submission
- Literature survey
- Proposal presentation

- Proposal design
- Software development
- Test plan
- Testing and QA

Chapter 2

Technical Keywords

2.1 Area of Project

Compiler Design

2.2 Technical Keywords

- Regular Expression
- Parser
- Abstract Syntac Tree
- Action Code
- Context free grammar
- BNF(Bottom up parsing)
- PEG(parsing expression grammer (k) grammar

Chapter 3

Introduction

3.1 Project Idea

Development of visual parser generator which is used for developing parsers without the use of any text based grammar,script or code with aim of learing and understand of grammer rules and parsing of context free grammer.

3.2 Motivation of the Project

The basic idea and motivation behind developing this project is to help out the users to learn about the parsing techniques easily.Through this project we are creating a learning tool which implements the processing of a regular expression using LL(k) parser generator and help users in quickly understand grammar-trees. For this we are providing numerous menus for editing grammar-trees in parsers The purpose of this project is to provide users a menu driven tool and other editing options.

3.3 Literature Survey

This section is involved with reading and examining the IEEE papers related to the topic and what all conclusion and be drawn from it that can be used as a part in the developent of the project.

Year	Publication	Authors	Title	Algorithms	Conclusion	Limitation
2014	School of Science and Technology, UFRN, Natal, Brazil	Srgio Medeiros and Fabio Mascarenhas and Roberto Ierusalimschy	Left recursionin Parsing Expression Grammars.	Packrat parsing and virtual parsing machine.	It will provide useful meaning for PEG's with Left-Recursive rules. it is extended version which is based on Packrat Parsing with supporting Direct as well as Indirect left recursion.This extension is based on Bounded Left Recursion.	It descibes left recursive PEGs in Parsing Machine which is extended version but it does not provide support for right recursive.
2011	University of San Francisco and ATandT Labs Research	Terence Parr and Kathleen S. Fisher.	LL(*): The Foundation of the ANTLR Parser Generator.	LL(*) Parsing Strategy and Associates grammer rules analysis Algorithm.	LL(*) can accepts left-recursive grammers in which by it can recognize context-sensitive lanuages.It will also provide support for Debugging and Error Handling.It will generate DFA which can be used for Back-Tracking when it's operations gets fail.	LL's is good for Error Handling and Unrestricted grammar actions. but it can fail during Back-Tracking Operation then it becomes overhead.
2008	Conference Paper in King Juan Carlos University	Francisco J. Almeida-Martnez and Jaime Urquiza-Fuentes and ngel Velzquez-Iturbide	Visualization of abstract syntax trees within language processors courses	Lex/ Yacc/ Semantic Rules Associated API in JAVA.	VAST provides an Application Programming Interface that Representing Visualizations Of Abstract Syntax Trees completely Independent of the Parser Generator and it also provides Advance Interface	The VAST interface provides two views by which user can navigate but it has side effect of the Parser execution.

Year	Publication	Authors	Title	Algorithms	Conclusion	Limitation
2011	Journal of Software Engineering and Applications.	Nazir Ahmad Zafar	LR(K) Parser Construction Using Bottom-Up Formal Analysis.	CFG and Z notation to construct LR(K) parser and Z/Eves tool.	It will support Automatic parser generation and handles a large number of grammar classes. It will describe the actual result which we expect by real source code by Validation.	It provides Z/Eves tool which is suitable for programming pratice but there are many tools available which already provides the same meaning and working with Z is lesser effective.
2005	IEEE Transcation of Software Engineering.	Teoder Rus	Parsing Languages by Pattern Matching	LL/ LR/ Stack/ TDT/ ODT/ FIF	It can be used as driver for Code Generation and Code Optimization in Translator.It'll recognizes source language in source code. For this it is known as Program Evaluator.	There in not any natural support for structured Programming and Time Complexity is also Increases.
1995	University of Minnesota	T. J. PARR and R. W. QUONG	ANTLR: A Predicated-LL(k) Parser Generator	LR/ LALR/ Lex/ Yacc/ RDP using c++.	ANTLR is useful for PCCTS.It has Many features.It generates Recusive Descent Parsers by integrating Lexical Analyzer and Abstract Syntax Analyzer in c or c++ which is easy to compaitable with another Applications.	Our work on ANTLR continues,We are currently developing GUI,that might highlights conflicting Syntax Paths for Invalid Grammer.

Year	Publication	Authors	Title	Algorithms	Conclusion	Limitation
2009	University of Minnesota	Luis Tari, Phan Huy Tu, Jorg Hakenberg, Yi Chen, Tran Cao Son, Graciela Gonzalez and Chitta Baral	Parse Tree Database for Information Extraction.	PTDB/ PTQL/ SQL Query/ Filtering Query.	It'll provide the capabilities of managing Internal Representation such as Parse Tree semantic Information which is not provided by previous Frameworks.	Still it is not compatible with Regular Expression and it has more Redundancy in compute confidence for extracted Information.

Table 3.1: Literature survey

3.3.1 Summary

Some serious works has been done in the field of parser generation but most of them suffers from some limitations. Most of the projects discussed in the literature survey concerned with the coding part only without giving any visualisation to the user as how the code is actually going to be executed and how the parser tree looks for various grammer rules defined by users. Some projects are good but they deals with only one parsing technique (mostly works only with LL(k)). Some did mainly the code optimisation part.

Some tried to provided frameworks or GUI to the user but they are not as such user friendly and some are not compatible with regular expressions. Apart from this some gives conflicted path for invalid grammers. For some there time complexity is bit on higher side. So in this project we tried to provide a GUI based internal representation of parse tree to give symentic information wich is not provided by previous frameworks along with action code.

Chapter 4

Problem Definition and scope

4.1 Problem Statement

To develop a visual parser generator UI as a learning tool to develop parsers without having any textual grammar specification ,code or script.

4.1.1 Goals and objectives

To learn to process text input using both regular expressions and parser generators.

To create a learning tool which implements the processing of a regular expression using LL(k) parser generator.

The parser generator consists of user defined grammar tree i.e a collection of tokens and expressions with user defined rules,action code, .this will help out the users to learn about the parsing techniques easily.

4.1.2 Statement of scope

Users need tools for understanding and solving their problems in effiecient and flexible manner. They want to automate their tasks and generate output that can be easily applied into their application. This apllication provides facilities for automatic abstract syntax tree construction.

4.1.3 Software Context

Applications that have all their logic within the grammar (as actions) can be run without additional effort or coding by using the command-line runner. The runner is activated with a saved grammar file and the user-provided input-data. It instantiates a parser from the grammar, and then runs it to process the input-data. The parser's output is printed to the console.

4.1.4 Major Constraints

The development of the project is based according to the basic knowledge of the user and their curiosity of understanding the basic implement of language parser and how the parser works with the spacific grammer rules. The software is also designed keeping in mind the general understanding level of user such that they can easily modify the grammer rules and develop their own modified language .

4.1.5 Methodologies Of Problem Solving And Efficency Issues

The first step which comes under this topic is Identifying of the problem. Make a clear idea about what the problem exactly is, when this happens, how often it happen and what effect it causes. Approach it rationally and believe that every problem is solvable if it is tackled appropriately.Next step is finding appropriate solutions to the problem. Think about three - four possible ways that you can use to specify what went wrong. Adopt appropriate strategy in order to understanding the main causes for the problem and that can handle the problem in a better way also consider alternative solutions which may help in solving the problem.

Now after finding a solution for the problem next step is designing a Plan of Action and its implementation. Work with these strategy and plans. Analyse the output generated in this process and if the output is corret then the problem is solved and if the these strategy doesnt work, then go back to the list of possible strategies and select another one which may suits it better. Continue with this process untill you find a valid solution to your problem.

4.1.6 Outcome

This software will allow the users to understand how rules are parsed and how the grammer tree can be constructed for the given grammer. The required output of code can be seen in the log section of GUI. If any error occured then it will be reflected in red and rest will be in black if no error occured. Finally Abstract Syntex Tree will represent grammer rules given by users along with some memory statistics.

4.1.7 Applications

- It allows development of parsers without any code or script of any kind with user friendly interface.
- Using this application we can help people with a little set of parser knowledge to delevop their own grammer.
- It supports automatic parser generation and handles a large number of grammar classes.

It allows the programmers/users to store their own set of grammer rules for later purposes like reviewing, testing, or modification of grammer.

4.1.8 Hardware Resources Required

One or More system with JRE installed on the OS.

4.1.9 Software Resources Required

Operating System:
Linux (Ubuntu)
Linux has lots of support for all sorts of languages. Unless its notoriously known for being specific to a certain platform ,then the chances that you can get it on Linux are extremely high.
Programming Languages:
Java
A general-purpose computer programming language that is concurrent, class-based, object-oriented,[14] and specifically designed to have as few implementation dependencies as possible.
JavaScript
A lightweight, interpreted programming language. It is designed for creating network-centric applications. It is complimentary to and integrated with Java. JavaScript is very easy to implement because it is integrated with HTML. It is open and cross-platform.

Chapter 5

Project Plan

The planning for the project will be divided in the following subtasks:

- Project title selection
- Discussion and proposal
- Proposal submission
- Literature survey
- Proposal presentation
- Proposal design
- Software development
- Test plan

5.1 Project Estimates

5.1.1 Project Resources

- Linux Opearting system, with JDK installed in it.
- Scala installed on OS.
- Github account
- JRE Java Runtime Environment.
- Eclipse IDE

5.2 Risk Management

5.2.1 Risk Identification

After Thorough review of scope document, requirements specifications and schedule following risk may occur-
1.The end users may sometime not understand the full and vise use of this project.
2.This project might not provide a wide variety of customisation options to the users.
3.This project may create problem with the large data set and to many modifications.

5.2.2 Risk Analysis

Risk analysis involves the process of analysing the threat to users, market, private and public sectors, govenment agencies and other organisations. Risk analysis includes various topic in it ,some of these are assessment of risk, characterization of risk, risk communication, management related to risk and policy to handle risk.

Risk Assessment of the project involves identifying of the risk that is what are the reasons cause the risk and how they are introduced into software. We have tried to remove all such factors which can cause risk to the users.

Risk Assessment also includes evaluation and measurment of the probability and severity of risks.we have take all the measured to avoid any kind of risk and also used component that seeks to identify and measure the risks factor.

Next part which comes under risk Assessment is risk mitigation that is on occurance of the risk then what can be done to about the risks. We have taken all the primilary measures to prevent the occurance of risk and if not then tried to minimise its effect on individual. The risks for the Project can be analyzed within the constraints of time and quality.

5.3 Team Organisation

5.3.1 Team structure

Our team consists of four members, each one of them, able to work on one another domain.
Different profiles in our project where:
Developer
Tester
Project Manager
Analyst.

5.3.2 Management reporting and communication

Our project guide is Prof. S. R. Dhore (HOD Computers). We discussed the idea of the project with our guide. Different new approaches and guidelines were provided by him.
First Discussion : Dicusson of the Project idea and its scope.
Second Discussion : Discussion on different modules of the project and approaches involved with it.
Third Disucssion : Presentation of the project.

Chapter 6

Software requirement specification

6.1 Introduction

An SRS minimizes the time and effort required by developers to achieve desired goals and also minimizes the development cost. A good SRS defines how an application will interact with system hardware, other programs and human users in a wide variety of real-world situations. Parameters such as operating speed, response time, availability, portability, maintainability, footprint, security and speed of recovery from adverse events are evaluated.

The Software Requirements Specification (SRS) is a communication tool between stakeholders and software designers. The specific goals of the SRS are:

- Facilitating reviews.
- Describing the scope of work
- Providing a reference to software designers
- Providing a framework for testing primary and secondary use cases
- Including features to customer requirements
- Providing a platform for ongoing refinement (via incomplete specs or questions).

Figure 6.1: Specification Types

6.1.1 Purpose and Scope of Document

Purpose:
Purpose of an activity, project or procedure represents the reason for the change, induction or migration in a brief way.
Scope of an activity, project or procedure represents their limitations or defines the boundaries of its application.
The purpose of this SRS is to describe the details of the parser generator application. Moreover, this document explains the purpose of each modules of generator, the features of said modules, and the interactions between modules, helps users to learn about the parsing techniques easily.

6.1.2 Overview of responsibilities of Developer

- Analysis: resulting in models, schema, and business rules.Examining the IEEE research papers.
- Design: resulting in the software architecture
- Coding: the development, proving, and integration of software
- Testing: the systematic discovery and debugging of defects
- Operations: the installation, migration, support, and maintenance of complete systems

6.2 Usage Scenario

- Represent grammar rules in the form of grammar tree.
- Modify grammer tree(with nodes) according to the need using terminal and non terminal symbols.
- For grammar tree, mention the action code associated with each node.
- Take regular expression as input, process the regular expression on the grammar tree created.

6.2.1 Use Case Diagram

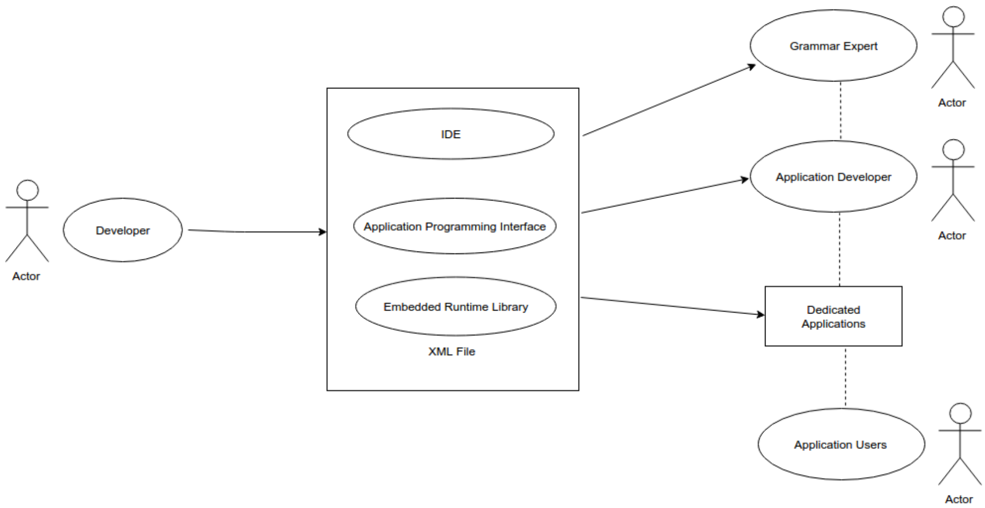


Figure 6.2: Use Case Diagram

6.3 Set Theory

The LL(k) languages comprise the largest class of context-free languages that can be parsed deterministically from the top down. The LL(k) grammars generate the LL(k) languages, so the LL(k) class of grammars is of at least theoretical interest in any consideration of top-down parsing methods. In practice, LL(k) parsing techniques for values of k that are greater than one have generally not been used.

This is because both the size of the parser and the complexity of the grammar analysis grow exponentially with k. Even in the LL(1) parsing method, the size of the tables, and so the size of the parser is a matter of some concern. That is why much attention has been focused on table compaction methods for LL(1), as well as for other parsing methods. Given that the LL(1) parse table is considered uncomfortably large, it is not surprising that the size of the parse tables for LL(k) grammars for values of k that are greater than one has been considered prohibitively large.

Even so, there has been a small body of work devoted to the parsing of LL(k) grammars. Aho and Ullman (1972) give a completely general method of analyzing and parsing LL(k) grammars. The price paid for this generality is impracticality in many cases. The computational complexity of the method is too great for most grammars of useful size. Yoshida and Takeuchi (1992) present a more practical method for parsing a subset of the LL(k) grammars relatively efficiently. Most current parser generators use an approximate lookahead method that has linear complexity, but very weak recognition capability.

6.3.1 Definition of LL(k)

The LL(k) methods produce a leftmost parse by examining the input string from left to right. Parsing decisions are made based on the two criteria that follow:

1. The already-parsed prefix of a sentence in the language
2. The next k input symbols

That is, the production that is used to expand the current nonterminal that is being parsed is chosen based on a knowledge of the string of terminal symbols that has been produced thus far by the parse, in conjunction with a knowledge of the next k input tokens to be encountered in the input string. This can be represented symbolically as in the following definition:

Definition: A grammar G = (N, T, P, S) is said to be LL(k) for some fixed natural number k when for any two leftmost derivations in the grammar as in the following:

1. $S \Longrightarrow_{lm}^{*} x\ A\ \mathit{delta}_1 \Longrightarrow_{lm} x\ \mathit{omega}_1\ \mathit{delta}_1 \Longrightarrow_{lm}^{*} x\ y_1$
2. $S \Longrightarrow_{lm}^{*} x\ A\ \mathit{delta}_2 \Longrightarrow_{lm} x\ \mathit{omega}_2\ \mathit{delta}_2 \Longrightarrow_{lm}^{*} x\ y_2$

if

$$\mathrm{FIRST}_k\ (\ y_1\ ) = \mathrm{FIRST}_k\ (\ y_2\ )$$

it implies that

$$\mathit{omega}_1 = \mathit{omega}_2$$

In the general case, the LL(k) grammars are quite difficult to parse directly. This is due to the fact that the left context of the parse must be remembered somehow. As can be seen from the definition, each parsing decision is based both on what is to come as well as on what has already been seen of the input. How then is it that the LL(1) grammars are so easily parsed? The answer is that the class of LL(1) grammars is strong. The strong LL(k) grammars are a subset of the LL(k) grammars that can be parsed without knowledge of the left-context of the parse. That is, each parsing decision is based only on the next k tokens of the input for the current nonterminal that is being expanded. A definition of the strong LL(k) grammars follows.

Definition: A grammar G = (N, T, P, S) is said to be strong LL(k) for some fixed natural number k if for all nonterminals A, and for any two distinct A-productions in the grammar

A --> *alpha*
A --> *beta*

$$\mathrm{FIRST}_k\ (\ \mathit{alpha}\ \mathrm{FOLLOW}_k\ (A)\ )\ \mathit{intersection}\ \mathrm{FIRST}_k\ (\ \mathit{beta}\ \mathrm{FOLLOW}_k\ (A)\ ) = \emptyset$$

As can be seen from the definition, the strong LL(k) grammars are defined in terms of FIRSTk and FOLLOWk sets only. These sets consist entirely of parts of a derivation that are to come, not on any part of a derivation that has already been completed. Definitions of the FIRSTk and FOLLOWk sets are as follows.

```
Definition:  The FIRSTk set of a string of symbols in
a grammar is a set of k-length strings of terminal symbols that may begin
a sentential form derivable from the string of symbols in the grammar.
More specifically, for a grammar G = ( N, T, P, S )

FIRSTk ( alpha )  =
    { w  |  alpha  ==>*  wbeta }  union
    { epsilon  if alpha  ==>*  epsilon }
where
    alpha  in ( N union T )+
    w  in T+
    beta  in ( N union T )*

For the case of the empty string:

    FIRSTk ( epsilon )  = { epsilon }

Note that the length of w may be less than or equal to k, in which case

    beta = epsilon.
```

```
Definition:  The FOLLOWk set of a string of
symbols in a grammar is a set of k-length terminal symbol strings in
the grammar that may follow the string of symbols in some
sentential form derivable in the grammar.
More specifically, for a grammar G = ( N, T, P, S ):

FOLLOWk ( beta )  =
    { w  |  S  ==>*  alpha beta gamma and w in FIRSTk ( gamma ) }
    union
    { epsilon  if  S  ==>*  alpha beta }
where
    alpha, gamma  in ( N union T )*
    beta  in ( N union T )+
    w  in T+
```

Why is it that the LL(1) grammars have the strong property while LL(k) grammars for values of k that are greater than one may or may not be strong? Briefly, the problem lies in the nature of the FOLLOWk sets. A string is included in a FOLLOWk set if it can follow another string in some context. With a lookahead string of more than one symbol, the same lookahead string may occur in more than one context when a null production is applied in one case and not in the other. The string may be a valid FOLLOWk string in one context, but not in another. This problem cannot occur with FOLLOW strings that are of length one, so that all LL(1) grammars are strong LL(1). This makes the LL(1) class of grammars particularly easy to parse. Consider the following grammar as an example:

$$A ==> aBaa$$

$$A ==> bBba$$

$$B ==> b$$

It is LL(2), but not strong LL(2). It can be seen by inspection of the grammar that:

$$FOLLOW2(B) = aa, ba$$

So if the grammar is strong LL(k), then the lookahead "ba" should uniquely predict one or the other of the two B-productions in the grammar because nonterminal B is nullable. In fact, the lookahead "ba" predicts both of the last two productions in the grammar. If production 3 is used in production 1 to expand nonterminal B, the lookahead is "ba". If production 4 is used in production 2 to expand nonterminal B, again the lookahead is "ba". The problem is that in the first case, the lookahead string is constructed from a concatenation of terminal b from production 3 and terminal "a" from production 1. Thus a lookahead string that does not apply to production 1 is constructed. That is, the lookahead string "ba" cannot follow nonterminal B in production 1. The only lookahead that can follow B in production 1 is string "aa". This impreciseness in the FOLLOW2 set is what causes Grammar 4.1 not to be strong LL(2).

The solution to this problem is to use context information to know which production to use. If production 1 is being parsed and the lookahead is "ba", then use production 3 to expand B. If the parsing context is production 2 and the lookahead is "ba", then production 4 must be used to expand nonterminal B. The definition of LL(k) grammars clearly handles this type of construction by using the information available from the left-context of the parse. In this case, only the single terminal symbol that precedes nonterminal B in the first two productions is needed to determine the correct prediction. In the general case, the entire prefix of the parse may be required in order to know which production to use.

6.3.2 Strong LL(k) Parsing

The standard LL(1) method of parsing can be applied to any strong LL(k) grammar, with only slight modification. This is because the strong LL(k) languages for values of k that are greater than one are also predictive in the same way that the LL(1) languages are predictive, without regard to the left-context of the parse. This means that given a lookahead string of length k, the choice of the next production to be used in the parse can be always be determined uniquely. So, strong LL(k) grammars for any value of k can be analyzed and parsed in the same way, except for the fact that the length of the lookahead string is equal to the k-value of the grammar. In the strong LL(k) method, FIRSTk and FOLLOWk sets are used to construct a parse table of the form:

$$Nxw|winT * and|w| = k$$

The rows of the parse table are the same as in the LL(1) construction. However, the columns of the parse table consist of all k-length strings of terminal symbols that are k-length permutations of the terminal set. The method used to construct the strong LL(k) parse table is identical to the method that is used to construct the LL(1) parse table, except that the FIRSTk, FOLLOWk and PREDICTk sets are used instead of the FIRST, FOLLOW and PREDICT sets. The PREDICTk set is defined in terms of the FIRSTk and FOLLOWk sets in a manner very similar to the way that the PREDICT set is defined in terms of the FIRST and the FOLLOW sets. At this point, it should be apparent that the FIRST, FOLLOW and PREDICT sets are just the FIRSTk, FOLLOWk and PREDICTk sets for the case where k = 1. The definition of the PREDICTk set follows.

Definition: The **$\mathrm{PREDICT}_k$ set** for a production in a grammar is a set of terminal symbols as follows.

```
PREDICT_k ( A --> alpha ) =
    if epsilon in FIRST_k ( alpha )
        ( FIRST_k ( alpha ) - epsilon )  union  FOLLOW_k ( A )
    otherwise
        FIRST_k ( alpha )
```

The LL(1) parsing algorithm only needs to be modified slightly to make it applicable to strong LL(k) parsing. The lookahead needs to be changed so that it is a k-length string of tokens, rather than a single terminal symbol. Also, the next production to be used to expand the current nonterminal symbol is obtained from a strong LL(k) parse table. The table look up parameters are the nonterminal symbol and the current k-length lookahead string. Even though a k-length lookahead string is used to make the parsing decisions, the input string is still scanned a only single token at a time as in the LL(1) parsing method. The strong LL(k) parsing algorithm is shown in figure 4.1.

```
Get the first k input tokens as a lookahead string
Push the start symbol of the grammar onto the stack

While the stack is not empty and input remains
    Pop a grammar symbol from the stack.
    If the symbol is a terminal
        Match it against the first lookahead token
        Get the next input token and append it to the lookahead string
        Advance the beginning of the lookahead string by one token
    Else if the symbol is a nonterminal
        Get the next production number from the parse table
        Push that production onto the stack
    Else if the symbol is an action
        Perform the action
End while
```

6.3.3 Strong LL(k) Complexity

The complexity of the strong LL(k) parsing algorithm is within a constant factor of the complexity of the LL(1) parsing algorithm. This factor depends on k, but since k is not an input to the parsing routine itself, k can be considered to be a constant in this case. The additional complexity is due to the fact that the lookahead is a string, rather than just a single terminal symbol. The lookahead string must be analyzed somehow in order to convert it into an index into the columns of the parse table. The complexity of this analysis is proportional to the length of the lookahead string, k. An efficient numbering all of the k-length permutations of the terminal set is not difficult to imagine. Similarly, given a k-length input string, it should not be too difficult to map that string to the proper index by analyzing its component terminal symbols one by one. However, this is still an amount of work that must be done in addition to the work that is done by the LL(1) parsing routine, so the complexity of strong LL(k) parsing is somewhat greater than that of LL(1) parsing.

The strong LL(k) parsing algorithm examines k symbols at a time of the input string. Since the input pointer is advanced only one symbol at a time, this means that the input string is examined approximately k times. However, since k is a constant in this case, the complexity of the strong LL(k) parsing algorithm is still linear in the length of the input as follows.

$$W(n) = O(n)$$

The complexity of the analysis of a strong LL(k) grammar for values of k that are greater than one is significantly greater than the complexity of the analysis of LL(1) grammars. The number of possible k-length lookahead strings is exponential in k as follows.

$$|T|k$$

This means that the computation of the FIRSTk and the FOLLOWk sets will also be exponential in k. Since k is one of the inputs to the problem of analyzing an strong LL(k) grammar, it follows that strong LL(k) analysis is exponential in the size of the input. This can also be seen by inspection of the size of the strong LL(k) parse table, which is

$$|N||T|k$$

The problem with objects that are of exponential size is that their size grows very rapidly as the exponent increases. This is especially true in the case of the strong LL(k) parse table. For many typical applications, the size of the parse table that is required for strong LL(k) is essentially unmanageable for any k greater than one. This is because the number of columns in the parse table is the number of terminals in the grammar raised to the power k. For the case of a typical programming language having about 80 terminal symbols, the parse table size increases by a factor of about 80 for the strong LL(2) parse table as compared to the LL(1) parse table. Even with compaction methods applied, this size is generally considered too large to be practical for most compilers.

6.3.4 Data Description

1. PackratParsers
PackratParsers is a component that extends the parser combinators provided by scala.util.parsing .combinator.Parsers with a memoization facility (Packrat Parsing).
Packrat Parsing is a technique for implementing backtracking, recursive-descent parsers, with the advantage that it guarantees unlimited lookahead and a linear parse time. Using this technique, left recursive grammars can also be accepted.

2. Parsers
A component that provides generic parser combinators. There are two abstract members that must be defined in order to produce parsers: the type Elem and scala.util.parsing.combinator .Parsers.Parser. There are helper methods that produce concrete Parser implementations.A Parsers may define multiple Parser instances, which are combined to produced the desired parser. The type of the elements these parsers should parse must be defined by declaring Elem (each parser is polymorphic in the type of result it produces).
There are two aspects to the result of a parser:
(i)success or failure
(ii)the result.

6.4 Functional Model and Description

6.4.1 Data Flow Diagram

The above class diagram represents the interection between vaious parts of the GUI. It represent how different classes are utilised by GUI to process the user input and generate output. This class diagram shows Parser part(includes RegexParser, PackratParser) constituting main API.
The diagram shows SimpleLexingRegexParsers consists of a lexical analyzer. The lexer or lexical analyzer improves the working of RegexParsers and also provides other features which can be seen in full-fledged parsers.

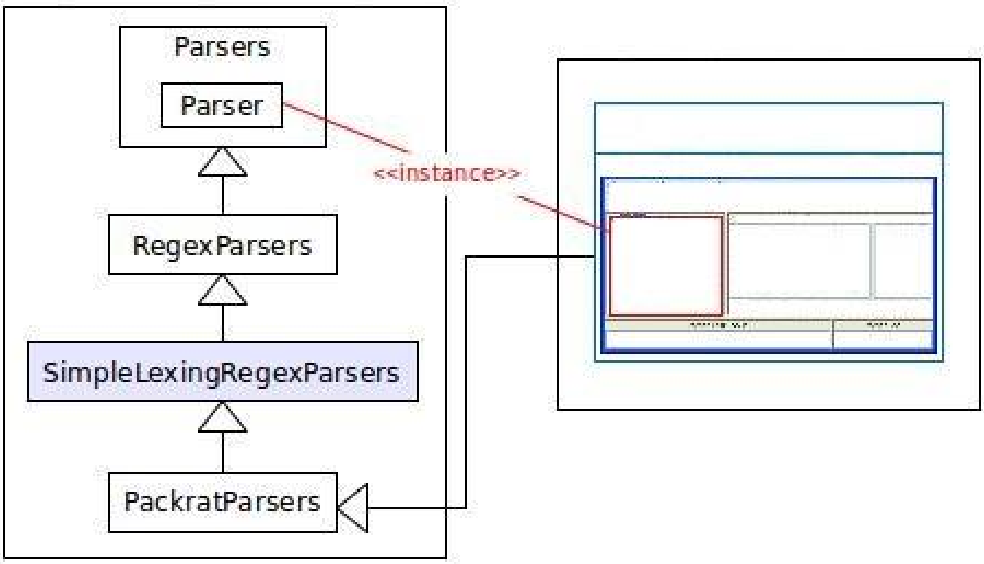


Figure 6.3: Class Diagram

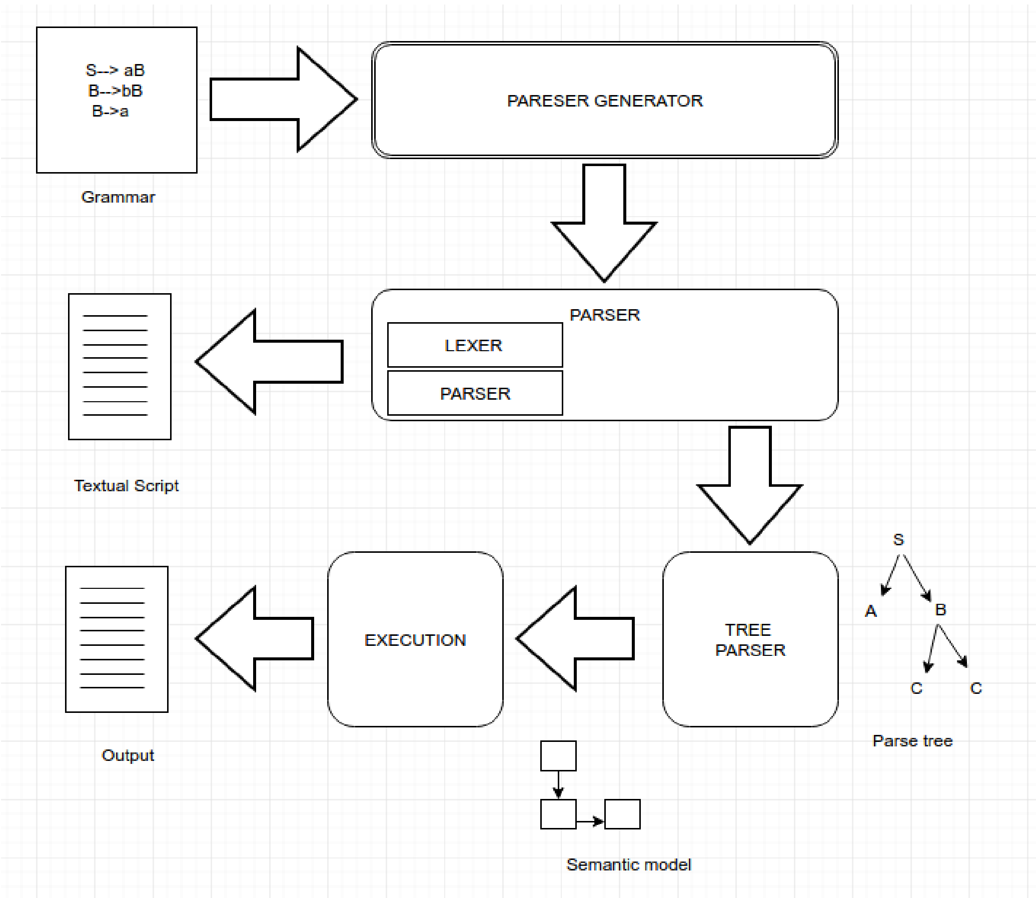


Figure 6.4: Data Flow Diagram

Chapter 7

Detailed Design Document

7.1 Introduction

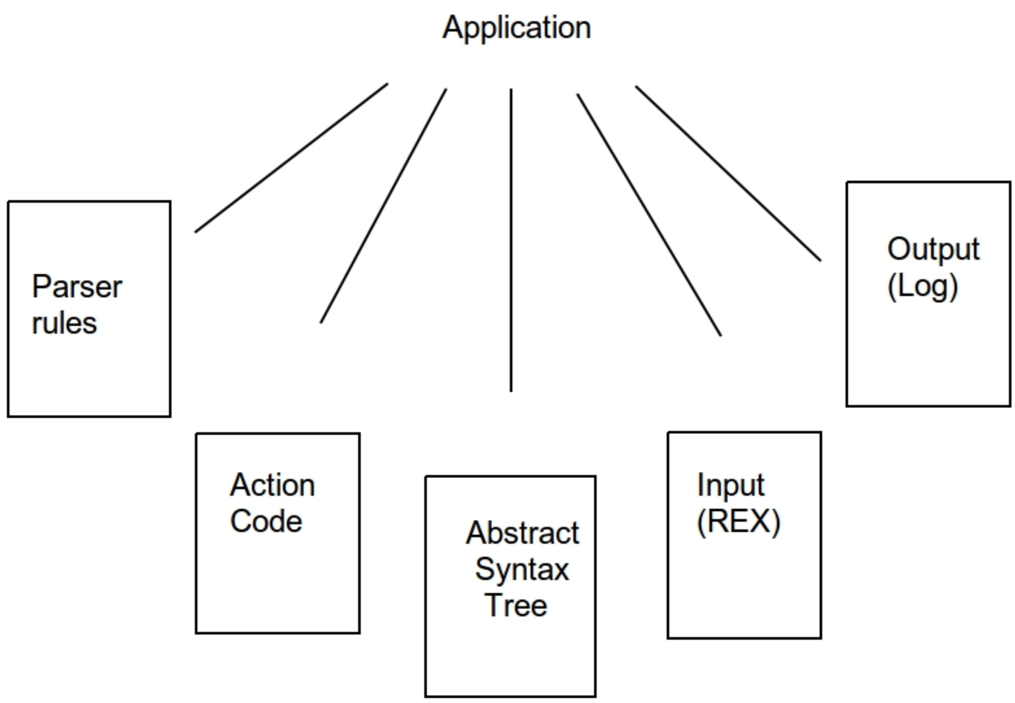


Figure 7.1: Modular Architecture

The application is mainly divided into five modules- Parser rules, action code, abstract sytax tree, Input: Regular expressions , Output: Log.

Parser rules are represented in the hierarchical tree structure with each nodes associated with some symbol and optional action code. Action code is the function taking two or more arugments as input for the required node of the grammar tree and implementing it whenever a regular expression is parsed.

Abstract syntax tree is a data structure that will be created according to the grammar rules specified. Input consists of a regular expresssion with combination of operators and operands. Output consist of a log generated for the inputed reglar expression.

7.2 Architectural Design

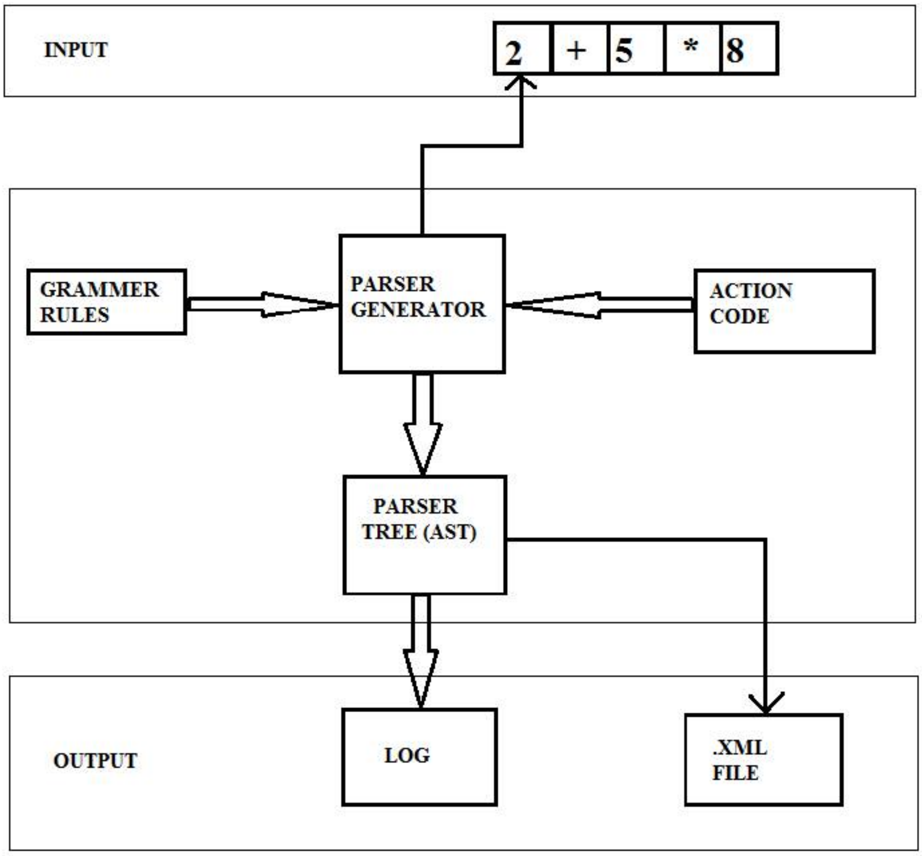


Figure 7.2: Architecture Design

The architectue of the application consist of three important components-input, output and the parser generator. Parser generator is the important component responsible for all operations. It takes grammar rules represented in the form of grammar trees. For each node, an optional action codes is assocated which specifies what operation needs to be performed when any regex will be parsed according to that gramar tree.

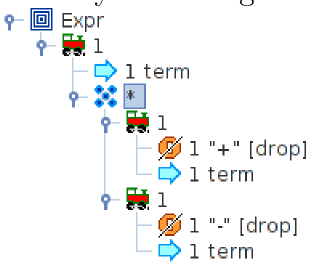
Using the help of the Parser combinators, AST is created. The ast tree helps in parsing the reruglar expression taken in the form of input to be parsed. After parsing, log is created according to the output. The grammar rules are there after are stored in the backend in a xml file, which can be used for futer reference and can be modified and updated.

7.2.1 Illustration

Consider a regular expression

```
3 + 4 - 2 * 8 + 6
```

Initially create a grammar tree according to the specific grammar rules.



For each and every node of the grammar tree, there is an asscociated optional action code. The action code specifies what action to be taken or what the expreesion sould have the value at the particular node.

```
function (arg)
 {
  if (!arg)
 {
    return;
  }
  var term = arg[0].doubleValue();
  var list = arg[1];
  for (var i = 0; i < list.size(); ++i)
 {
    var pair = list.get(i);
  }
  return term;
}
```

According to the grammar rules, the parse tree or the Abstract Syntax tree is genereated which is displayed at the right side of the IDE. It shows how the expression will be evaluated structurally.

```
List(
|   Choice(
|   |   Array(0, @term),
|   |   Array(1, @term)
|   )
)
```

After the execution of the expression, a log is generated which contains the solution of the regular expresion inputed in the input text area and a pareser log.

```
Array(Array(Array(0, 3), List()), List(Array(0, Array(Array(0, 4),
ray(Array(0, 6), List()))))

Array(Array(Array(0, 3), List()), List(Array(0, Array(Array(0, 4),
ray(Array(0, 6), List()))))

Array(Array(Array(0, 3), List()), List(Array(0, Array(Array(0, 4),
ray(Array(0, 6), List()))))

Array(Array(Array(0, 3), List()), List(Array(0, Array(Array(0, 4),
ray(Array(0, 6), List()))))
```

7.3 Data design

7.3.1 Internal software data structure

Action Code:
Action code part basically define how the code for a particular grammer rule looks like. Whenever a new rule is defined by user than the user has to write correspoing code for that rule explaining its flow . This action code defines a function which accept an argument to process input intered by the user. User has to explicitly write code for every new rule.

Saved Grammar File:
The rules written by the user in the software can be saves for further use and for learning purposes. For this we are making an XML file which will stored the grammer used by the user. Basically we are not storing code in these XML file rather they contains parser related content only which can be utilised in .

Parser:
This part of the sotware contains a program which is used to parse a given text in a format. Graphical User Interface works as a parser because it is able to parse textual information when Testing Parser. A program which loads a source file containing grammar in it also act as parser.

7.3.2 Global data structure

Abstract Syntax Tree:
An Abstract syntax tree is a tree which representation syntactic structure of the source code written in a programming language. Every node in a tree denotes a symbol occurred in the code. The syntax is "abstract" in not representing every detail appearing in the real syntax. For instance, grouping parentheses are implicit in the tree structure, and a syntactic construct like an if-then-else expression can be represented by a single node with three branches.

7.4 Component Design

Parser Rules
Parsers uses parser rules in order to recognizing strings in the language. Generally a parsing expression grammer consists of various elements like a finite set nonterminal symbols, a finite number of terminal symbols, finite set P of parsing rules and the starting expression. Parsing expression generally look like a hierarchical expression similar to a regular expression.

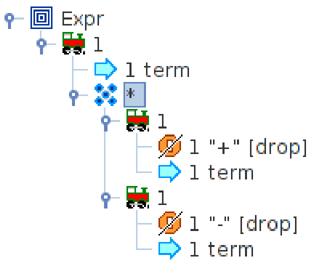
Each parsing rule in parsing expression is of the form S Ae or S eA, where S is a starting symbol, A is a nonterminal symbol and e is a parsing expression.

Parser rules consist of a set of parser rules for parsing grammer or regular expression. In this project our main focus is to use LL(K) parser. An LL parser is a top-down parser.It works with subsets of context-free languages. It parses the input from Left to right. LL(k) parser uses k tokens of lookahead when parsing a sentence.

In every step, the parser processes the next symbol available in the input string and the top-most symbol from the stack. If the value of input string and the stack-top symbol become equal, the parser discards both of them, leaving only the unmatched symbols in the input stream and on the stack.

In order to fill the parsing table, we have to decide which grammar rule the parser should choose if it sees a nonterminal A on the top of its stack and a symbol a on its input stream. To simplify and make it user friendly we have made a parse tree which specifies the paser rule specified by the user so that user can see it clearly and understand it in a good way.

Each parse tree has a root note which describe the main rule which have a sub section describing the sub rule and further sub divided accordingly as the rule says. For every rule a specific label is assigned to it so that each parser rule can be distinguished from othere rule.



Action Code
The Action code part in this software is basically the code form representation of the rule defined by user during working with regular expressions.

The function defined by the user in action code accepts only one argument variable that cab be used for two purposes. In first case one part return the value generated by parser during the execution of regular expression and in second case the other value returned is null which means nothing is processed by the parser.

The function in the action code is invoked twice : one when parsing of the related node starts and other when parsing of the tree nodes end. Any startup tasks can be performed by the function in first iteration and in second iteration it processes syntex tree part. In order to differenciate between two function call type it compares the values returned by function argument. Initially the value of argument is set to null but on second iteration a valid value is assigned to it. For proper functioning the function in the action coden must process the value of the argument correctly and then act accordingly.

Parse (AST) Tree
An abstract syntax tree (AST), or just syntax tree, is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. The syntax is "abstract" in not representing every detail appearing in

```
function (arg)
 {
  if (!arg)
 {
    return;
  }
  var term = arg[0].doubleValue();
  var list = arg[1];
  for (var i = 0; i < list.size(); ++i)
 {
    var pair = list.get(i);
  }
  return term;
}
```

Figure 7.3: Action Code

the real syntax. For instance, grouping parentheses are implicit in the tree structure, and a syntactic construct like an if-condition-then expression may be denoted by means of a single node with three branches.

The Abstract Syntax Tree structure for any given grammar can be determined by the multiplicity, arrangement, type and various other features of the nodes of that grammar tree. Beginning with the root node, the given rules can be applied to the grammer tree iteratively :

The Root node of a container as its tree is similar to that of the contained tree.
A tokens belonging to the given string, matching with the given pattern

A Regular expression token belongs to a string and a section of the input string is similar to this specified pattern – is it often represented as lexeme.

A pattern belonging to an object structure containing the items belongs to the child nodes of the tree.

The first case belongs to an array of two objects in which the first member is boxed int which symbolises
the (0-based) index of the matching alternative, and the second member is

the abstract syntex tree contituted by the similar alternatives.

A RepSep belongs a object list containing the abstract syntex tree of all similarly placed items
Containing element's AST represents nothing to the SemPred.

```
List(
|  Choice(
|  |  Array(0, @term),
|  |  Array(1, @term)
|  )
)
```

Input–Regular Expression

```
3 + 4 - 2 * 8 + 6
```

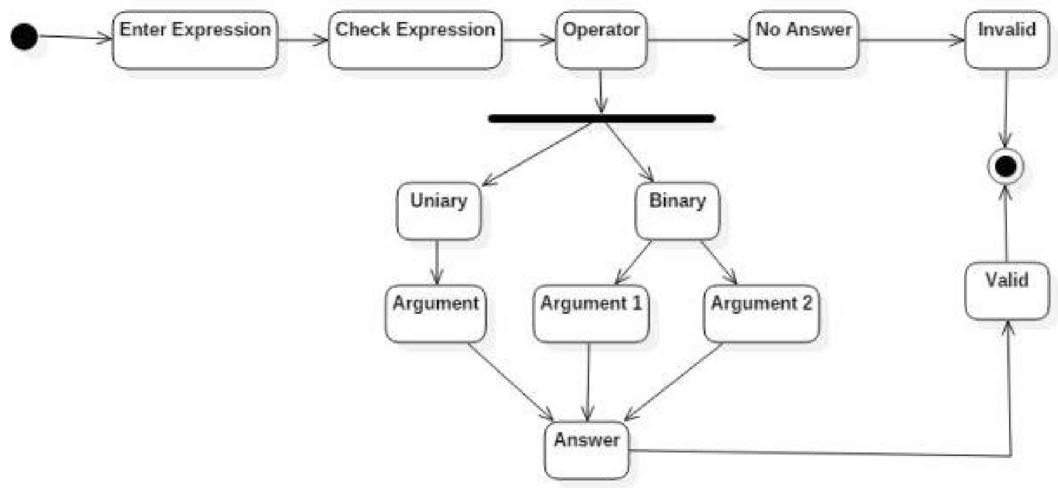


Figure 7.4: Regular Expression Flow

Parser Log

The generated result of the testing can be seen at the Parser Log area. We can modify the given input in the script and simulate a failure position.This technique can be extended to run an exhaustive set of tests. Parser-rules designed for testing with this technique are automatically distinguished with a special icon in the toolbar's dropdown list.

```
Array(Array(Array(0, 3), List()), List(Array(0, Array(Array(0, 4),
ray(Array(0, 6), List())))
```

```
Array(Array(Array(0, 3), List()), List(Array(0, Array(Array(0, 4),
ray(Array(0, 6), List())))
```

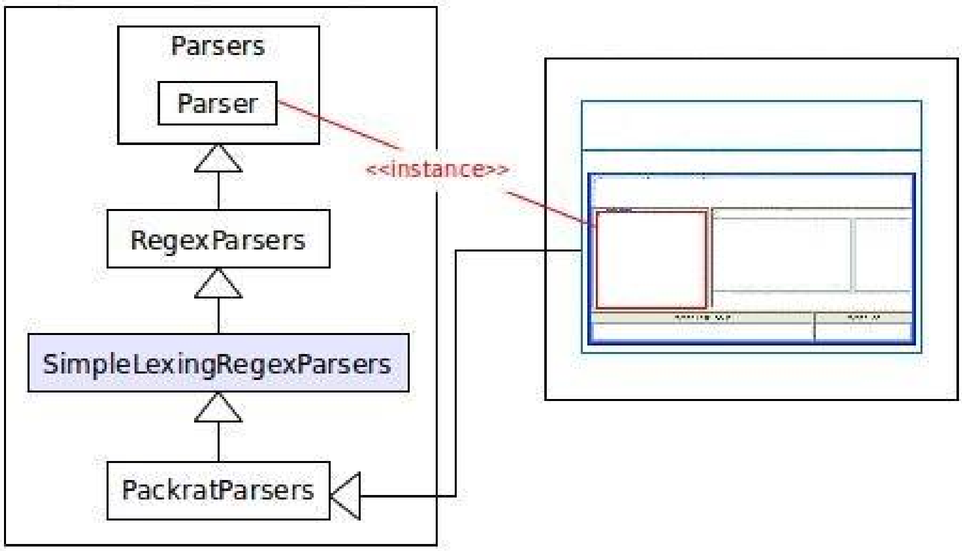
```
Array(Array(Array(0, 3), List()), List(Array(0, Array(Array(0, 4),
ray(Array(0, 6), List())))
```

```
Array(Array(Array(0, 3), List()), List(Array(0, Array(Array(0, 4),
ray(Array(0, 6), List())))
```

Lexer

Tokens has been divded into two parts i.e. literals regex.
Literal tokens is disigned in such a way that it's precedence covers regex tokens. If same input is get matched by literal and regex,the lexer would return the literals but not the regex itself.
The tokens which is get matched longest piece of input always defeats another one. Literals always designed in such a way that have precedence over regexs and only when the matching lengths of RE are equal.
There are no scope tokens, but a workaround which is available for most situations by using them.
All tokens, literals and regex's, have to be declared before using it as described in Creating Tokens.

7.4.1 Class Diagram



The above class diagram shows the flow of combinator classes how the Graphical User Interface uses. The design of Parser Regex Parser and PackratParsers is based on API used by Scala.

The Class Simple Lexing Regex Parser consist a lexical analyzer.It improves simple functio of Regex Parser i.e. literal() and regex() and provides characteristics like Full Fledged lexer.

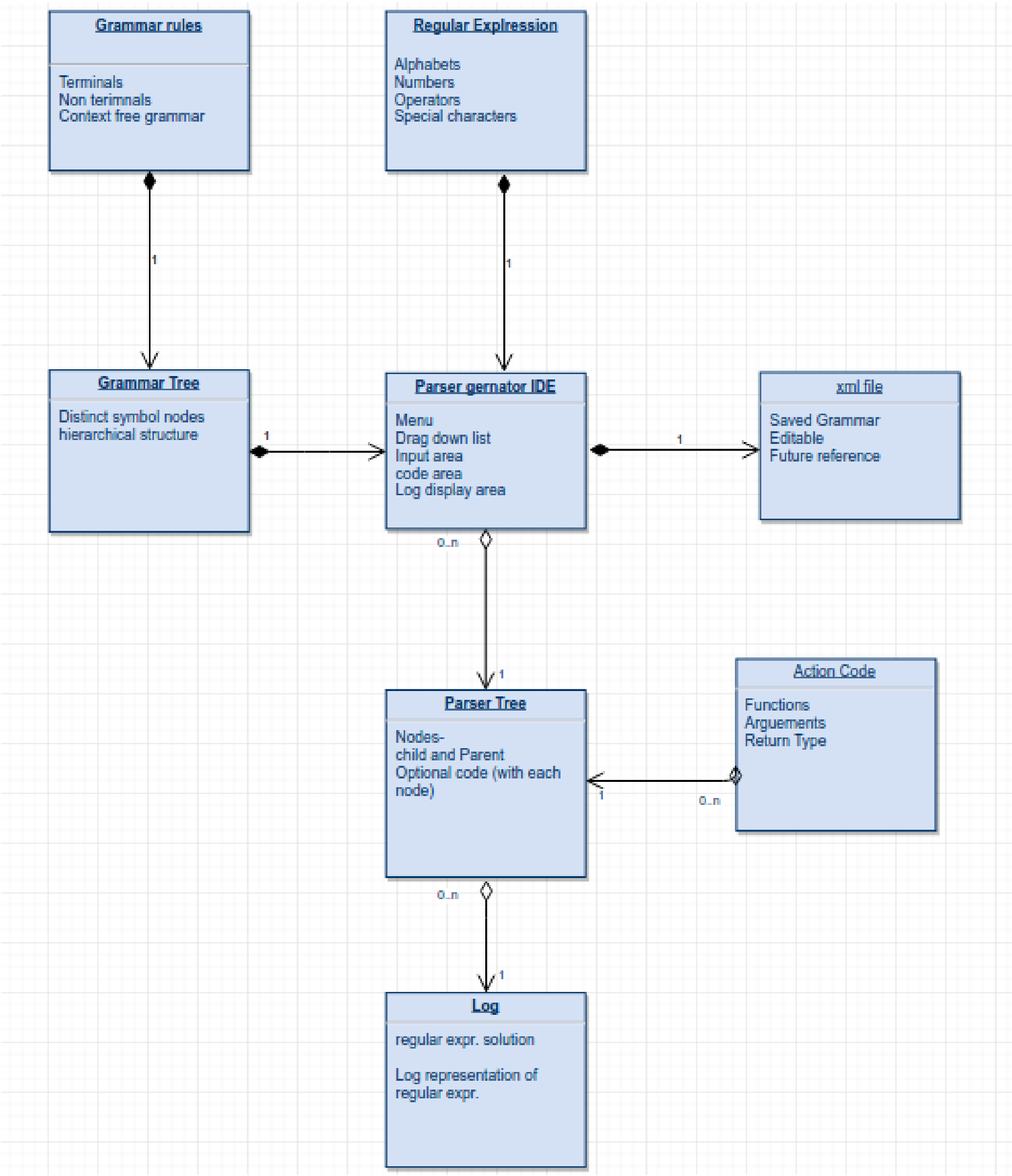


Figure 7.5: Data Flow

7.5 The GUI

7.5.1 Menu bar

The Menu Bar that needs to be designed will have the following containing menus:

- File - Its aim to organize different operations. When all the operations has been done by developing and testing the parser,the grammar file which contains an eXtensible MarkUp Language representation of the Graphical User Interface which contains entered data.Grammar File which contains .vll extensions can be loaded from a user program by using Application Programming Interface.
- View It's aim to support dintinct operations like visual grammar-tree representation with additional details which would be easy for user to corporate with parser.
- Tokens - Tokens can be generated by using Lexical Analyzer which use's literals, regex.These tokens can be updated by editing tools.Find can be used to represent parser-rules pointing towards to a particular token.User can import as well as export token libraries by using Import and Export items.
- Rules - Existing rule can be updated according to user's application like new rules can be added for a particular node in grammar-tree as well as name can be also changed by giving another name.
- Globals - It provides details about whitespace,comments and the user's other query for a given parser.
- est - Parser can be run by taking parse input and file which contains regular expression.According to derived grammar-tree, it will select choice from tree and decide the choice for Abstract Stntax Tree.
- Log - It is located at the right bottom of the Graphical User Interface which operats two different operations like Copy and Clear.
- Help - Gives access to documentation, sample grammars.

7.5.2 Tool bar

- File It have different operations like New for creatinf new file, Open for opening a already existed file,Save for current saving purpose and SaveAs would provide file name at the file saving time.

- BackButton is used to show that how Grammar Tree is represented.
- Combo box is used to select rules by accessing node's of given Grammar Tree.
- Rules Existing rule can be changed like adding some rule or it can be optimized according user's application.
- Tokens Tokens can be generated by Lex by using literals,regex.These tokens can be imported or exported from another file.
- Test It takes RE as parse input for execution as well as it can stop parsing operations at any time.
- Log It provides two operations i.e. Copy log Clear log.

7.5.3 Grammar Tree

The Grammar Tree will be at the top left of GUI. It provides structure for given a parser with the help of parser rule by presenting node and these node be accessed by userand it will also supports grammar-tree editing functions.

7.5.4 AST Area

The Abstract Syntax Tree which is known as Parse Tree is placed at right of the visual grammar-tree. It's structure is represented by the grammar tree nodes which is easy to understand for larger Parse Tree.

7.5.5 Action Code Area

It has textual representation which is placed at right top.Code for grammar-tree nodes is written by user's according to their application by writing java code.

7.5.6 Testing Area

There are two area at the bottom of the Graphical User Interface which represents tesing area where Regular Expression is used as test input for the parser.After successfully loading the file which contains input is processed by clicking a button then it generats output in the Log area.Error message for Invalid input or any kind of of reason which is invalid through the parser is represented in red color otherwise normal outcome is represented as black.It also calculates run time as well as Meomory used by program during execution which is also represented along with output.

Chapter 8

Project Implementation

8.1 Introduction

The development of the application involves creation of an user interface oriented IDE that will help the user to learn grammer rules. Implementation of any application involves direction oriented work process. Inorder to accomplish this part of the application, we are building the modules one by one and testing them individually as well as intergrated or whole.

8.2 Tools and Technologies used

Tools:
Elipse IDE: A new project is created under the name of LL(K) parser generator. This project has three different sections- api, combinators and GUI.

Firstly the GUI, it contains all the files that are required for the creation of the IDE and the user interaction thing. Here all the action that needs to be done from the user side is taken care, may be it would be creating the grammar or displaying the output.

The combinators, these contains files that are required for the parser generation.It involves parsing,creation of tokens, checking the input string etc. and various functions that involves the grammar rules.

The API, here consists the files that helps the action to be performed by the user.It is between the GUI and the combinator. These API helps in performing the operations on the input specified by the user and the operations performed by the combinators that need to be dislayed to the user side.It gives a way in which it provides various options to the user for handling the combinator output.

Technolgies:
Java AWT:
AWT is heavyweight i.e. its components are using the resources of OS.The java.awt package provides classes for AWT api such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

Languages:
Scala:
Scala is a type-safe JVM language that incorporates both object oriented and functional programming into an extremely concise, logical, and extraordinarily powerful language.It relieves many of the age-old, time-consuming Java programming headaches.
Javascript:
Javascript has advantages in terms of speed, simplicity,versatility and server load.

8.3 Methodologies/Algorithm Details

8.3.1 Algorithm :LL(k) parsing

An LL Parser accepts LL grammar. LL grammar is a subset of context-free grammar but with some restrictions to get the simplified version, in order to achieve easy implementation. LL grammar can be implemented by means of both algorithms namely, recursive-descent or table-driven.

LL parser is denoted as LL(k). The first L in LL(k) is parsing the input from left to right, the second L in LL(k) stands for left-most derivation and k itself represents the number of look aheasds. Generally $k = 1$, so LL(k) may also be written as LL(1).

We may stick to deterministic LL(1) for parser explanation, as the size of table grows exponentially with the value of k. Secondly, if a given grammar is not LL(1), then usually, it is not LL(k), for any given k.

```
Given below is an algorithm for LL(1) Parsing:


Input:
   string ω
   parsing table M for grammar G

Output:
   If ω is in L(G) then left-most derivation of ω,
   error otherwise.

Initial State : $S on stack (with S being start symbol)
   ω$ in the input buffer

SET ip to point the first symbol of ω$.

repeat
   let X be the top stack symbol and a the symbol pointed by ip.

   if X∊ Vt or $
      if X = a
         POP X and advance ip.
      else
         error()
      endif

   else /* X is non-terminal */
      if M[X,a] = X → Y1, Y2,... Yk
         POP X
         PUSH Yk, Yk-1,... Y1 /* Y1 on top */
         Output the production X → Y1, Y2,... Yk
      else
         error()
      endif
   endif
until X = $     /* empty stack */
```

A grammar G is LL(1) if A ? a — are two distinct productions of G:
for no terminal, both a and derive strings beginning with a.
at most one of a and can derive empty string.
if ? t, then a does not derive any string beginning with a terminal in FOLLOW(A).

8.3.2 Algorithm : Error Recovery

The algorithm presented here attempts only a very simple error recovery mechanism for token errors. When an unexpected token is encountered it is simply grafted as an error token in the parse tree. This makes it fairly robust under errors due to extra tokens but susceptible to errors due to omitted tokens. It is not hard to construct scenarios where a single omitted token will cause the entire remaining input to be parsed incorrectly. This does not have the impact it would in a batch parser because parsing is limited to the cursor position and feedback is immediately given to the user. Limiting parser to the cursor position prevents the tokens to the right of the cursor from being parsed incorrectly based on an omitted token.

```
< Set Consistent Parse >
while (we have an error token) do
    if (token is parsable) then
        < Parse Token >
        if (Completed Structure)
            < Update Consistent Parse >
        endif
    else
        break;
    endif
endwhile
```

8.4 Verification and Validation for Acceptance

Verification and validation are independent procedures that are used together for checking that a product, service, or system meets requirements and specifications and that it fulfills its intended purpose.

8.4.1 Verification

Verification is intended to check that a product, service, or system (or portion thereof, or set thereof) meets a set of design specifications.In the development phase, verification procedures involve performing special tests to model or simulate a portion, or the entirety, of a product, service or system, then performing a review or analysis of the modeling results. In the post-development phase, verification procedures involve regularly repeating tests devised specifically to ensure that the product, service, or system continues to meet the initial design requirements, specifications, and regulations as time progresses.

At the time of development of each modules, they were tested independently. They were checked against the real output vs the expected output.If both of them were resulting to the same, the verification that the module is created is right or the output that it is producing is correct.

And for each module different cases of testing were considered as that of functional testing- considering all the edge cases.

8.4.2 Validation

The main purpose of our project was to build and IDE or application following the conditions-

- To learn to process text input using both regular expressions and parser generators.
- To create a learning tool which implements the processing of a regular expression using LL(k) parser generator.
- The parser generator consists of user defined grammar tree i.e a collection of tokens and expressions with user defined rules,action code, .this will help out the users to learn about the parsing techniques easily.

After building such application, it was tested against the above points, whether it is covering all the points or not. Based on this validation of our project was carried. It satisfies the crieteria what it was being aimed to be developed.

Chapter 9

Software Testing

9.1 Types of Testing Used

We have done Unit testing, Integration testing and functional testing on the application created.

9.1.1 Unit Testing

In unit testing, each module was tested individually.In this project we have checked all the modules with there corresponding output.

Grammar Tree Module:
In this Module the rules are created by modifying the root node of the grammar tree.The Grammar tree that has been created is can be individually tested as well as can be tested in combined form. The creation of the heirarichal tree is done in such a manner that any further adding of node will be automatically be enabled or disabled based on the rule creation.
Abstract Syntax tree Module:
The tree was basically the heirarichal structure of the grammar tree, how all the elements(Terminals and Non-terminals) are connected and organized with each other.So testing of this AST was mostly done by matching the structure of the tree that has been genearated with the expected sturcture for the grammar tree. If both of them were same, then AST generated was correct.

Action Code Module:
The action code is the optional part for each node. Without this, the strutrual log was generated for the expression. Applying action code to node

was specifying what action needs to be taken at that particulaar node. The action code was purely on the user side. SO testing of the action code will be from the users side. The action code basically consists of the logic. So, the action of a partiluar node is decided by the user.

Input Regular Expression Module:
Talking out Input module, it is taking correct input from the user. It checks whether the input string is according to the defined rules or not and if not then it gives error to the user to enter correct input string. For this we have used exceptional handling to deal with wrong input string.

Output Parser Log Module:
This part was generating the log or the output.If the solution of the regular expression returned by thid log module is correct, then this module is correct. If the solution id not upto the mark, then there may be problems in the grammar tree cration or action code.

9.1.2 Intergration Testing

In this phase in software testing in which individual software modules are combined and tested as a group. It occurs after unit testing and before validation testing.In this we are verifying functional, performance, and reliability requirements placed on major design modules.
In our product,major five modules get integrated together i.e Grammar tree Module,Abstract Syntax Tree Module,Action Code Module,Input Module and Parser Log Module.And we tested them combinedly by providing the test Grammar tree to first Module.Then,we provided the test input regular expression to the input Module.Which then generates the test Abstract Syntax tree in AST Module and Parser Log output in Output Module.
At the end by verifying the output results of different Modules with the test inputs we tested the given integrated module system.

9.1.3 Functional Testing

In order to perform Functional testing we have taken care of all the scenarios related to various cases which a user can provide to the system.
To make it easy and simple we are initially considering regular expressions which may range from siple plus minus to completex multiplication with

brackets. Normally the system can deal with almost all kind of regular expressions and generates error if it encounters some irrelevent data or out of bound cases.

9.2 Test Cases and Test Results

For checking the proper exectution of the software, we tested the software with different regular expression, functions,equations etc. Each test case involves with different functions to be tested.
Following are the test cases considered:

Test Case 1:
1) Recognition of built-in token types
2) Associating code with recognized patterns
3) Both styles of comment
’a’ ” ’
’ ”’’ ’ ” ’06’ ’77’ ’77’
123.4 45D 456.8e-33d // double
123.4f 67F 23.8e+5f // float

Test Case 2:
1) Recognition of signed numeric values
// double values ...
123.456d +123D -123.456d +456.8e-33 -456.8e-33d 456.8E+33 -456e33D
// float values ...
-9.8765f +98765F 9.8765f -23.8e+5f 23.8e5f 23.8e-15F -23.8e-5f

Test Case 3:
1) Semantic predicates in a sequence
/* length: */ 5 /* x-values: */ 20 23 75 67 78 /* y-values: */ 1 1 1 1 1
/* length: */ 5 /* x-values: */ 2345 0 765 90 345 /* y-values: */ 2345 0 765 90 345

Test Case 4:
1) Error handling and recovery
sumOne = 1 + 2 * 3 + 4 / 2 + 5;

sumTwo = 1 - x /* Error here */ 2 * 3 - 4 / 2 - 5;

Test Case 5:
1) Simple expression evaluation
sumOne = 1 + 2 - 3 + 4 + 5;
sumTwo = 1 - 2 + 3 - 4 - 5;
sumThree = 1+2-3+4+5;
sumFour = 1-2+3-4-5;
verify sumOne is 9, sumTwo is -7, sumThree is 9, sumFour is -7;

Chapter 10

Result

A visual parser-generator is an Integrated Development Environment was created used for the development of parsers without using text based grammar, script or code.

- It presents user given rules in the form of trees which gives a graphical view to parsel rules. It uses various icons in order to represents nodes in the tree where each node represent one rule.
- These generated trees represents Parsing Expression Grammer and can work with LL(k) grammars. The executable of these code is done by a single click of the button.
- This apllication provides facilities forautomatic abstract syntax tree construction helping users need tools for understanding and solving their problems in effiecient and flexible manner. They want to automate their tasks and generate output that can be easily applied into their application.
- It allowsusers to develop, edit, understand and test agrammar in an interactive userfriendly environment. This GUI visualizes the operations in generating parse trees and action code generation.

10.1 Screenshots

Below are some of the screenshots of the application, how an user can create rules, tokens literals etc.

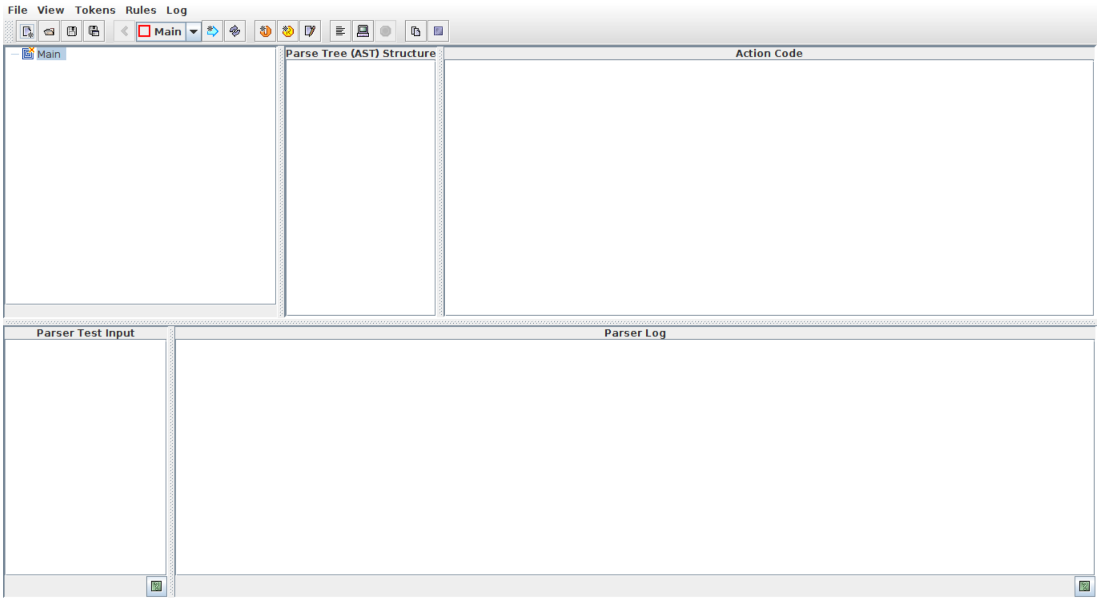


Figure 10.1: Rule Creation -1

10.2 Outputs

When the user clicks the generate output button,

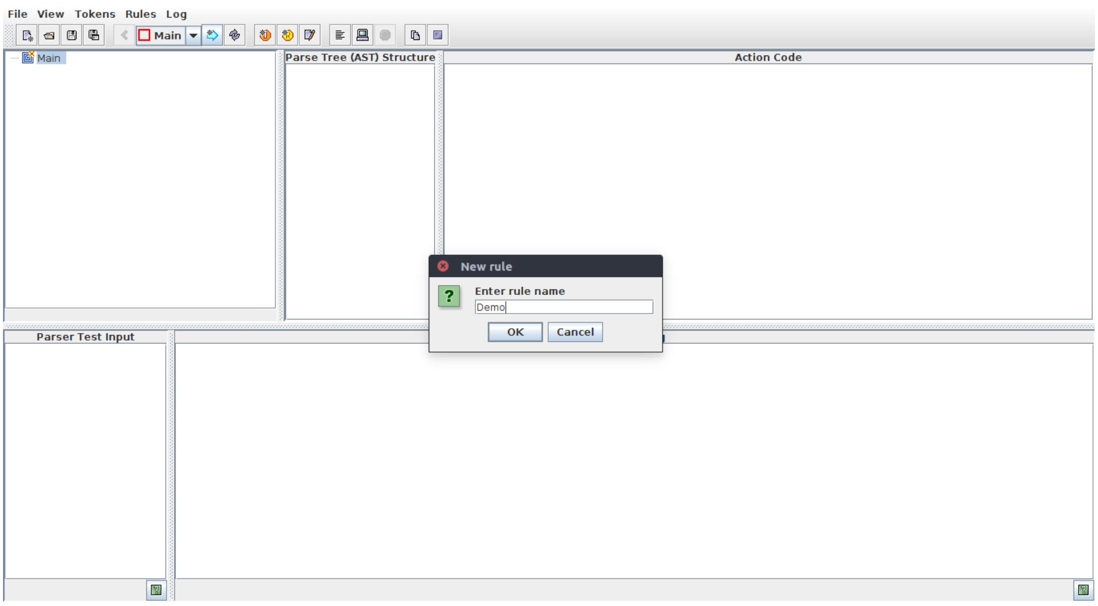


Figure 10.2: Rule Creation -2

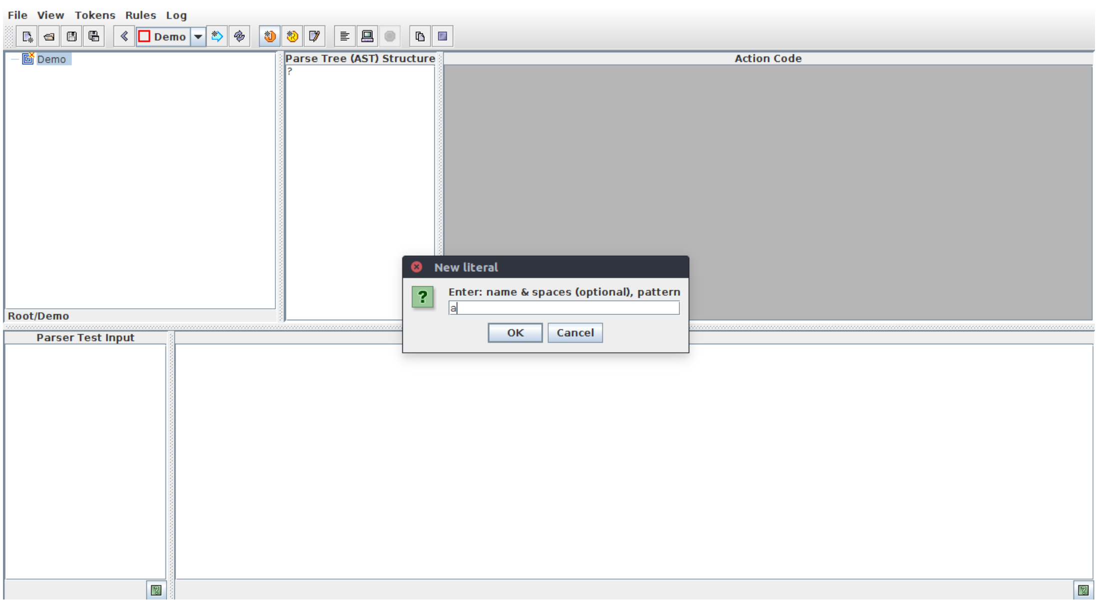


Figure 10.3: Rule Creation -3

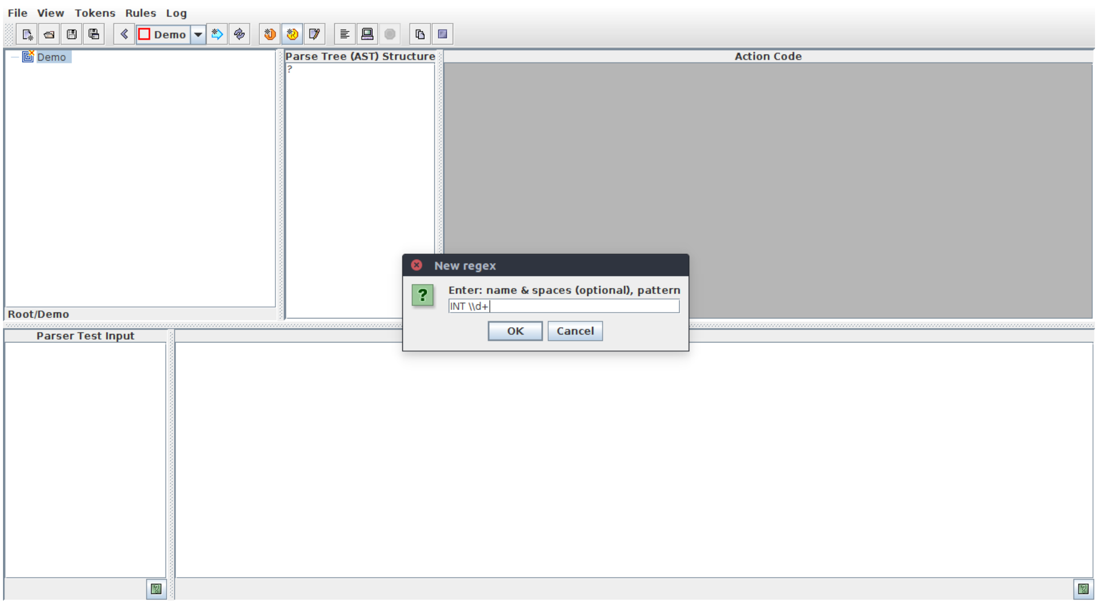


Figure 10.4: Rule Creation -4

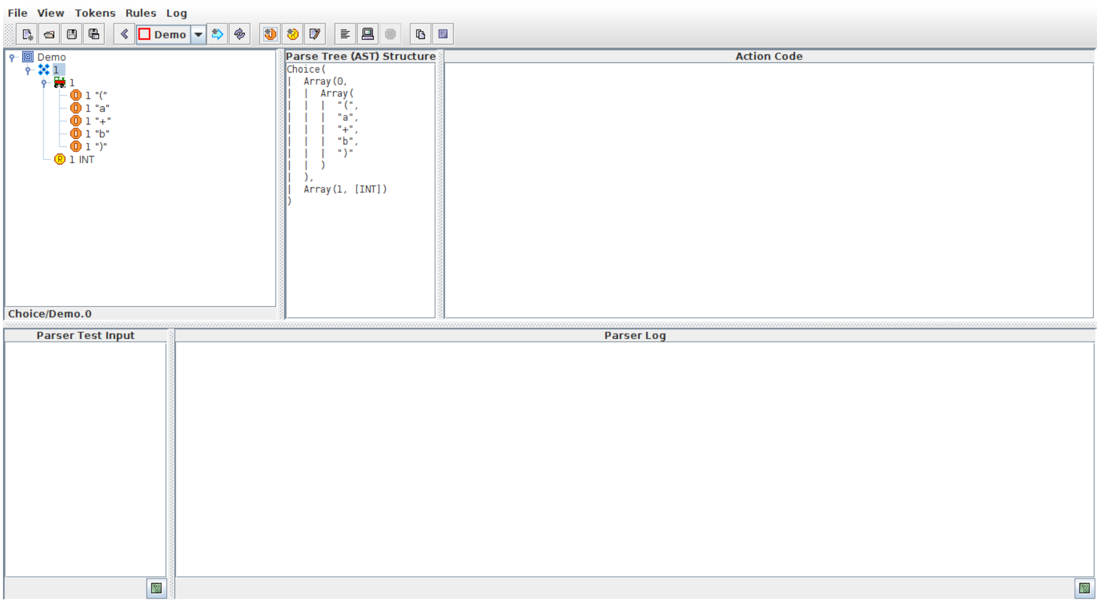


Figure 10.5: Rule Creation -5

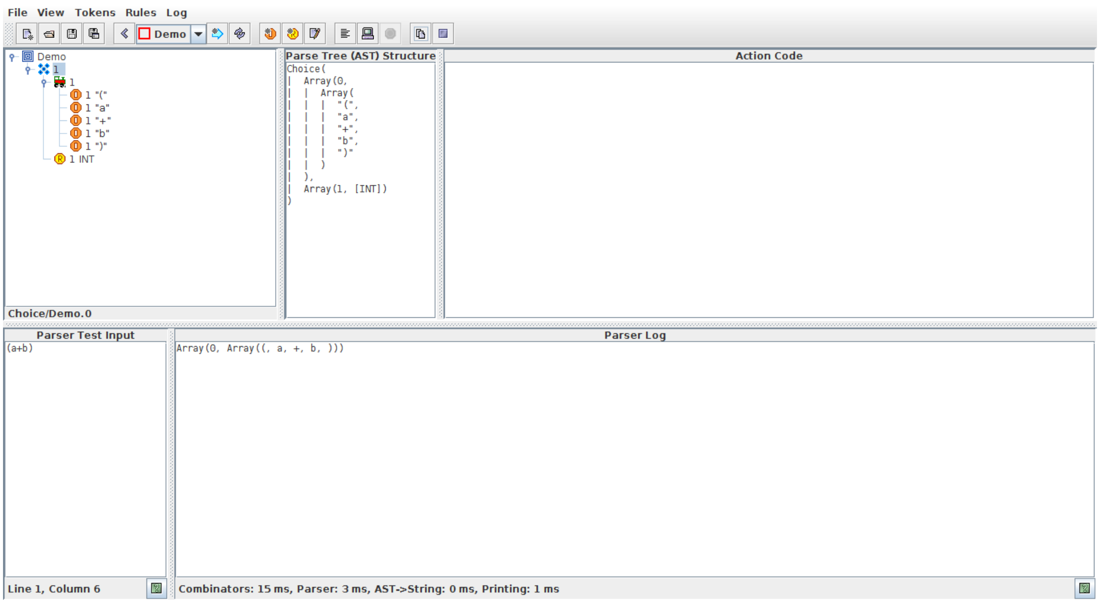


Figure 10.6: Output-1

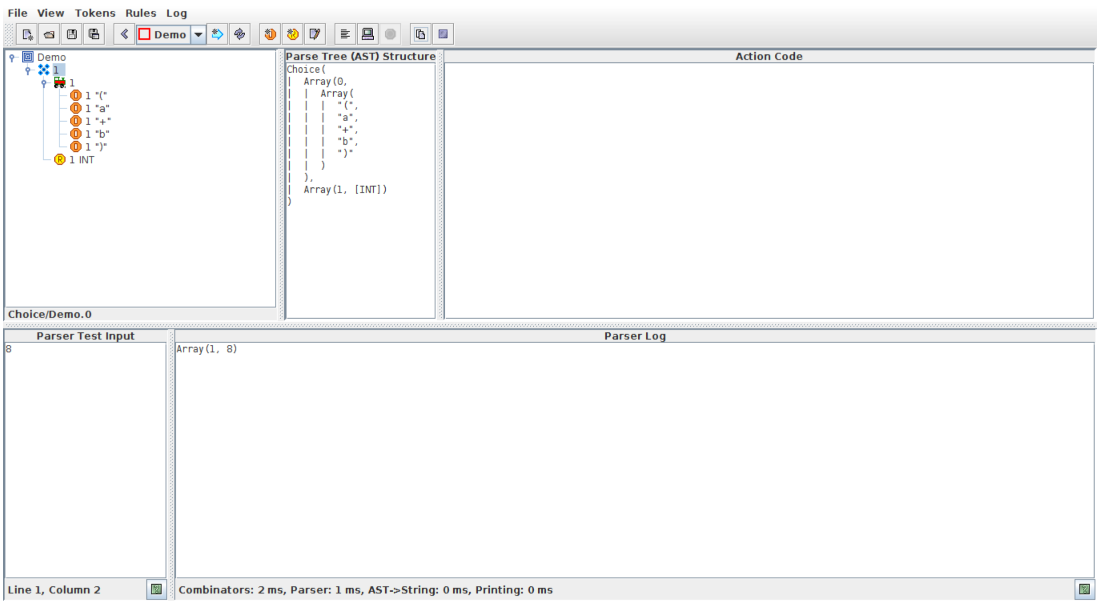


Figure 10.7: Output-2

Chapter 11

Deployment and Maintenance

11.1 Installation and un-installation

No specific installation action is required. Double-clicking executable.jar starts the VisualLangLab GUI within which you can create, modify, run, and test grammars. Users on Unix-like systems may first need to perform a "chmod +x executable.jar".

Alternatively, you may enter "java -jar executable.jar" at a command prompt.

11.2 User help

Before the user try to start using the application

- action-code: a function literal, coded in Javascript or Scala, associated with any grammar-tree node (also known as semantic action or just action in CS theory). The function is executed when VisualLangLab matches parser input to the node. more details can be found in AST and Action Code.html

- **saved-grammar file:** a XML file containing parsers developed with the VisualLangLab GUI. These files do not contain any code – just a description of all the parser-specific information available to the GUI. The format of a grammar-file is very intuitive, and it should be easy to use XSLT or something similar to translate it into code or other form

- **grammar-tree:** the visual tree that is used to display a parser-rule at the top left of The GUI. To the GUI user the grammar-tree is the parser-rule and there is no other user-accessible form of the parser-rule. Because of this, grammar-tree and parser-rule are used interchangably in the documentation.

- **parser:** generally used to refer to a program or body of code that parses text in a particular format (or grammar), but see note below. The GUI itself is a parser because it can parse text when Testing Parsers. An application program that loads a grammar-file by Using the API is also a parser.

- **parser-rule:** non-trivial parsers are simplified by breaking them down into a number of simpler parser-rules (often called just rule), but see note below. Each parser-rule is known by a unique name, and one of the parser-rules must be distinguished as the top-level (entry-point) entity

Chapter 12

Conclusion and Future scope

The development of a parser-generator Integrated Development Enviornment for designing parsers without using any grammar specification or without using any script or code is done.It represents parser rules by grammar-tree using different icons for the grammar-tree each nodes. They can be changed as well as executed, and can be run by User at any time by clicking a button.They want to automate their tasks and generate output that can be easily applied into their application. This aplication provides facilities for automatic abstract syntax tree construction.

This application accepts context-free grammars for parser generation. It allows users to develop, edit, understand and test agrammar in an interactive userfriendly environment. This GUI visualizes the operations in generating parse trees and action code generation. This tool is implemented for the LL(K) parsers only. In the future, it can be extended to other parser also like LR, SLR etc.

Chapter 13

References

[1] Incremental LL(1) Parsing in Language-Based Editors, *John J. Shilling*

[2] ANTLR: A Predicate LL(k) Parser Generator, *T. J. P ARR University of Minnesota, AHPCRC, 1100 Washington Ave S Ste 101, Minneapolis.*

[3] Visualization of Syntax Trees for Language Processing Courses, *Francisco J.Almeida-Mart nez, Jaime Urquiza-Fuentes J. Angel Vel azquez-Iturbide*

[4] Programming Language Processors in Java, *David A. Watt and Deryck F. Brown, Pearson Education.*

[5] Modern Compiler Implementation in Java, *A.W. Appel, Cambridge University Press.*

[6] Compilers: Principles, Techniques and Tools *A.V. Aho, R. Sethi, and J.D. Ullman, Addison Wesley.*

[7] https://en.wikipedia.org/wiki/Parser combinator.

[8] https://en.wikipedia.org/wiki/Parse tree.

Annexure A

Laboratory assignments on Project Analysis of Algorithmic Design

IDEA Matrix is represented in the following form. Knowledge canvas represents about identification of opportunity for product

I	D	E	A
Increase : Parsing capacity	Discover : Grammer Tree	Educate : Students, Faculties	Accurate : Regular Expression Output, Log
Improve : Handling of complex grammer rules	Deliever : Source Code, Abstract Syntex Tree, Log	Evaluate : Expression Through AST	Advance : Bottom up Parsers
Ignore : Ambiguous grammers	Dcrease : Complexity in Evaluation of Expression	Eliminate : Redundant Code	Avoid : Irrelevent Input Expression

Table A.1: IDEA Matrix

Annexure B

Project Problem Statement Feasibility Assessment

The development of an application or an IDE for the problem statement id difficult to implement in its normal form.It is np-complete, as large instances of the problems would be impractical to solve.NP consists of all problems whose answers can be easily verified.Find an x such that f (x) is minimized, which isnt easily verified. But, Is there an x such that f (x) less than c? usually is easily verified (assuming f is easy to compute), so its in NP. So if P = NP, we can solve problems of that type quickly. We can also quickly solve problems of the sort.

The proof focuses on the verification step of determining membership in NP. Consider the instance of the problem is showing the generation of an intractable number of items that must be verified, making the nondeterministic solution unverifiable in polynomial time.The LL(k) derives every k-length string in its alphabet, and contains 2k conflicts. Since k is an input to SLL(k) testing it should be clear that any certificate for this grammar must necessarily contain an exponential number of parts that must be individually verified. Each conflict is by nature and by definition separate from every other conflict.

There is no way to somehow combine conflicts to verify more than one at a time. This exponential number of conflicts is not deterministically verifiable in polynomial time.For the case of k greater than 1, SLL(k) verification would seem to be a function of the number and length of the lookahead strings that must be compared for equality.

Annexure C

Reviewers Comments of Paper Submitted

1. Paper Title: *Implementation of User Interface for LL(k) Parser Generator*
2. Name of the Conference/Journal where paper submitted :*International Journal of Advance Engineering and Research Development (IJAERD)*
3. Paper accepted/rejected : *Accepted*
4. Review comments by reviewer : *Nil*
5. Corrective actions if any : *No*

Annexure D

Plagiarism Report

ImplementationofUlforLL(k)parsergenerator

ORIGINALITY REPORT

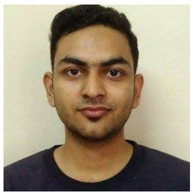
%19	%18	%3	%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	vll.java.net Internet Source	%7
2	en.wikipedia.org Internet Source	%3
3	www.scala-lang.org Internet Source	%2
4	docslide.us Internet Source	%1
5	www.ijarcsse.com Internet Source	%1
6	www.ukessays.com Internet Source	%1
7	T. J. Parr. "ANTLR: A predicated-LL(k) parser generator", Software Practice and Experience, 07/1995 Publication	%1
8	en.m.wikipedia.org Internet Source	%1
9	www.tutorialspoint.com Internet Source	%1

Annexure E

Information of Project Group Members



1. Name : Alok Singh
2. Date of Birth :06-04-1994
3. Gender : Male
4. Residential Address : Alok Singh, AIT , Dighi Hills, Alandi Road, Pune-411015.
5. E-Mail : aloksingh13138@aitpune.edu.in
6. Contact No. : 7263867299
7. Placement Details : Oracle, TCS
8. Paper Published : Yes

International Journal of Advance Engineering and Research Development (IJAERD)

DOI: 10.21090/IJAERD

Impact Factor: 4.72 (SJIF-2016)

ISSN (online): 2348-4470 ISSN(print): 2348-6406

Contact No: +91-9033132543 Email id: editor.ijaerd@gmail.com

Website: www.ijaerd.com

CERTIFICATE

The board members of the International Journal of Advance Engineering and Research Development are hereby awarding this certificate to

Alok Singh

In recognition of the publication of the paper entitled

Implementation of User Interface for LL(k) Parser Generator

Paper ID#: **"IJAERD2017V04I0466558"**

Published in IJAERD, **Volume 04 Issue 04, April-2017**

Editorial Board

of

International Journal of Advance Engineering and Research Development (IJAERD)

e-ISSN : 2348 - 4470, p-ISSN : 2348-6406

http://www.ijaerd.com/

Managing Director

E-mail (s): editor.ijaerd@gmail.com, editor@ijaerd.com

Note: "This document is electronically signed and reproduction of this document in any form is strictly prohibited"

1. Name : Irale Yogesh Lahu
2. Date of Birth : 08-09-1995
3. Gender : Male
4. Residential Address : Yogesh Irale, AIT , Dighi Hills, Alandi Road, Pune-411015.
5. E-Mail : iraleyogeshlahu13238@aitpune.edu.in
6. Contact No. : 7767983052
7. Placement Details : Amdocs, TCS
8. Paper Published : Yes

International Journal of Advance Engineering and Research Development (IJAERD)
DOI: 10.21090/IJAERD
Impact Factor: 4.72 (SJIF-2016)
ISSN (online): 2348-4470 ISSN(print): 2348-6406
Contact No: +91-9033132543 Website: www.ijaerd.com Email id: editor.ijaerd@gmail.com

CERTIFICATE

The board members of the International Journal of Advance Engineering and Research Development are hereby awarding this certificate to

Irale Yogesh Lahu

In recognition of the publication of the paper entitled

Implementation of User Interface for LL(k) Parser Generator

Paper ID#: **"IJAERD2017V04I0466558"**

Published in IJAERD, **Volume 04 Issue 04, April-2017**

Editorial Board

of

International Journal of Advance Engineering and Research Development (IJAERD)

e-ISSN : 2348 - 4470, p-ISSN : 2348-6406

http://www.ijaerd.com/

Managing Director

E-mail (s): editor.ijaerd@gmail.com, editor@ijaerd.com

Note: "This document is electronically signed and reproduction of this document in any form is strictly prohibited"

1. Name : Pankaj Kumar
2. Date of Birth : 05-12-1994
3. Gender : Male
4. Residential Address : Pankaj Kumar, AIT , Dighi Hills, Alandi Road, Pune-411015.
5. E-Mail : pankajkumar13154@aitpune.edu.in
6. Contact No. : 8390886493
7. Placement Details : Screen Magic, TCS
8. Paper Published : Yes

International Journal of Advance Engineering and Research Development (IJAERD)

DOI: 10.21090/IJAERD

Impact Factor: 4.72 (SJIF-2016)

ISSN (online): 2348-4470 ISSN(print): 2348-6406

Contact No: +91-9033132543 Website: www.ijaerd.com Email id: editor.ijaerd@gmail.com

CERTIFICATE

The board members of the International Journal of Advance Engineering and Research Development are hereby awarding this certificate to

Pankaj Kumar

In recognition of the publication of the paper entitled

Implementation of User Interface for LL(k) Parser Generator

Paper ID#: **"IJAERD2017V04I0466558"**

Published in IJAERD, **Volume 04 Issue 04, April-2017**

Editorial Board

of

International Journal of Advance Engineering and Research Development (IJAERD)

e-ISSN : 2348 - 4470, p-ISSN : 2348-6406

http://www.ijaerd.com/

Managing Director

E-mail (s): editor.ijaerd@gmail.com, editor@ijaerd.com

Note: "This document is electronically signed and reproduction of this document in any form is strictly prohibited"

1. Name : Rohit Rawat
2. Date of Birth :13-03-1996
3. Gender : Male
4. Residential Address :Rohit Rawat, AIT , Dighi Hills, Alandi Road, Pune-411015.
5. E-Mail : rohitrawat13226@aitpune.edu.in
6. Contact No. : 9403980678
7. Placement Details : Barclays
8. Paper Published : Yes

International Journal of Advance Engineering and Research Development (IJAERD)
DOI: 10.21090/IJAERD
Impact Factor: 4.72 (SJIF-2016)
ISSN (online): 2348-4470
ISSN(print): 2348-6406
Contact No: +91-9033132543
Email id: editor.ijaerd@gmail.com
Website: www.ijaerd.com

CERTIFICATE

The board members of the International Journal of Advance Engineering and Research Development are hereby awarding this certificate to

Rohit Rawat

In recognition of the publication of the paper entitled

Implementation of User Interface for LL(k) Parser Generator

Paper ID#: **"IJAERD2017V04I0466558"**

Published in IJAERD, **Volume 04 Issue 04, April-2017**

Editorial Board

of

International Journal of Advance Engineering and Research Development (IJAERD)

e-ISSN : 2348 - 4470, p-ISSN : 2348-6406

http://www.ijaerd.com/

Managing Director

E-mail (s): editor.ijaerd@gmail.com, editor@ijaerd.com

Note: "This document is electronically signed and reproduction of this document in any form is strictly prohibited"