A Tool for Teaching Parsing Techniques

A thesis submitted
in Partial Fulfillment of the Requirements
for the Degree of

Master of Technology

by

Nimisha Agarwal

to the
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
July, 2015

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

CERTIFICATE

It is certified that the work contained in the thesis titled **A Tool for Teaching Parsing Techniques**, by **Nimisha Agarwal**, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Amey Karkare
17/7/2015

Prof Amey Karkare

Department of Computer Science & Engineering

IIT Kanpur

July, 2015

ABSTRACT

Interactive Educational Systems are an emerging field in today's world. They extend the effectiveness of traditional classroom based education by making courses more reachable and manageable. Students are able to absorb knowledge at their own pace and time, rather than synchronize with the speed of instruction in classrooms.

Compiler Design is an important subject in UG CSE curriculum. In this thesis, we have developed a tool for teaching parsing techniques for a compiler design course. The tool helps impart knowledge of various parsing techniques to users through automatically generated problems and hints. Problems are generated based on a Context Free Grammar (CFG) given as input. The tool evaluates the solutions attempted by the user for these problems. Upon evaluation, if the solutions provided by the users are incorrect, it generates hint questions.

The problems generated by the tool follow a general Multiple Choice Question (MCQ) pattern, where a user is given a problem with a set of possible choices, 1 or more of which are correct. The incorrect solutions are the ones where a correct option is not chosen, or an incorrect option is chosen, or both. The hints are generated in the forms of (simplified) questions to direct student toward the correct solution. Hint generation procedures involve different types of algorithms, of which the input string generation algorithm is notable. For an incorrect answer for parse table provided by the user, this algorithm enables creation of an input string that distinguishes a successful parse from an unsuccessful one. We believe that the interactive nature of our tool will help users to learn from their own mistakes through experiments, and reduce the burden on teachers or teaching assistants.

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

To my parents

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

Acknowledgements

I would like to express my gratitude towards my thesis supervisor Prof. Amey Karkare, for providing me his valuable guidance and support throughout this thesis work. It would not have been possible to complete this thesis in due time and come up with an effective tool without his aid. He has provided me with the necessary resources to gather knowledge on building tools for programming languages and compilers. He has also enlightened me on solutions to problems which seemed too difficult to solve for me. I am highly thankful for all the motivation that he has provided to accomplish the thesis work.

I would also like to thank Mohit Bhadade for creating a web-based graphical user interface for the tool developed in this thesis. This has made it possible to demonstrate the real-world capabilities of the tool, by making it usable by students.

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

Contents

List of Algorithms

List of Figures

Chapter 1

Introduction

Interactive Educational Systems is an emerging technology nowadays. It started with school level education for courses like Maths, English, etc. Presently interactive courses are available for many technical and advanced studies. Notable examples include Coursera[1] and Khan Academy[2].

In earlier times, classroom teaching was the only way to impart education. It has been an effective mode of education for several ages, but it has constraints. There are always students in a class who are not able to interact much with their professors or classmates. Classroom education is restricted to a particular physical location, so it is not available to a significantly large audience. Above that, classrooms are generally conducted for a limited time and hence it is not possible to solve all the queries or doubts of students, then and there.

Now, an alternative method is evolving to solve the problems of classroom education. Students can be provided with questions for practice which can be evaluated by the system itself. This reduces the overhead of instructors and hence allows accommodation of a large number of students in the course. Problems can also be generated by such systems[3]. These systems also provide feedback to the students after evaluating their answers.

In this thesis, we propose an Interactive Educational System for the parsing phase of Compilers. Problems are generated for various concepts in the parsing phase of compilers, on which user solutions are evaluated. Hints in the form of questions and messages are generated whenever incorrect attempts are made by the users.

1.1 Objective

The purpose of this Thesis is to create an Interactive Tutoring System for the parsing phase of Compilers. The main objective is to generate problems automatically and evaluate the solution to those problems, which are submitted by the students. If a solution is wrong, then the system generates hint problems which guide the student to reach to the correct solution to the main problem. Otherwise it generates the next main problem. This work-flow is accomplished by calculating the solution to the problem beforehand. This pre-computed solution is then compared with the solution given by the student. The goal of this thesis is to generate problems automatically and provide immediate feedback to the students, while solving problems.

1.2 Motivation of Work

Students generally face a lot of problems in learning compiler technology. They do not understand the different techniques involved in various phases of program compilation. Parsing is one of such phases, which consists of a large number of techniques. Students face many problems in grasping these techniques.

Parsing techniques mostly follow different algorithms to solve a parsing problem. This requires a step by step method to reach to the solution to such a problem. If a student makes a mistake in even one step, then the result is completely wrong. It is also very difficult for the student or the instructor to figure out such a mistake as it involves recalculation of all the steps.

Therefore, we needed a system that can provide students with a large number of problems for practice. We also needed to make them realise their mistakes by generating feedback. Feedback is a necessary step to guide a student to the correct solution. Our feedback provide students with hint questions of different types to make them understand what went wrong and how to reach to the correct solution.

1.3 Outline of the Solution

A tool has been developed for the task of teaching parsing techniques to end users. The tool takes as input a context free grammar and uses it as a basis for generating questions. Several options are provided to the user initially, on the choice of domain for which questions would be generated. These domains are related to the various parsing techniques in compiler technology. Questions are generated on the domain specified by the user, using various algorithmic procedures. The algorithms are concerned with preprocessing, primary problem generation and hint generation. These algorithms are specific to the domain for which they are generated.

The normal workflow involves the following steps:

1. A context free grammar is taken as input from the user.
2. A choice of domain is taken as input from the user.
3. A primary question is generated based on the above two pieces of information.
4. If the user answers the problem incorrectly, then hints are generated for the same question in the form of questions or messages, depending on the context.
5. When a correct solution to the problem is received, another question from the same domain is generated and presented to the user.

A notable algorithm in this thesis is on input string generation for an incorrectly filled cell in a LL parsing table. This is a part of the hint generation procedures of the LL parsing domain.

1.4 Contribution of Thesis

In this thesis, we have designed algorithms to generate questions for the parsing phase of compilers. It generates problems automatically for the following different techniques involved in this phase:

- First set
- Follow set
- LL Parsing (which includes LL Parsing Table and LL Parsing Moves)
- SLR Parsing (which includes SLR Canonical set, SLR Parsing Table and SLR Parsing Moves)

We have tested the system on various grammars. The system also includes functionality like generation of input strings for a particular cell of the LL Parsing Table (in which a student has made a wrong entry while filling the table) and showing LL Parsing moves on that string. This enables the student to understand, why the value she entered in the cell is incorrect.

1.5 Thesis Organization

Rest of the thesis is organised as follows:

Chapter 2 covers the background knowledge required to understand this thesis. It covers concepts from compiler technology, with additional stress on parsing techniques.

Chapter 3 presents an overview of the tool developed during the thesis work. It describes the architecture of the tool along with the various phases in the workflow of the tool. A few examples are also included to provide a better understanding of the concepts and the algorithms used in the tool.

Chapter 4 contains all the major algorithms required to build the educational tool. The procedures for problem generation in each of the covered domains have been described here through the use of plain English and pseudo-code illustrations. Wherever necessary, examples and figures have also been included to provide a better understanding of the procedures.

Chapter 5 captures the interface exported by the tool. This interface is required to enable the tool to be used via a graphical user interface, for effectiveness of instruction. The chapter covers the description of the files used to perform this integration.

Chapter 6 illustrates the working of the tool through the use of narrations and screenshots. It covers an interactive walkthrough of the tool.

Chapter 7 gives an idea about the related work that has been done in this area.

Chapter 8 concludes this thesis with statements about its usability and relevance to educational systems.

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

Chapter 2

Background

This chapter provides an overview of the concepts required to understand this thesis. Brief explanations about concepts in Compiler Design are covered here. For details readers can refer to any standard compiler textbook[4].

2.1 Compiler

A translator is a program that converts a program written in one programming language (the source language) into a program in another language (the object or target language). A compiler is a translator, whose source language is a high-level programming language (such as C, C++), and object language is a low-level language such as assembly language or machine language.

The compilation process is divided into 5 phases as shown in Figure 2.1.

Following is a brief description of each of the phases of a Compiler:

1. **Lexical Analysis:** The first phase of Compiler is called Lexical Analysis. The tool (lexical analyzer or scanner) built for this task, separates the characters of the source language into groups that logically belong together. These groups are called tokens.

   The usual tokens are:

   - **Keywords:** Keywords are the reserved words in a programming language. They have the fixed meaning and cannot be changed by the user. Examples include 'IF', 'WHILE', etc.
   - **Identifiers:** These are the variables and function names defined by the user in the program, such as 'x', 'sum'.
   - **Operator Symbols:** They symbolize a specific action and operate on certain values. Examples include '<', '+', etc.
   - **Punctuation Symbols:** They separates words or phrases (that have meaning to a compiler). Examples include '(', ',', etc.

   The output of this phase is a stream of tokens, which is passed to the next phase of Compiler.

2. **Syntax Analysis:** The syntax analyzer or parser groups tokens together into syntactic structures. If A*B+C is a string (containing 5 tokens), then it

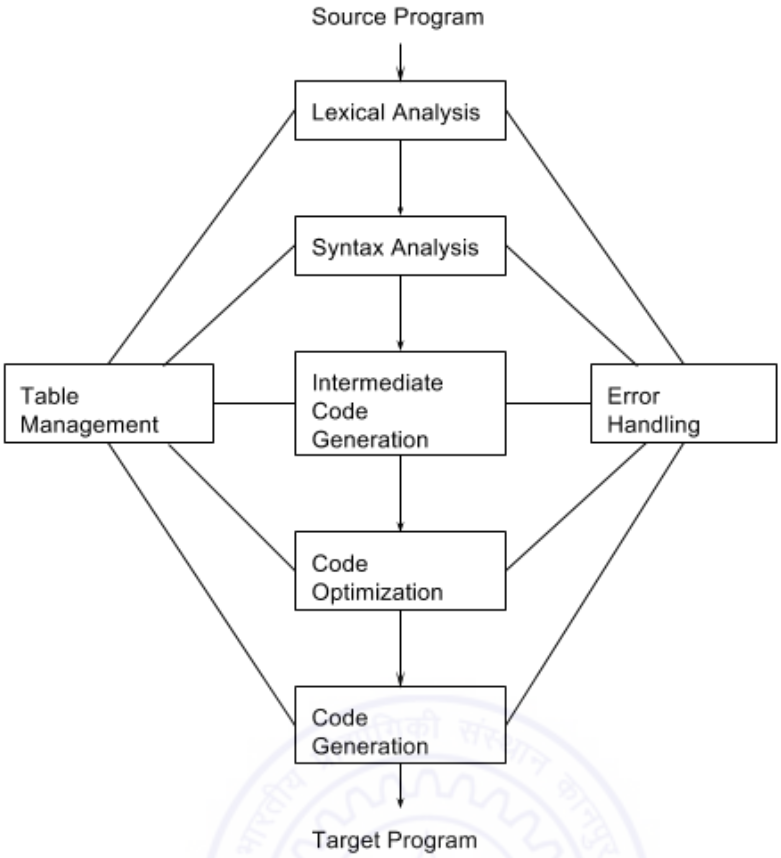

Figure 2.1: Phases of Compiler[4]

can be considered as (A*B)+C or as A*(B+C), depending on the language definition. The syntactic structure determines how these tokens are to be grouped together into larger structures called syntactic categories such as: **expressions** - sequences of operator and operands (constants and identifiers) or **statements** - multiple expressions can be combined to form statements.

The syntactic structure can be represented as a tree, known as a parse tree. Tokens appear on the leaves of this tree and the interior nodes of the tree represent strings of tokens that logically belong together.

Example 2.1 *The three tokens representing 'A+B' can be grouped into an expression. A possible parse tree for 'A+B' can be represented as in Figure 2.2.*

3. **Intermediate Code Generation:** The intermediate code generator uses the structure produced by the syntax analyzer to create a stream of simple instructions. These instructions can be in the form of "Macros".

   Macros are small functions which can be converted into assembly language code.

   Example 2.2 *For addition, a macro can be defined as:*

```
MACRO   ADD2    X,Y
        LOAD    Y
```

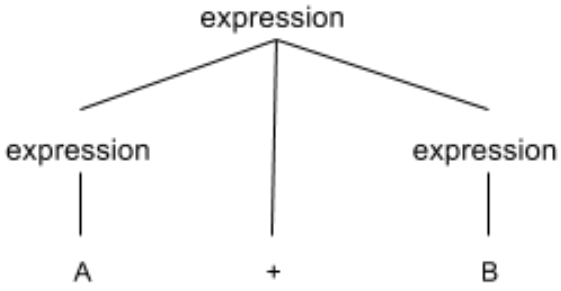

Figure 2.2: A possible parse Tree for A+B

```
        ADD     X
        STORE   Y
ENDMACRO
```

For A+B, this macro can be used as:

```
LOAD    B
ADD     A
STORE   B
```

4. **Code Optimization:** This phase is used to improve the intermediate code so that the final object program runs faster and/or takes less space. Its output is another intermediate code program which is an improved version of the previous one.

   Example 2.3 *Suppose we have the following code snippet:*

```
while(j<=10)
{
    i = 2;
    j = j + i;
}
```

   Here, the assignment i = 2 is executed in every iteration of the loop. But, the value of i remains constant in all iterations and is not changed in the loop. So, this assignment statement can be placed outside the loop to optimize the code such as

```
i = 2;
while(j<=10)
{
    j = j + i;
}
```

5. **Code Generation:** This is the final phase of Compiler. It produces the object code by deciding on the memory locations for data, selecting code to access each datum, and selecting the registers in which each computation is to be done.

   Example 2.4 *For the instruction "A+B", code can be generated as:*

```
MOV A, R0
ADD B, R0
```

6. **Table Management:** Table Management or book-keeping keeps track of the names used by the program and records essential information about each. An example of information is the type (integer, real) of a variable. For storing such information a data structure known as a symbol table is used.

7. **Error Handling:** The error handler warns the programmer about the flaws in the source program, by issuing a diagnostic and adjusting the information being passed from phase to phase, so that compilation can proceed till the last phase.

2.1.1 Some Terminology

Some terms related to the parsing phase of Compiler have been used in this thesis. A brief description of these terms is given below:

- **Alphabet:** The set of symbols used in a programming language is called the alphabet of that programming language. Eg: {a,b}.

- **Language:** Any set of strings formed from some specific alphabet. Eg: L = {{ϵ, a, b, aa, bb, ab, ba} | {a,b} $\in$ alphabet}.

- **Kleene Closure:** It is a unary operation defined as follows: if V is a set of symbols or characters, then V* (kleene closure) is the set of all strings over symbols in V, including the empty string ϵ.

- **Grammar:** A grammar is a set of production rules, to form strings from a language's alphabet. A grammar checks the validity of strings, by checking their syntactic correctness.

  A grammar G = (N, Σ, P, S) consists of the following components:

  - A finite set **N** of non-terminals (synonym for syntactic categories), that is disjoint with the strings formed from G.
  - A finite set Σ of terminals (synonym for tokens) that is disjoint from N.
  - A finite set **P** of production rules, each rule of the form
    $(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$
    where * is the Kleene closure and $\cup$ denotes set union.
  - The symbol **S**, where S $\in$ N is the start symbol.

  Example 2.5 *The grammar G, where* $N = \{S, B\}$, $\Sigma = \{a, b, c\}$, *S is the start symbol, and P consists of the following production rules:*

  $S \rightarrow aBSc$

  $S \rightarrow abc$

  $Ba \rightarrow aB$

$Bb \rightarrow bb$

- **Context-free Grammar:** A context-free grammar(CFG) is a grammar in which the left-hand side of each production rule consists of only a single non-terminal symbol.

  Example 2.6 *The grammar with $N = \{S\}$, $\Sigma = \{a, b\}$, S the start symbol, and the following production rules, P:*

  $S \rightarrow aSb$

  $S \rightarrow ab$

- **Derivation:** A derivation replaces a non-terminal on the LHS of a production with its respective RHS.

  Example 2.7 *One of the strings, which can be derived from the grammar in example 2.6, by the following steps:*

  $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb$

  If the leftmost non-terminal is always expanded in the derivation, to acquire the string, then it is a leftmost derivation. Similarly if the rightmost non-terminal is always expanded, then it is a rightmost derivation.

  Example 2.8 *Suppose the grammar is:*

$$\begin{aligned} E &\rightarrow T\ E' \\ E' &\rightarrow +\ E \\ &\mid\ \epsilon \\ T &\rightarrow 0 \end{aligned}$$

  then, the following is the leftmost derivation

  $E \Rightarrow TE' \Rightarrow 0E' \Rightarrow 0 + E \Rightarrow 0 + TE' \Rightarrow 0 + 0E' \Rightarrow 0 + 0$

  and the rightmost derivation can be

  $E \Rightarrow TE' \Rightarrow T + E \Rightarrow T + TE' \Rightarrow T + T \Rightarrow T + 0 \Rightarrow 0 + 0$

- **Sentential Form:** Let G = (N, Σ, P, S) be a CFG, and $\alpha \in (N \cup \Sigma)^*$. If $S \overset{*}{\Rightarrow} \alpha$ (where $\overset{*}{\Rightarrow}$ means derivation in zero or more steps), then α is the sentential form.

  If $S \underset{lm}{\Rightarrow} \alpha$ (leftmost derivation), then α is a left-sentential form and if $S \underset{rm}{\Rightarrow} \alpha$ (rightmost derivation), then α is a right-sentential form.

  Example 2.9 *Consider the example 2.6. Each of {S, aSb, aaSbb, aaabbb}, derived from the set of production rules, is a sentential form.*

- **Lookahead:** Some parsing algorithms use a technique of looking ahead certain tokens in order to decide which rule to use. The maximum number of tokens that a parser can use to decide which rule it should use, is known as lookahead.

- **Handle:** A handle of a string is a substring that matches the right side of a production and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a rightmost derivation.

  A handle of a right-sentential form γ is a production $A \to \beta$ and a position of γ where the string β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ. That is, if $S \overset{*}{\Rightarrow} \alpha Aw \Rightarrow \alpha\beta w$, then $A \to \beta$ in the position following α is a handle of $\alpha\beta w$. the string w to the right of the handle contains only terminal symbols.

- **DFA:** Deterministic Finite Automaton is a finite state machine that accepts or rejects finite strings of symbols and only produces a unique computation (or run) of the automaton for each input string.

  A DFA A = (Q, Σ, δ, q_0, F) consists of:

  1. A finite set of states, denoted by Q.
  2. A finite set of input symbols, denoted by Σ.
  3. A transition function, denoted by δ, that takes as arguments a state and an input symbol and returns a state. Eg: If q is a state and a i an input symbol, then $\delta(q, a)$ is that state p such that there is an arc labeled a from q to p.
  4. A start state, denoted by q_0. It is one of the states in Q.
  5. A set of final or accepting states F. The set F is a subset of Q.

2.1.2 Parsing Phase of Compiler

Parsing is the second phase of Compiler. It is performed after Lexical Analysis. A sequence of tokens (output of the lexical analyzer), is passed to parsing phase as input. The parser then checks whether the input string is syntactically correct or not. For this task, a CFG is used to define the syntax of a programming language. A parsing table is constructed, using the CFG, which is used to parse the input strings.

There are various parsing techniques which are discussed in the later parts of this chapter. The following section briefly describes FIRST and FOLLOW set, which is necessary for constructing parsing tables.

2.1.2.1 Preliminaries

- **FIRST:** If α is any string of grammar symbols, then FIRST(α) is the set of all terminals, that appear in the beginning of strings derived from α. If $\alpha \overset{*}{\Rightarrow} \epsilon$, then ϵ is also in FIRST(α).

  To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ϵ can be added to any FIRST set.

  1. If X is a terminal, then FIRST(X) is X.

2. If X is a non-terminal and X $\rightarrow$ aα is a production, then add a to FIRST(X). If X $\rightarrow \epsilon$ is a production, then add ϵ to FIRST(X).
3. If X $\rightarrow Y_1Y_2....Y_k$ is a production, then for all i such that all of Y_1,....,Y_{i-1} are non-terminals and FIRST(Y_j) contains ϵ for j = 1, 2, ..., i-1 (i.e $Y_1Y_2....Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$), add every non-$\epsilon$ symbol in FIRST(Y_i) to FIRST(X). If ϵ is in FIRST(Y_j) for all j = 1, 2, ..., k, then add ϵ to FIRST(X).

Example 2.10 *Suppose the grammar is*

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow \epsilon | +E \\ T &\rightarrow 0|1 \end{aligned}$$

Then

FIRST[E] = {1, 0}

FIRST[E'] = {+, ϵ}

FIRST[T] = {1, 0}

- **FOLLOW:** FOLLOW(A), for non-terminal A, is the set of terminals, that can appear immediately to the right of A, in some sentential form. That is, $S \stackrel{*}{\Rightarrow} \alpha Aa\beta$ for some α and β. If A is the rightmost symbol in some sentential form, then we add $ to FOLLOW(A).

  1. $ is in FOLLOW(S), where S is the start symbol.
  2. If there is a production $A \rightarrow \alpha B\beta$, $\beta \neq \epsilon$, then everything in FIRST(β) but ϵ is in FOLLOW(B).
  3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B\beta$ where FIRST(β) contains ϵ (i.e., $\beta \stackrel{*}{\Rightarrow} \epsilon$), then everything in FOLLOW(A) is in FOLLOW(B).

Example 2.11 *For the grammar used in example 2.10*

FOLLOW[T] = {$, +}

FOLLOW[E] = {$}

FOLLOW[E'] = {$}

2.1.2.2 LL Parsing

It is a top-down parsing technique. The first L in LL(1) stands for parsing the input from Left to right and the second L for performing Leftmost derivation of the input string (1 is the number of lookaheads). Because of this lookahead, the parser is categorised as a predictive parser. To parse an input string, it starts with the root and works down to the leaves. Here, start symbol is the root of the parse tree and the sequence of terminals, in the input string, forms the leaves of the tree. Every parse tree represents a string generated by the grammar.

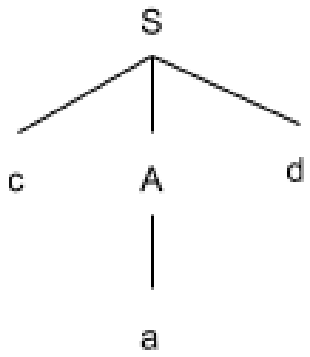

Figure 2.3: A possible parse Tree for string cad

Example 2.12 *If the grammar is:*
$S \rightarrow cAd$
$A \rightarrow a \mid ab$
A possible parse tree for the string 'cad' is represented in figure 2.3.

There are several difficulties with this parsing technique. One of them is Left-recursion. A grammar G is said to be left-recursive if it has a non-terminal A such that there is a derivation $A \stackrel{+}{\Rightarrow} A\alpha$ ($\stackrel{+}{\Rightarrow}$ represents derivation in 1 or more steps) for some α. A left-recursive grammar can cause the top-down parser to go into an infinite loop. Hence such grammars are not LL(1) and are not supported by LL Parsing.

Another issue with this parsing technique is left-factoring. Suppose if we have a production such as: A → abc | ab, then on seeing a, we could not tell, which production to choose, in order to expand the statement. For solving this problem, left-factoring is performed on the grammar. It is the process of factoring out the common prefixes of alternates. The following example explains the procedure of left-factoring:

Example 2.13 *If* $A \rightarrow \alpha\beta \mid \alpha\gamma$ *are two productions in the grammar, then after left-factoring, the production rules are of the form:*
$A \rightarrow \alpha A'$
$A' \rightarrow \beta \mid \gamma$

To parse an input string using LL parsing, LL Parsing table is required. Following is a brief description of a LL parsing table and the calculation of LL parsing moves.

LL Parsing Table
It is a two-dimensional array M[A, b], where A is a non-terminal and b is a terminal or $. The entries in the LL parsing table are filled by using FIRST and FOLLOW sets. An input string is parsed by using this table.

Algorithm 1 described below, can be used to construct the LL parsing table for a grammar G.

The undefined entries are usually left blank, instead of writing error in them.

Example 2.14 *For the grammar used in example 2.10, LL Parsing table can be filled using the above algorithm as:*

Algorithm 1 Construction of LL Parsing Table

Require: Grammar G.
Ensure: Parsing Table M.
1: For each production A $\rightarrow \alpha$ of the grammar, do steps 2 and 3.
2: For each terminal a in FIRST(α), add A $\rightarrow \alpha$ to M[A, a].
3: If ϵ is in FIRST(α), add A $\rightarrow \alpha$ to M[A, b] for each terminal b in FOLLOW(A). If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add A $\rightarrow \alpha$ to M[A, \$].
4: Make each undefined entry of M error.

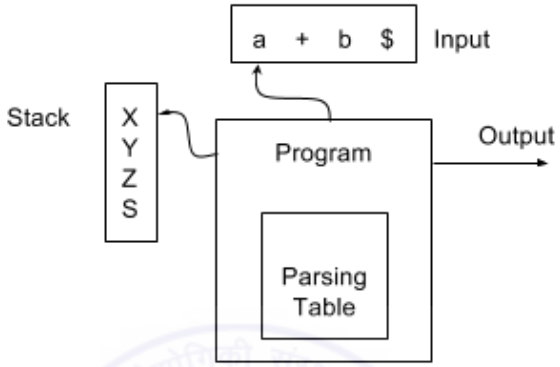

Figure 2.4: Model of a predictive parser(taken from textbook[4])

	0	***1***	***+***	***$***
T	$T \rightarrow 0$	$T \rightarrow 1$		
E	$E \rightarrow T\ E'$	$E \rightarrow T\ E'$		
E'			$E' \rightarrow +\ E$	$E' \rightarrow \epsilon$

A grammar is said to be LL(1) if there are no multiply-defined entries in its parsing table. Left-recursive grammars are not LL(1). Also, left-factoring must be performed on grammars if required, otherwise, they may result in multiple entries in a cell of LL parsing table.

LL Parsing Moves

The parser has an input, a stack, a parsing table, and an output as shown in figure 2.4. The input contains the string to be parsed, followed by \$, the right endmarker. The stack contains a sequence of grammar symbols, preceded by \$ (the bottom-of-stack marker). The contents of the stack, in each step, show the leftmost derivation on the input string. Output shows the action done in each step.

If X is the symbol on top of the stack and a is the current input symbol, then following are the rules that determine the action of the parser.

1. If X = a = \$, the parser halts and announces successful completion of parsing.

2. If X = a $\neq$ \$, the parser pops X off the stack and advances the input pointer to the next input symbol.

3. If X is a nonterminal, the program consults entry M[X, a] of the parsing table M. If M[X, a] = X $\rightarrow$ UVW, the parser replaces X on top of the stack by WVU (with U on top). If M[X, a] = error, the parser calls an error recovery routine.

The following example shows the parsing of an input string by applying the above rules. Initially, the stack contains the start symbol of the grammar preceded by $.

Example 2.15 *For the same grammar used in example 2.10, if the input string is "0 + 1", then the parsing moves are:*

STACK	***INPUT***	***OUTPUT***
$\$E$	$0+1\$$	
$\$E'T$	$0+1\$$	$E \rightarrow T\ E'$
$\$E'0$	$0+1\$$	$T \rightarrow 0$
$\$E'$	$+1\$$	
$\$E+$	$+1\$$	$E' \rightarrow +\ E$
$\$E$	$1\$$	
$\$E'T$	$1\$$	$E \rightarrow T\ E'$
$\$E'1$	$1\$$	$T \rightarrow 1$
$\$E'$	$\$$	
$\$$	$\$$	$E' \rightarrow \epsilon$

2.1.2.3 SLR Parsing

It is one of the LR parsing techniques. It is a bottom-up parsing method. As the name implies, LR parsers scan the input from left-to-right and construct a rightmost derivation in reverse. SLR stands for Simple LR Parsing. K is usually 1 for this parsing. It is the easiest to implement, but may fail to produce a table for certain grammars.

This parsing works on left-recursive grammars, but it is necessary to perform left-factoring if required.

To construct the SLR parsing table, the canonical collection of SLR items are calculated. In later parts of this section, a brief discussion about canonical sets, SLR parsing table and moves is given.

Canonical Collection of SLR Items

To construct the parsing table, a DFA from the grammar is constructed. The DFA recognizes viable prefixes of the grammar, that is, prefixes of right-sentential forms that do not contain any symbols to the right of the handle.

SLR item of a grammar G is defined as a production of G with a dot at some position on the right side, which indicates how much of a production we have seen at a given point in the parsing process. Thus, production $A \rightarrow XYZ$ generates the four items:

$A \rightarrow .XYZ$
$A \rightarrow X.YZ$
$A \rightarrow XY.Z$
$A \rightarrow XYZ.$

and the production "$A \rightarrow \epsilon$" generates only one item, "$A \rightarrow .$" . As an example, the second item in the above productions, would indicate that we have just seen on the input, a string derivable from X and that we next expect to see a string derivable from YZ. These items are grouped together as itemsets, which are further grouped to form a Canonical SLR collection. To construct this collection we need to define an augmented grammar and two functions - CLOSURE and GOTO.

- **Augmented Grammar:** If G is a grammar with start symbol S, then G', the augmented grammar for G, is G with a new start symbol S' and production $S' \rightarrow S$. This new starting production is used to indicate to the parser when it should stop parsing and announce acceptance of the input. This would occur when the parser was about to reduce by $S' \rightarrow S$.

- **CLOSURE:** If I is a set of items for a grammar G, then the set of items CLOSURE(I) is constructed from I by the rules:

  1. Every item in I is in CLOSURE(I).
  2. If $A \rightarrow \alpha.B\beta$ is in CLOSURE(I) and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow .\gamma$ to I, if it is not already there.

  Following example illustrate the CLOSURE function:

  Example 2.16 *Suppose the augmented grammar is:*

  $E'' \rightarrow E$

  $E \rightarrow TE'$

  $E' \rightarrow \epsilon | + E$

  $T \rightarrow 0|1$

  If I is the set of one item $[E'' \rightarrow .E]$*, then CLOSURE(I) contains the items*

  E" → .E

  E → .TE'

  T → .1

  T → .0

- **GOTO:** GOTO(I, X), where I is a set of items and X is a grammar symbol, is the closure of the set of all items $[A \rightarrow \alpha X.\beta]$ such that $[A \rightarrow \alpha.X\beta]$ is in I. Following example illustrates the calculation of GOTO function:

  Example 2.17 *For the augmented grammar used in example 2.16, if I is the set of items* $\{[E'' \rightarrow .E], [E \rightarrow .TE'], [T \rightarrow .0], [T \rightarrow .1]\}$*, then GOTO(I, T) consists of:*

  E → T.E'

  E' → .

  E' → .+E

The algorithm 2 is used to construct C, the canonical collection of sets of LR(0) items for an augmented grammar G'.

Following example illustrates the construction of collection of sets of LR(0) items:

Example 2.18 *For the augmented grammar used in example 2.16, SLR Canonical set is*

Algorithm 2 Canonical collection of sets of SLR items Construction

1: C = CLOSURE($S' \rightarrow .S$)
2: **repeat**
3: **for** each set of items I in C and each grammar symbol X such that GOTO(I, X) is not empty and is not in C **do**
4: add GOTO(I, X) to C
5: **end for**
6: **until** no more sets of items can be added to C

I0: $T \rightarrow .1$
$T \rightarrow .0$
$E" \rightarrow .E$
$E \rightarrow .TE'$

I1: $E \rightarrow T.E'$
$E' \rightarrow .$
$E' \rightarrow .+E$

I2: $T \rightarrow 1.$

I3: $T \rightarrow 0.$

I4: $E" \rightarrow E.$

I5: $T \rightarrow .1$
$E \rightarrow .TE'$
$E' \rightarrow +.E$
$T \rightarrow .0$

I6: $E \rightarrow TE'.$

I7: $E' \rightarrow +E.$

SLR Parsing Table

This table is divided into two parts - Action and Goto. Algorithm 3 shows the construction of SLR parsing action and goto functions from the DFA that recognizes viable prefixes. Each entry in the table determines whether to shift the input symbol on the stack or to reduce a string of symbols on top of stack by a single symbol using the productions of grammar.

Following example shows the SLR parsing table for an augmented grammar.

Example 2.19 *For the grammar used in example 2.16, SLR Parsing table is*

STATE	***ACTION***				***GOTO***		
	0	***1***	***+***	***$***	***T***	***E***	***E'***
0	*s3*	*s2*			*1*	*4*	
1			*s5*	*r4*			*6*
3			*r1*	*r1*			
4				*accept*			
5	*s3*	*s2*			*1*	*7*	
6				*r3*			
7				*r5*			

Algorithm 3 Construction of SLR Parsing Table

Require: C, the canonical collection of sets of items for an augmented grammar G'.
Ensure: If possible, an LR parsing table consisting of a parsing action function ACTION and a goto function GOTO.

Let C = I_0, I_1,...., I_n. The states of the parser are 0, 1,..., n, state i being constructed from I_i. The parsing actions for state i are determined as follows:

1: **if** $[A \rightarrow \alpha.a\beta]$ is in I_i and GOTO(I_i, a) = I_j **then**
2: set ACTION[i, a] to "shift j". Here a is a terminal.
3: **end if**
4: **if** $[A \rightarrow \alpha.]$ is in I_i **then**
5: set ACTION[i, a] to "reduce $A \rightarrow \alpha$" for all a in FOLLOW(A).
6: **end if**
7: **if** $[S' \rightarrow \alpha S.]$ is in I_i **then**
8: set ACTION[i, $] to "accept".
9: **end if**

The goto transitions for state i are constructed using the rule:

10: **if** GOTO(I_i, A) = I_j **then**
11: GOTO[i, A] = j.
12: **end if**
13: All entries not defined by rules 1 through 12 are made "error".
14: The initial state of the parser is the one constructed from the set of items containing [S'→ .S].

Blank entries are the "error" entries.

SLR Parsing Moves

An LR parser consists of two parts - a driver routine and a parsing table. The driver routine is the same for all LR parsers, but the parsing table changes from one parser to another. Figure 2.5 shows the model of LR Parsers.

The input is read from left to right, one symbol at a time. The stack contains a string of the form $s_0X_1s_1X_2s_2....X_ms_m$, where s_m is on top. Each X_i is a grammar symbol and each s_i is the state.

The entry ACTION[s_m, a_i], where s_m is the state on top of stack and a_i is the current input symbol, can have one of four values:

1. shift s.

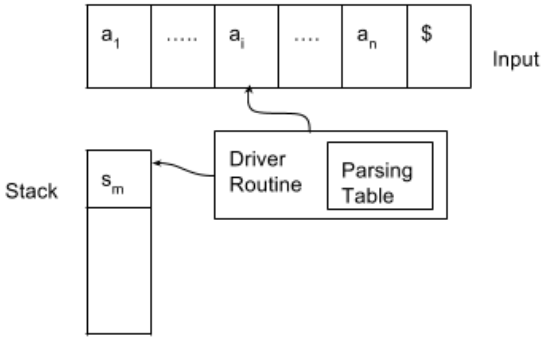

Figure 2.5: Model of a SLR parser[4]

2. reduce $A \rightarrow \beta$.
3. accept.
4. error.

The function GOTO takes a state and grammar symbol as arguments and produces a state.

A configuration of an LR parser is a pair whose first component is the stack contents and whose second component is the unexpected input:

(s_0 X_1 s_1 X_2 s_2 ... X_m s_m, a_i a_{i+1} ... a_n \$)

The configurations resulting after each of the four types of move, on consulting the parsing action table entry ACTION[s_m, a_i], are as follows:

1. If ACTION[s_m, a_i] = shift s, the parser executes a shift move, entering the configuration

   (s_0 X_1 s_1 X_2 s_2 ... X_m s_m a_i s, a_{i+1} ... a_n \$)

   Here the parser has shifted the current input symbol a_i and the next state s = GOTO[s_m, a_i] onto the stack. a_{i+1} becomes the new current input symbol.

2. If ACTION[s_m, a_i] = reduce $A \rightarrow \beta$, then the parser executes a reduce move, entering the configuration

   (s_0 X_1 s_1 X_2 s_2 ... X_{m-r} s_{m-r} A s, a_i a_{i+1} ... a_n \$)

   where s = GOTO[s_{m-r}, A] and r is the length of β, the right side of the production. Here the parser first popped 2r symbols off the stack (r state symbols and r grammar symbols), exposing state s_{m-r}. The parser then pushed both A, the left side of the production, and s, the entry for ACTION[s_{m-r}, A], onto the stack. The current input symbol is not changed in a reduce move. For the LR parsers we shall construct, X_{m-r+1} ... X_m, the sequence of grammar symbols popped off the stack, will always match β, the right side of the reducing production.

3. If ACTION[s_m, a_i] = accept, parsing is completed.
4. If ACTION[s_m, a_i] = error, the parser has discovered an error and calls an error recovery routine.

Following example illustrates the calculation of LR Parsing moves:

Example 2.20 *For the augmented grammar used in example 2.16, if the input string is 0 + 1, then paring moves are:*

STACK	***INPUT***
0	$0 + 1\$$
003	$+1\$$
$0T1$	$+1\$$
$0T1 + 5$	$1\$$
$0T1 + 512$	\$
$0T1 + 5T1$	\$
$0T1 + 5T1E'6$	\$
$0T1 + 5E7$	\$
$0T1E'6$	\$
$0E4$	\$

Chapter 3

Overview of the Tool

This chapter gives an overview of the tool. It describes the tool from a high level of abstraction in order to give a basic understanding of how it works. There are two main components of the tool - the Core Engine and Interface.

3.1 Core Engine

There are 5 phases of program compilation - lexical analysis, syntax analysis (parsing), semantic analysis, code optimization and code generation, as described in chapter 2. This tool is developed for the parsing phase of compiler. Questions are generated to teach various parsing techniques in compiler. These questions are of the form of Multiple Choice Questions (MCQ). The questions are of two types - primary questions and hint questions. The primary questions are of the form of Multiple Choice Multiple Answers (MCMA), while the hint questions are of the form of Multiple Choice Single Answers (MCSA). The system takes a grammar as input and uses that to generate questions in the subsequent stages. Figure 2.1 shows the work-flow of the tool. The different steps in the work-flow are describe below:

3.1.1 Preprocessing

The system takes a grammar as input and performs the necessary processing required to generate problems on and evaluate the solution given by the student, for these

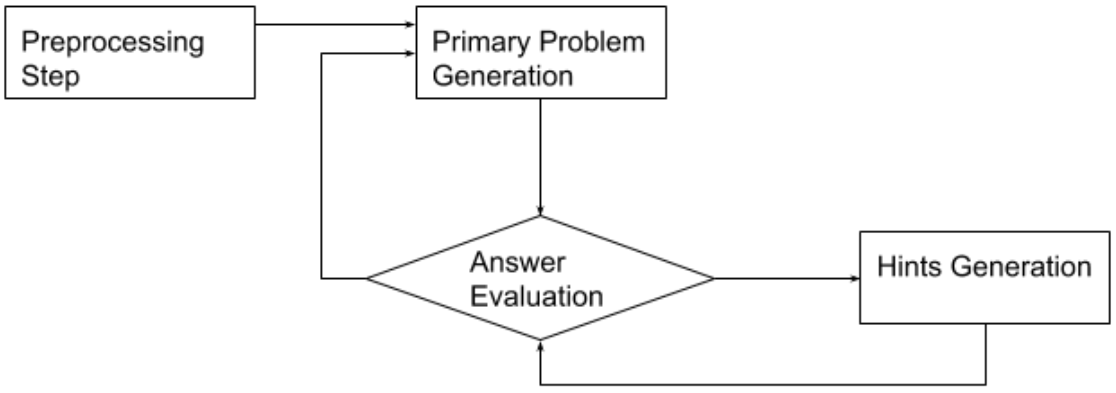

Figure 3.1: Core Engine

problems. Processing includes determining FIRST set, FOLLOW set, LL Parsing Table, LL Parsing moves (on the input string given by the user), SLR Canonical Set of items, SLR Parsing Table and SLR Parsing Moves (on the input string given by the user). These are calculated using the rules described in chapter 2.

The preprocessing performed depends on the domain of questions to be generated. This option is given by the user. The data structures generated in the preprocessing step are used as input in the next stages of problem generation.

3.1.2 Primary Problem Generation

Based on the choice of technique made by the user, primary problems are generated for that domain. For each technique, certain data structures are generated as described in section 3.1.1. The tool generates questions on the basis of these data structures. All possible values of these data structures are given as options for the MCQs. Users have to select multiple options which collectively form the solution to the problem.

Example 3.1 *For the grammar used in example 2.10, non-terminals are E', E and T, and the terminals are 1, 0 and +. If a user makes a choice of FIRST set to generate questions, then questions will be generated for all the non-terminals in the grammar. These questions are of the form:*

Which symbols should be included in FIRST[sym] (where sym represents the non-terminals in the grammar) ?

Options: *The options are the terminals in the grammar.*

This type of primary question is used for all the techniques. But variations occur for some of the techniques. For example, in the case of GOTO function in SLR Canonical set, there are two types of primary questions:

Example 3.2 *For the grammar used in example 2.10 one type of primary question is*

If I is the set of items [E → T.E'], [E' → .], [E' → .+E], and X is the symbol +, then which of the following items will be contained in GOTO(I, X) ?

Options: *E' → +.E T → .0 T → .1 T → 1. E → .TE' T → 0. E" → .E E" → E.*

And the other type is

In GOTO(I, X), if X is the grammar symbol E, then which of the following itemsets can act as I to get itemset [E' → +E.] as result ?

Options:

I0: T → .1 E" → .E T → .0 E → .TE'

I1: E → T.E' E' → . E' -> .+E

I2: T → 1.

I3: T → 0.

I4: E" → E.

I5: T → .0 E → .TE' E' → +.E T → .1

I6: E → TE'.

I7: E' → +E.

In some cases, problems are generated for all the values in the data structure generated in the preprocessing step. But in most other cases, the tool selects a random subset of these values. Then the system generates problems on these values. For example, if the questions are to be generated for LL Parsing Table, then instead of filling entries in all the cells of parsing table, random indexes of the cells of LL Parsing table are chosen and the tool generates questions only for those cells. A sample sequence of questions is shown in examples from 3.3 to 3.6.

Example 3.3 *Which grammar rules should be included in the cell marked with ? in the following LL parsing table ?*

Options: $T \rightarrow 0$ $T \rightarrow 1$ $E \rightarrow T\ E'$ $E' \rightarrow \epsilon$ $E' \rightarrow +\ E$ *ERROR*

	0	***1***	***+***	***$***
T			*ERROR*	*ERROR*
E	$E \rightarrow T\ E'$		*ERROR*	*ERROR*
E'	*ERROR*	*ERROR*	$E' \rightarrow +\ E$	*?*

Example 3.4 *Which grammar rules should be included in the cell marked with ? in the following LL parsing table ?*

Options: $T \rightarrow 0$ $T \rightarrow 1$ $E \rightarrow T\ E'$ $E' \rightarrow \epsilon$ $E' \rightarrow +\ E$ *ERROR*

	0	***1***	***+***	***$***
T			*ERROR*	*ERROR*
E	$E \rightarrow T\ E'$	*?*	*ERROR*	*ERROR*
E'	*ERROR*	*ERROR*	$E' \rightarrow +\ E$	

Example 3.5 *Which grammar rules should be included in the cell marked with ? in the following LL parsing table ?*

Options: $T \rightarrow 0$ $T \rightarrow 1$ $E \rightarrow T\ E'$ $E' \rightarrow \epsilon$ $E' \rightarrow +\ E$ *ERROR*

	0	***1***	***+***	***$***
T	*?*		*ERROR*	*ERROR*
E	$E \rightarrow T\ E'$		*ERROR*	*ERROR*
E'	*ERROR*	*ERROR*	$E' \rightarrow +\ E$	

Example 3.6 *Which grammar rules should be included in the cell marked with ? in the following LL parsing table ?*

Options: $T \rightarrow 0$ $T \rightarrow 1$ $E \rightarrow T\ E'$ $E' \rightarrow \epsilon$ $E' \rightarrow +\ E$ *ERROR*

	0	***1***	***+***	***$***
T		*?*	*ERROR*	*ERROR*
E	$E \rightarrow T\ E'$		*ERROR*	*ERROR*
E'	*ERROR*	*ERROR*	$E' \rightarrow +\ E$	

3.1.3 Answer Evaluation

In this step, the solution given by the user for the problem generated in the previous step, is evaluated. In order to achieve this, the solution given by the user is compared with the solution computed by the tool in the preprocessing step. If both the solutions match, then the control transfers to the primary Problem Generation step again, as described in section 3.1.2, where it generates the question for the next value.

However, if the solution is wrong, the tool finds the degree of correctness of the solution provided by the user. To do this, the tool takes into account - a) the incorrect options which are marked in the solution, and b) the correct options which are not marked by the user as a part of her solution. For all such options, hint questions are generated in the next step, in order to guide the user to the correct solution of the problem generated in the previous step.

Example 3.7 *Suppose that the primary question generated is:*
Which symbols should be included in FIRST[T] ?
Options: *1 0 +*
Now consider that the user gives the solution as + , 0, whereas the right solution is 0, 1. On comparison, the system finds that the solution given by the user is wrong. The system thus classifies '+' as incorrectly chosen option and '1' as correct option omitted by the user.

3.1.4 Hints Generation

In this step, the system generates hint questions when the solution provided by the user is incorrect. Questions are generated, taking into account, the incorrect options marked by the user, as well as the correct options omitted by the user, in her provided solution.

There are two types of hint questions which are generated for all the parsing techniques:

- **H1:** This type of question is generated for the incorrect options marked in the solution, in order to give a hint to the user that the option marked is wrong.
- **H2:** This is the other type of question which is generated for the correct options which are omitted by the user. The tool generates a MCSA question which tries to make the user realize the sort of scenarios in which the omitted option should be chosen.

Below is a detailed explanation of these questions:

- **H1:** These types of questions are generated in both cases when a correct option is omitted as well as when an incorrect option is chosen. It helps the user understand the rules used in the technique, by which a particular option must or must not be a part of the solution.

  Example 3.8 *For the grammar used in example 2.10, if the primary question is generated for FIRST[T] and the choices marked by the user are 1, +, then as '+', is an incorrect option, the hint question of type 1 is generated as:*

  According to which of the following rules, '+' is a part of FIRST[T] ?

  1. *If X is a terminal, then FIRST(X) is {X}.*
  2. *If X is a non-terminal and $X \rightarrow a\alpha$ is a production, then add a to FIRST(X). If $X \rightarrow \epsilon$ is a production, then add ϵ to FIRST(X).*
  3. *If $X \rightarrow Y_1Y_2....Y_k$ is a production, then for all i such that all of $Y_1, ..., Y_{i-1}$ are non-terminals and FIRST(Y_j) contains ϵ for $j = 1, 2, ..., i-1$ (i.e $Y_1Y_2....Y_{i-1} \rightarrow \epsilon$), add every non-$\epsilon$ symbol in FIRST(Y_i) to FIRST(X). If ϵ is in FIRST(Y_j) for all j = 1, 2, ..., k, then add ϵ to FIRST(X).*

4. No valid rule for this symbol.

Options: 1 2 3 4

- **H2:** This question is only generated if correct options in the solution are omitted by the user. It helps the user to realize that the omitted option must be a part of the correct solution.

  Example 3.9 *For the correct option '0' left out by the user in example 3.8, hint question of type 2 is generated as:*

  Should '0' be included in FIRST[T] ?

  Options: *Yes No*

Then question of type 1 (H1) is also generated for '0' by the system. This helps the user to understand the rules of the techniques properly. Thus, these questions direct the student to the correct solution of the primary question.

For the special case of LL Parsing Table, we have also introduced another type of hint question (H3). A parsing table is a data structure which is used to parse input strings using the grammar. Previously, these input strings were taken as input, as there was no defined way through which the tool can generate one. The current tool, generates these input strings automatically.

If any cell of the parsing table is filled incorrectly, then there exists a valid string for that cell, which is accepted by the grammar, but can not be parsed correctly by the newly filled parsing table. So the system is designed in such a way that it generates the smallest possible input string for the incorrectly filled cell of the LL Parsing Table. Then the hint question is generated using this string, in which the user is presented with the parsing moves on this input string, using the LL Parsing table filled by the user. This gives her a hint that the incorrect entry is filled in that cell.

Example 3.10 *For the grammar used in example 2.10, if the primary question is asked to fill the cell M[E'][+] of the LL Parsing Table M. And the entry filled is E' → ϵ, then the system will generate hint question as:*

STACK	***INPUT***	***OUTPUT***
$\$E$	$0+1\$$	
$\$E'T$	$0+1\$$	*E → T E'*
$\$E'0$	$0+1\$$	*T → 0*
$\$E'$	$+1\$$	
$\$$	$+1\$$	*E' → ϵ*

	0	***1***	***+***	***$***
T			*ERROR*	*ERROR*
E	*E → T E'*		*ERROR*	*ERROR*
E'	*ERROR*	*ERROR*	*E' → ϵ*	

LL Parsing on this input string is not working correctly due to the wrong entry made in the cell M[E'][+]. What is the correct value in this cell?

OPTIONS: *T → 0 T → 1 E → T E' E' → epsilon E' → + E ERROR*

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

Chapter 4

Algorithms

This chapter gives a description of the work-flow in problem generation and also detailed explanations of the algorithms used to generate problems for the tool. Primary problems along with hint questions are generated for the following broad domains:

- First and Follow
- LL Parsing
- SLR Parsing

The LL and SLR parsing problem domains have further sub-domains on which problems are generated by the tool. This chapter covers the algorithms for these sub-domains of problems, along with hint question generation algorithms and a few other auxiliary algorithms. Problems in the LL Parsing domain involve the following categories:

- LL parsing table
- LL parsing moves

Problems in the SLR Parsing domain involve the following categories:

- SLR canonical set
- SLR parsing table
- SLR parsing moves

The problem generation algorithms make a few assumptions on the grammar given as input. The following are the pre-conditions:

- Only Context Free Grammar (CFG) must be used.
- If LL Parsing technique is to be used, then the grammar must not be left-recursive.
- If the questions are to be generated for Parsing Moves of any Parsing Technique, then it is required that the grammar be unambiguous.

4.1 Problem Generation Workflow

This section describes the workflow of problem generation. Problems are generated for a specific domain which is chosen by the user. This information, along with the input grammar defines a set of possible problems which can be generated. These problems are known as primary problems. Each of these candidate problems are generated and given to the user to solve. Upon successfully solving one problem, the next candidate problem is presented to the user. The process of learning occurs when the user is not able to solve the presented primary problem. Hint questions are then generated to guide the user into reaching the correct solution and understanding her mistakes. The hint questions are of two types, based on the context on which they are generated:

- **H1:** This type of hint question is generated when an incorrect choice is marked by the user in her solution to the primary problem. The question tries to make the user realize that the choice marked is incorrect and should not be a part of the solution.

- **H2:** This type of hint question is generated when a correct choice is omitted by the user in the solution set of choices. The generated question tries the help the user understand why the choice should be a part of the solution.

The workflow of problem generation is depicted below using several algorithms. These algorithms show the common structure of all algorithms across the various domains. However the exact algorithms for preprocessing, problem generation and hint generation are domain specific, and are elaborated in the subsequent sections.

Algorithm 4 Preprocess grammar to create required data structures

```
1: function PREPROCESS(D)
2:     S := data structure on the basis of which questions will be generated.
3:     A := auxiliary data structures required to compute S and display hints to
   users.
4:     return (S,A)
5: end function
```

Algorithm 5 Generate questions for a specific domain

```
1: function GENERATE_QUESTIONS(D)
2:     (S,A) := PREPROCESS(D)
3:     for all context in S do
4:         P := GENERATE_PRIMARY_QUESTION(context, D)
5:         answered ← false
6:         while answered = false do
7:             answered ← EVALUATE_PRIMARY_QUESTION(P, D)
8:         end while
9:     end for
10: end function
```

Algorithm 6 Generate primary question based on a context in a specific domain

```
1: function GENERATE_PRIMARY_QUESTION(context, D)
2:     P := generated primary question based on D and context
3:     return P
4: end function
```

Algorithm 7 Evaluate primary question generated previously

```
Require: C = {c | c ∈ correct(P)}
Require: I = {c | c ∈ incorrect(P)}
1: function EVALUATE_PRIMARY(P, D)
2:     Display P to user
3:     CHOICES := read set of choices from user
4:     if CHOICES = C then
5:         Display "correct solution"
6:         return true
7:     else
8:         for all choice in CHOICES do
9:             if choice∈ I then
10:                H1 ← GENERATE_HINT_QUESTION(choice, P, D, 1)
11:                EVALUATE_HINT_QUESTION(H1)
12:            else if choice∈ C then
13:                C ← C − {choice}
14:            end if
15:        end for
16:        if C≠ ∅ then
17:            for all choice ∈ C do
18:                H2 ← GENERATE_HINT_QUESTION(choice, P, D, 2)
19:                EVALUATE_HINT_QUESTION(H2)
20:                H1 ← GENERATE_HINT_QUESTION(choice, P, D, 1)
21:                EVALUATE_HINT_QUESTION(H1)
22:            end for
23:        end if
24:        return false
25:    end if
26: end function
```

Algorithm 8 Generate hint question based on choice, problem and domain

```
1: function GENERATE_HINT_QUESTION(choice, P, D, type)
2:     H := generated hint question based on type, options of P, choice and domain
   D
3:     return H
4: end function
```

Algorithm 9 Evaluate hint question generated previously

Require: C := correct choice for H
1: **function** EVALUATE_HINT_QUESTION(H)
2: $correct \leftarrow false$
3: **while** $\neg correct$ **do**
4: display H
5: read choice
6: **if** choice = C **then**
7: correct = true
8: **else**
9: correct = false
10: **end if**
11: **end while**
12: display "correct solution"
13: **end function**

4.2 First and Follow

This domain of the problems attempts to teach First and Follow sets in compilers. The procedures that are specific to this domain include preprocessing, primary problem generation and hint question generation. We shall describe each of these procedures below.

4.2.1 First

Preprocessing for generation of questions in the First domain involves computation of first sets for all non-terminals in the grammar. This is depicted in algorithm 10. Algorithm 11 shows how primary problems are generated for a specific context derived from the preprocessing stage. Further, algorithm 12 explains the procedure for generating hint questions of a specified type and based on a choice from the solution set.

Algorithm 10 Preprocessing for First

1: **function** PREPROCESS_FIRST(G)
2: N := set of all non-terminals in grammar G
3: S := table for storing all first sets
4: **for all** n in N **do**
5: F := { t | t∈ first(n)}
6: S[n] = F
7: **end for**
8: **return** S
9: **end function**

In algorithm 12, hint question H (H1) contains all the rules described in 2.1.2.1, required to generate first set for each symbol in the grammar. Answer to this hint question contains the rule number which is satisfied by the choice.

Algorithm 11 Primary problem generation for First

```
1: function GENERATE_PRIMARY_QUESTION_FIRST(context, S)
2:     Q := "Which symbols should be included in FIRST[context] ?"
3:     F = S[context]
4:     return (Q, F)
5: end function
```

Algorithm 12 Hint question generation for First

```
1: function GENERATE_HINT_QUESTION_FIRST(choice, type, context)
2:     C := set of correct choices
3:     I := set of incorrect choices
4:     if type = TYPE_EXTRA then
5:         H := "According to which of the rules of first, choice is a part of FIRST[context] ? 1. If X is a terminal, ..... 4. No valid rule for this symbol."
6:         if choice∈ I then
7:             A := 4
8:         else if choice∈ C then
9:             A := 1 or 2 or 3
10:        end if
11:    else if type = TYPE_MISSING then
12:        H := "Should choice be included in FIRST[context] ?"
13:        A := "Yes"
14:    end if
15:    return (H, A)
16: end function
```

4.2.2 Follow

Preprocessing for generation of problems in the Follow domain, involves computing First and Follow sets for the given grammar. This is shown in algorithm 13. The procedures for primary problem generation and hint question generation are shown in algorithms 14 and 15 respectively.

Algorithm 13 Preprocessing for Follow

```
1: function PREPROCESS_FOLLOW(G)
2:     N := set of all non-terminals in grammar G
3:     S := table for storing all follow sets
4:     for all n in N do
5:         F := { t | t∈ follow(n)}
6:         S[n] = F
7:     end for
8:     return S
9: end function
```

Algorithm 14 Primary question generation for Follow

```
1: function GENERATE_PRIMARY_QUESTION_FOLLOW(context)
2:     Q := "Which symbols should be included in FOLLOW[context] ?"
3:     F = S[context]
4:     return (Q, F)
5: end function
```

In algorithm 15, hint question H (H1) contains all the rules described in 2.1.2.1, required to generate follow set for each symbol in the grammar. Answer to this hint question contains the rule number which is satisfied by the choice.

4.3 LL Parsing

This domain of problems attempts to teach users the LL parsing technique in compilers. Similar to the First and Follow domain, the same procedures have different algorithms that are specific to this domain. We shall be describing these procedures below. This domain is further divided into two sub-domains: LL parsing table and LL parsing moves. The following subsections cover these sub-domains.

4.3.1 LL Parsing Table

This sub-domain of problems attempts to teach users how to build a LL parsing table. Problems in this sub-domain involve filling out entries in the cells of a LL parsing table. The problems in this sub-domain are split into two levels, depending on the approach used for learning. These levels differ in the type of hint questions that are generated. Both these levels involve generation of a primary question, which instructs the user to fill up missing cells in the parsing table. It is when the user makes a wrong attempt, that the levels come into play:

Algorithm 15 Hint question generation for Follow

```
1: function GENERATE_HINT_QUESTION_FOLLOW(choice, type, context)
2:     C := set of correct choices
3:     I := set of incorrect choices
4:     if type = TYPE_EXTRA then
5:         H := "According to which of the rules of follow, choice is a part of
       FOLLOW[context] ? 1. $ is in FOLLOW(S), ..... 4. No valid rule for this
       symbol."
6:         if choice∈ I then
7:             A := 4
8:         else if choice∈ C then
9:             A := 1 or 2 or 3
10:        end if
11:    else if type = TYPE_MISSING then
12:        H := "Should choice be included in FOLLOW[context] ?"
13:        A := "Yes"
14:    end if
15:    return (H, A)
16: end function
```

- **Level 1:** The hint questions involve generation of a question of type H1, if an incorrect entry is made in one of the cells. If a correct entry is omitted in the cell, a question of type H2 is generated. This is similar to the work-flow of problem generation in the First and Follow domain.
- **Level 2:** In this level, only one category of hint is generated. It is assumed here that the grammar is unambiguous and hence a cell cannot contain more than one entry. Thus, for each incorrect cell entry in the table, a corresponding input string is generated. This is the shortest possible input string that excercises the incorrect cell entry during parsing. Using the entries filled up by the user in the parsing table, a sequence of parsing moves on the generated input string is displayed to the user as the hint. The user is then asked to correct the erroneous cell entry in the parsing table using this information.

The preprocessing required for question generation on LL parsing table is shown in algorithm 16. Primary problem generation is described using algorithm 17. Hint question for level 1 is depicted in algorithm 18. Input string generation is described in section 4.3.1.1.

Instead of filling all the entries of LL parsing table, the user is asked to fill some entries of the table. For this purpose, Algorithm 16 uses some (configurable by user) random numbers. These random numbers are generated on the index of cells of LL parsing table. The algorithm picks those random indexes one by one and generates primary question along with hint questions.

4.3.1.1 Input String Generation

This section describes the procedure for input string generation for problems in the LL parsing domain. Algorithms 19 through 23 show this procedure.

The following example (4.1) depicts the working of the procedure:

Algorithm 16 Preprocessing for LL parsing table

```
1: function PREPROCESSING_LLTABLE(G, First, Follow, T)
2:     N := set of all non-terminals in grammar G
3:     L := LL parsing table
4:     for all n in N do
5:         D := table containing all the cells of the corresponding n
6:         for all t in T do
7:             F := { x | x∈ cell[n][t]}
8:             D[t] = F
9:         end for
10:        L[n] = D
11:    end for
12:    R := set containing 4 random numbers on indexes of LL table
13:    S = {}                                  ▷ set containing cells to be questioned
14:    for all r in R do
15:        index := choose random cell index in L
16:        E := context corresponding to index
17:        S = S ∪ E
18:    end for
19:    return (L, S)
20: end function
```

Algorithm 17 Primary problem generation for LL parsing table

```
1: function GENERATE_PRIMARY_QUESTION_LLTABLE(context)
2:     Q := "Which grammar rules should be included in the context cell of the LL
   parsing table ?"
3:     index := index of questioned in table
4:     n := row name of cell referenced by index
5:     t := column name of cell referenced by index
6:     F = L[n][t]
7:     return (Q, F)
8: end function
```

Algorithm 18 Hint question generation for LL parsing table

```
1: function GENERATE_HINT_QUESTION_LLTABLE(choice, type)
2:     if type = 1 then
3:         H := "According to which of the rules of ll parsing table, choice belongs
       to the context cell ? 1. 1. If A → α is a production ..... 4. No valid rule."
4:         if choice∈ I then
5:             A := 4
6:         else if choice∈ C then
7:             A := 1 or 2 or 3
8:         end if
9:     else if type = 2 then
10:        H := "Should choice be a part of the context cell ?"
11:        A := "Yes"
12:    end if
13:    return (H, A)
14: end function
```

Example 4.1 *Suppose the grammar is*
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \epsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \epsilon$
$F \rightarrow (\ E\ ) \mid id$

The correct parsing table for this grammar is shown below:

	(	***)***	***id***	*******	***+***	***$***
T	*T→F T'*	*ERROR*	*T→F T'*	*ERROR*	*ERROR*	*ERROR*
F	*F→(E)*	*ERROR*	*F→id*	*ERROR*	*ERROR*	*ERROR*
E	*E→T E'*	*ERROR*	*E→T E'*	*ERROR*	*ERROR*	*ERROR*
T'	*ERROR*	*T'→ ϵ*	*ERROR*	*T'→* F T'*	*T'→ ϵ*	*T'→ ϵ*
E'	*ERROR*	*E'→ ϵ*	*ERROR*	*ERROR*	*E'→+ T E'*	*E'→ ϵ*

The graph generated by the algorithm for this grammar is shown in Figure 4.1.

The node containing the start symbol becomes the source node. Then, we apply Dijkstra's shortest path algorithm to find shortest path from the source to every other node in the graph. The resultant tree is shown in Figure 4.2.

If the user made an incorrect entry in M[T'][)] of LL Parsing Table (where M refers to the data structure containing the parsing table), then the node corresponding to T' becomes the destination node. The next step in the algorithm is to find the shortest path from the source to the destination in the tree using Dijkstra's shortest path algorithm. The path in the tree is shown in Figure 4.3.

Now, in order to build the input string, we have used a stack containing the symbols in the grammar. The stack is just like the stack used for LL Parsing Moves. It consists of all the symbols that occur in the path of input string generation. The stack consists of '$' initially. All the keys corresponding to the nodes and the symbols on the edges, obtained in the path from source to destination, are pushed on to the stack in the order in which the path is accessed.

If a non-terminal appears on the top of the stack then,

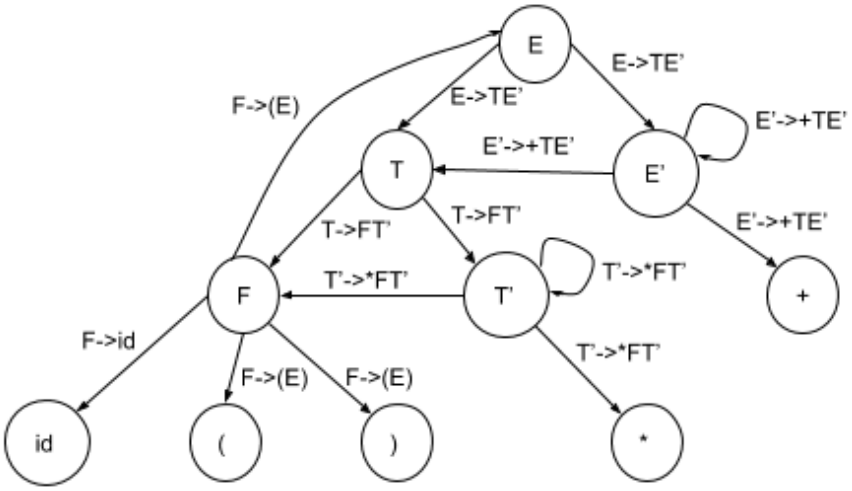

Figure 4.1: Graph generated on applying algorithm 23

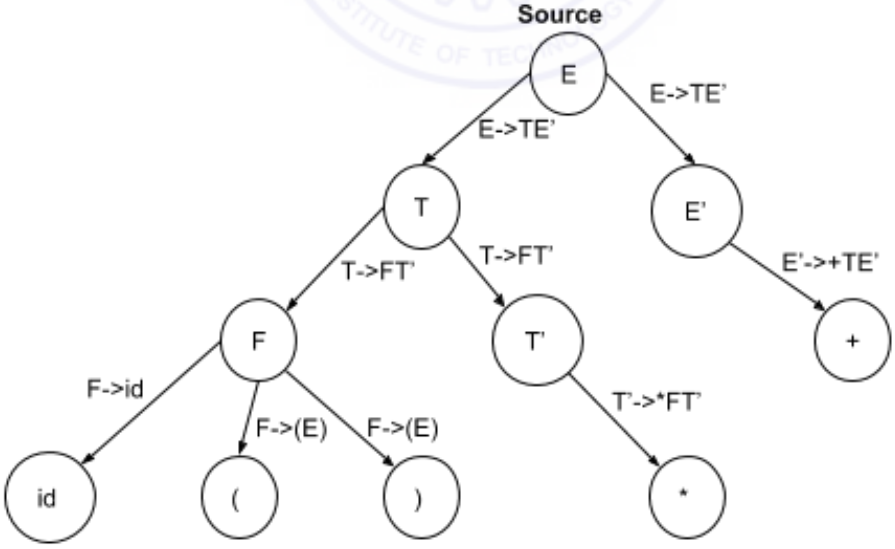

Figure 4.2: Tree after applying Dijkstra's shortest path algorithm to graph in figure 4.1 with E as source node

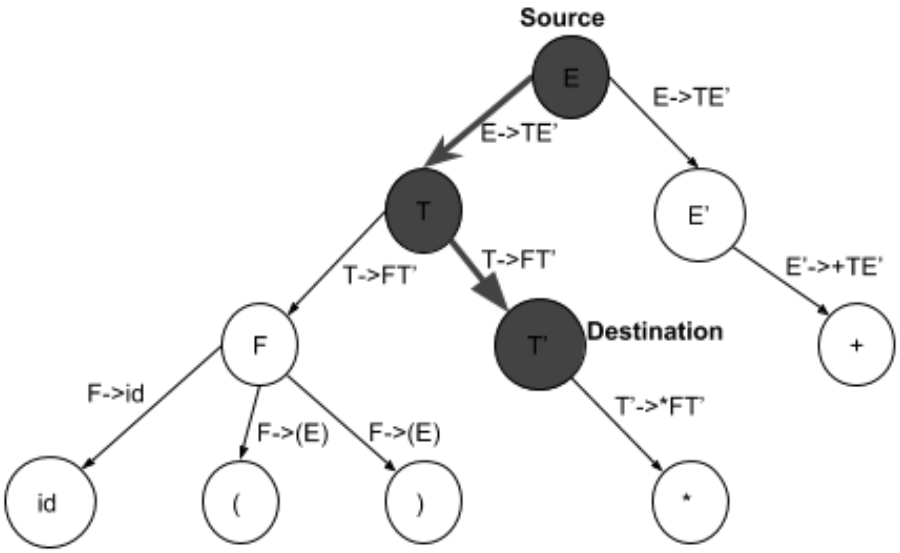

Figure 4.3: Path in the tree from E to T'

- *if it is the key of a node in the path, the outgoing edge from this node in the path, is pushed on to the stack in reverse order.*
- *otherwise the minimum length rule in the grammar for this non-terminal, is pushed on to the stack in reverse order.*

If a terminal symbol appears on top of the stack, it is popped from the stack and appended to the input string.

Following is the configuration obtained by using the path from E to T' shown in Figure 4.3.

stack	***input string***
$E'T'F	

On top of the stack, we have symbol F, which is not in the path from E to T', so we replace it by the shortest rule in the grammar for this non-terminal ($F \rightarrow id$). The configuration now becomes

stack	***input string***
$E'T'id	

Now, as a terminal appears on top of the stack, it is popped from the stack and becomes a part of the input string. Now the configuration becomes

stack	***input string***
$E'T'	*id*

Now, we have T'(destination) on top of the stack. As there is no rule of T' in the grammar which contains), we have to find a path from T' to). For this purpose T' now becomes the source node. Again Dijkstra's algorithm is applied, to find shortest paths from the new source to all other nodes in the graph. Figure 4.4 shows the tree obtained after applying Dijkstra algorithm.

Now, we need to reach from T' to) as) is the terminal symbolizing the cell column in which incorrect entry is made. For this purpose, the node corresponding to

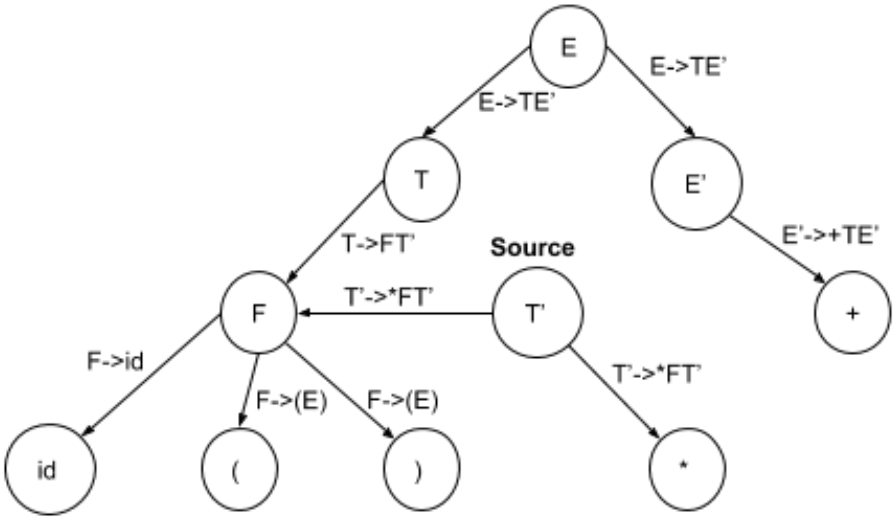

Figure 4.4: Tree after applying Dijkstra's shortest path algorithm to graph in figure 4.1 with T' as source node

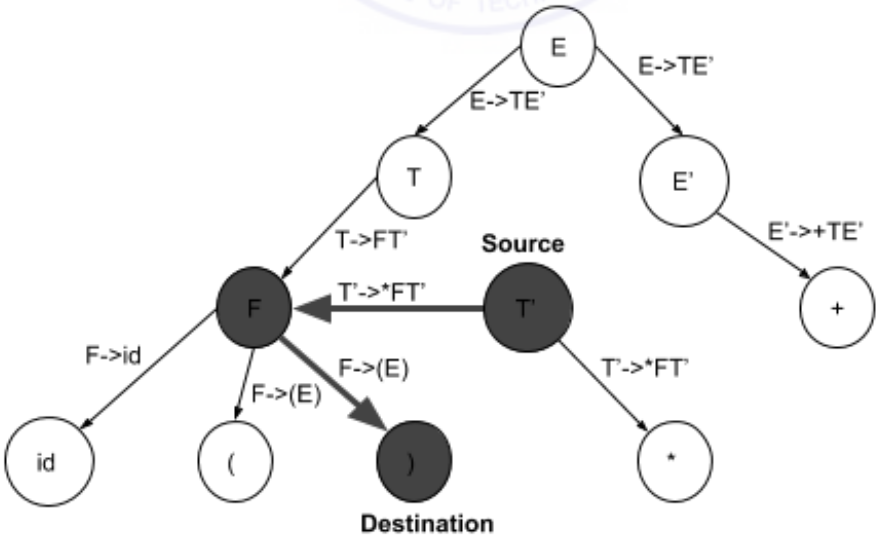

Figure 4.5: Path in the tree from T' to)

) becomes the destination node. The path from T' to) is picked up from this tree and is shown in Figure 4.5.

As T' is on top of the stack, it is popped from the stack and the outgoing edge from T' on the path is pushed on to the stack in reverse order. Same procedure, as described above, is applied on further steps. If T' appears again on top of the stack, then there is no need to find path to) because we already have taken care of excersing the entry M[T'][)]. We choose shortest rule derivable from T', just like we did for other non-terminals.

stack	***input string***
$\$E'T'F*$	id
$\$E'T'F$	$id*$
$\$E'T')E($	$id*$
$\$E'T')E$	$id*($
$\$E'T')E'T$	$id*($
$\$E'T')E'T'F$	$id*($
$\$E'T')E'T'id$	$id*($
$\$E'T')E'T'$	$id*(id$
$\$E'T')E'\epsilon$	$id*(id$
$\$E'T')E'$	$id*(id$
$\$E'T')\epsilon$	$id*(id$
$\$E'T')$	$id*(id$
$\$E'T'$	$id*(id)$
$\$E'\epsilon$	$id*(id)$
$\$E'$	$id*(id)$
$\$\epsilon$	$id*(id)$
$\$$	$id*(id)$

Now we have reached the end of the stack and obtained the input string for the cell M[T][)] filled incorrectly by the user. This input string is then used to generate hint questions.

4.3.2 LL Parsing Moves

This sub-domain of problems attempts to teach users how to parse an input string using the LL parsing table for the corresponding grammar. The problems in this sub-domain involve predicting the next move in a sequence of parsing moves. The algorithms specific to this domain of problems include preprocessing, primary problem generation and hint question generation. The procedure for preprocessing is shown in algorithm 24. Primary problem generation involves questions that are of the MCSA type. This is shown in algorithm 25. Hint questions H1 and H2 are generated using the procedure described in algorithm 26.

4.4 SLR Parsing

This domain of problems attempts to teach the SLR parsing technique to users. The types of questions covered in this domain include filling up entries in certain data structures. This domain is further divided into sub-domains: SLR canonical set,

Algorithm 19 Input string generation for LL parsing table

```
1: function GENERATE_INPUT_STRING(G, N, t)
2:     (E,V,LABELS) = GENERATE_GRAPH(G)
3:     MINRULES = MINRULES_GEN(G)
4:     S := start symbol for G
5:     SHORTEST = DJIKSTRA(E,V,S)
6:     PATH = SHORTEST[N]                          ▷ shortest path from S to N
7:     STACK = {$S}
8:     PARTIAL = ""
9:     (STACK,PARTIAL) = FIND_INPUT(PARTIAL, PATH, LABELS, MINRULES, G, STACK)
10:    SH = DJIKSTRA(E,V,N)
11:    PH = SH[t]                                  ▷ shortest path from N to t
12:    (STACK,PARTIAL) = FIND_INPUT(PARTIAL, PH, LABELS, MINRULES, G, STACK)
13:    INPUTSTRING = EMPTY_STACK(G, STACK, PARTIAL, MINRULES)
14:    return INPUTSTRING
15: end function
```

Algorithm 20 Generate partial input string

```
1: function FIND_INPUT(PARTIAL, PATH, LABELS, MINRULES, G, STACK)
2:     NT := set of all non-terminals in G
3:     while PATH !empty do
4:         next = NEXT(PATH)
5:         top = POP(STACK)
6:         if top = next then
7:             dest = SUCCESSOR(PATH, next)
8:             label = REVERSE(LABELS[(next,dest)])
9:             PUSH(STACK, label)
10:        else
11:            if top∈NT then
12:                minr = REVERSE(MINRULES[top])    ▷ reverses the minimum length rule for top
13:                PUSH(STACK, minr)
14:            else
15:                PARTIAL = APPEND(PARTIAL, top)
16:            end if
17:        end if
18:    end while
19:    return (STACK, PARTIAL)
20: end function
```

Algorithm 21 Empty stack and generate final input string

```
1: function EMPTY_STACK(G, STACK, PARTIAL, MINRULES)
2:     NT := set of all non-terminals in G
3:     while TOP(STACK)≠$ do
4:         top = POP(STACK)
5:         if top∈NT then
6:             minr = REVERSE(MINRULES[top])   ▷ reverses the minimum length
   rule for top
7:             PUSH(STACK, minr)
8:         else
9:             PARTIAL = APPEND(PARTIAL, top)
10:        end if
11:    end while
12:    return PARTIAL
13: end function
```

Algorithm 22 Generate minimum length rules for each non-terminal in grammar

```
1: function MINRULES_GEN(G)
2:     RULES := table containing rules for every non-terminal in G
3:     N := set of all non-terminals in G
4:     MINRULES := table containing all minimum length rules of every n∈N
5:     for all n in N do
6:         R = RULES[n]                          ▷ set containing all rules of n
7:         Rm = MIN(R)
8:         MINRULES[n] = Rm
9:     end for
10:    return MINRULES
11: end function
```

Algorithm 23 Graph generation from grammar

```
1: function GENERATE_GRAPH(G)
2:     RULES := table containing rules for every non-terminal in G
3:     N := set of non-terminals in G
4:     T := set of terminals in G
5:     V = N∪T                                          ▷ set of vertices in graph
6:     E = {}                                           ▷ set of edges in graph
7:     LABELS := table with keys belonging to E
8:     for all n in N do
9:         R = RULES[n]
10:        DEST = {}
11:        for all rule in R do
12:            for all symbol in rule do
13:                DEST = DEST∪{symbol}
14:                E = E∪(n,symbol)
15:                label = LABELS[(n,symbol)]
16:                if label is null or LENGTH(rule) < LENGTH(label) then
17:                    LABELS[(n,symbol)] = rule
18:                end if
19:            end for
20:        end for
21:    end for
22:    return (E,V, LABELS)
23: end function
```

Algorithm 24 Preprocessing for LL parsing moves

```
1: function PREPROCESS_LLMOVES(G, L, s)
2:     I := Valid input string for parsing with $ at the end
3:     M := sequence of moves of parsing on input string
4:     STACK := stack containing $ and s, with s on top
5:     index = 0                                        ▷ move number
6:     while TOP(STACK)≠$ do
7:         move = llmove[I]
8:         M[index] = move
9:         index = index + 1
10:    end while
11:    r := random number on index of moves
12:    return (M, r)
13: end function
```

Algorithm 25 Primary problem generation for LL parsing moves

```
1: function GENERATE_PRIMARY_QUESTION_LLMOVES(context)
2:     S := sequence of moves from index 0 to r-1       ▷ r is move number
3:     Q := S, "What will be the next move?"
4:     F = M[r]
5:     return (Q, F)
6: end function
```

Algorithm 26 Hint question generation for LL parsing moves

```
1: function GENERATE_HINT_QUESTION_LLMOVES(choice, type)
2:     if type = 1 then
3:         H := "By which of the following rule, do you think choice should be the
   next move? 1. If X = a = $, ..... 4. No valid rule."
4:         if choice∈ I then
5:             A := 4
6:         else if choice∈ C then
7:             A := 1 or 2 or 3
8:         end if
9:     else if type = 2 then
10:        H := "Probably context should be the next move. What do you think ?"
11:        A := "Yes"
12:    end if
13:    return (H, A)
14: end function
```

SLR parsing table and SLR parsing moves. These are described in the subsections that follow.

4.4.1 SLR Canonical Set

SLR canonical sets have been described in section 2.1.2.3. Questions are generated on concepts involving item sets (described in 2.1.2.3). These questions involve filling up entries in these item sets and choosing correct item sets for a particular context. These questions are divided into two categories - SLR closure and SLR goto.

4.4.1.1 SLR Closure

SLR Closure of an item set have been described in section 2.1.2.3 of chapter 2. In short it is a function which is used to build a SLR canonical set. The problems generated in this sub-domain attempt to teach users to combine SLR items into different item sets. The problems involve completion of the incomplete item sets. A incomplete item set is the result of applying GOTO function on a complete item set.

The algorithms specific to this sub-domain are preprocessing, primary problem generation and hint question generation. The procedure for preprocessing is shown in algorithm 28, for primary problem generation in algorithm 29 and for hint generation in algorithm 30.

Algorithm 27 Augmenting the grammar

```
1: function AUGMENT_GRAMMAR(G, s)
2:     aug := {s′ → s}                          ▷ s' is the new start symbol
3:     aug = aug∪G
4: end function
```

Algorithm 28 Preprocessing for SLR Closure

```
1: function PREPROCESS_SLRCLOSURE(G, s)
2:     aug = AUGMENT_GRAMMAR(G, s)
3:     C = SLRCANONICALSET(aug)                          ▷ SLR canonical set
4:     R := set containing 4 random numbers on indexes of C
5:     S = {}                          ▷ set containing item sets to be questioned
6:     for all r in R do
7:         index := choose random item set index in C
8:         E := context corresponding to index
9:         S = S ∪ E
10:    end for
11:    return S
12: end function
```

Algorithm 29 Primary problem generation for SLR Closure

```
1: function GENERATE_PRIMARY_QUESTION_SLRCLOSURE(context)
2:     Q := "If I is the set of items context, then which items will be contained in
   CLOSURE(I) ?"
3:     itemset = C[context]          ▷ item set corresponding to closure of context
4:     return (Q, itemset)
5: end function
```

Algorithm 30 Hint question generation for SLR Closure

```
1: function GENERATE_HINT_QUESTION_SLRCLOSURE(choice, type)
2:     if type = 1 then
3:         H := "By which of the following rule, do you think choice should be
   contained in CLOSURE(context) ? 1. Every item in I is in CLOSURE(I) ..... 3.
   No valid rule."
4:         if choice∈ I then
5:             A := 3
6:         else if choice∈ C then
7:             A := 1 or 2
8:         end if
9:     else if type = 2 then
10:        H := "Should choice be contained in CLOSURE(I) ?"
11:        A := "Yes"
12:    end if
13:    return (H, A)
14: end function
```

4.4.1.2 SLR GOTO

This sub-domain of problems attempts to teach user how to find GOTO of an itemset on the symbols of grammar. The item sets are then grouped together to form SLR canonical set. Problems in this sub-domain involve computation of item-sets by finding GOTO of other item-sets on the symbols of grammar. Problems in this domain are divided into two levels, depending on the approach used for learning. These levels differ in the type of primary problems. Number of hint questions remain same. However, they differ in type and flow.

The two levels for generation of problems are described below:

- **Level 1:** The primary problem in this level involve marking the items to form the item set, which is resulted on finding the GOTO of an item set on a symbol of the grammar. Two hint questions - H1 and H2 are generated. H1 is generated for the incorrect item marked in the solution. If a correct item is omitted in the item set, questions of both types are generated. This is similar to the work-flow of problem generation in the First and Follow domain.

- **Level 2:** The primary problem in this level involve marking the item sets that result in the displayed item set on applying GOTO function. A set of item sets form the correct solution for each primary problem in this level. Hint questions involve generation of questions of type H1 as well as of type H2 for each incorrect item-set selected. If a correct item set is omitted in the answer, a question of type H2 is generated. Therefore, number of hint questions for both - incorrect entries and omitted correct entries are reversed from that in level 1 of GOTO.

Preprocessing for generation of questions in this domain involves finding the GOTO of item sets on the symbols in the grammar. Instead of generating questions on all item sets, questions are generated for some item sets. For this purpose, random numbers are generated on the indexes of these item sets. The algorithm of preprocessing for SLR Goto is same as the algorithm of preprocessing for SLR Closure. This is described in algorithm 28.

4.4.2 SLR Parsing Table

This sub-domain of problems attempts to teach users how to build a SLR parsing table. Problems in this sub-domain involve filling out entries in the cells of a SLR parsing table.

SLR table is calculated in preprocessing step, using the SLR canonical set, first and follow set for a grammar. The procedure for this is shown in algorithm 33. The algorithm for primary problem generation is shown in algorithm 34. Hint questions of both types - H1 and H2 are generated using the procedure described in algorithm 35.

4.4.3 SLR Parsing Moves

This sub-domain of problems attempts to teach users how to parse an input string using the SLR parsing table for the corresponding grammar. Input string is entered by a user on which parsing is performed. The problems in this sub-domain involve

Algorithm 31 Primary problem generation for SLR Goto function

```
1: function GENERATE_PRIMARY_QUESTION_SLRGOTO(context, C, level)
2:     if level = 1 then
3:         Q := "If I is the set of items context and X is the grammar symbol, then
   which of the following items will be contained in GOTO(I, X) ?"
4:         index := index of questioned item set
5:         F = C[index]
6:     else if level = 2 then
7:         Q := "In GOTO(I, X), if X is the grammar symbol, then which of the
   following item sets can act as I to get item set context as result ?"
8:         index := index of questioned items set
9:         I := item set in question
10:        X := grammar symbol
11:        F = {}                                          ▷ set of item sets
12:        for all itemset in C do
13:            if GOTO[itemset, X] = I then
14:                F = F∪itemset
15:            end if
16:        end for
17:    end if
18:    return (Q, F)
19: end function
```

predicting the next move in a sequence of parsing moves. The parsing moves on an input string are calculated in the preprocessing step which is described in algorithm 36. Primary problem generation involves questions that are of the MCSA type. The procedure for this is shown in algorithm 37. Hint questions H1 and H2 are generated using the procedure described in algorithm 38.

Algorithm 32 Hint question generation for SLR Goto function

```
1: function GENERATE_HINT_QUESTION_SLRGOTO(choice, type, level)
2:     if level = 1 then
3:         if type = 1 then
4:             H := "By which of the following rule, do you think choice should be
       contained in GOTO(I, X) ? 1. If [A -> α.Xβ] is in I,..... 3. No valid rule."
5:             if choice∈ I then
6:                 A := 3
7:             else if choice∈ C then
8:                 A := 1 or 2
9:             end if
10:        else if type = 2 then
11:            H := "Should choice be contained in GOTO(I, X) ?"
12:            A := "Yes"
13:        end if
14:    else if level is 2 then
15:        if type = 1 then
16:            H := "Will GOTO(I, X) where I is the itemset containing context and
       X is the grammar symbol, result in the itemset choice ?"
17:            if choice∈ I then
18:                A := "No"
19:            else if choice∈ C then
20:                A := "Yes"
21:            end if
22:        else if type = 2 then
23:            H := "What will be the GOTO of itemset choice on symbol X ?"
24:            A := GOTO[itemset, X]
25:        end if
26:    end if
27:    return (H, A)
28: end function
```

Algorithm 33 Preprocessing for SLR Table

```
1: function PREPROCESS_SLRTABLE(C, First, Follow, aug, symbols)
2:     L := SLR parsing table
3:     for all c in C do
4:         D := table containing all the cells of the corresponding c
5:         for all s in symbols do
6:             F := x | x∈cell[c][s]
7:             D[s] = F
8:         end for
9:         L[c] = D
10:    end for
11:    R := set containing 4 random numbers on indexes of LL table
12:    S = {}                          ▷ set containing cells to be questioned
13:    for all r in R do
14:        index := choose random cell index in L
15:        E := context corresponding to index
16:        S = S ∪ E
17:    end for
18:    return (L, S)
19: end function
```

Algorithm 34 Primary problem generation for SLR parsing table

```
1: function GENERATE_PRIMARY_QUESTION _SLRTABLE(context)
2:     Q := "Fill the entries which should be included in context cell of SLR parsing
   table ?"
3:     index := index of questioned in table
4:     n := row name of cell referenced by index
5:     t := column name of cell referenced by index
6:     F = L[n][t]
7:     return (Q, F)
8: end function
```

Algorithm 35 Hint question generation for SLR parsing table

```
1: function GENERATE_HINT_QUESTION_SLRTABLE(choice, type)
2:     if type = 1 then
3:         H := "By which of the following rule, you have included 's 7' in this cell ?
   1. If [A -> alpha . a beta] is in Ii and GOTO(Ii, a) = Ij,..... 6. No valid rule."
4:         if choice∈ I then
5:             A := 6
6:         else if choice∈ C then
7:             A := 1 or 2 or 3 or 4 or 5
8:         end if
9:     else if type = 2 then
10:        H := "Does choice should be included in this cell ?"
11:        A := "Yes"
12:    end if
13:    return (H, A)
14: end function
```

Algorithm 36 Preprocessing for SLR parsing moves

```
1: function PREPROCESS_SLRMOVES(aug, L, s)
2:     I := Valid input string for parsing with $ at the end
3:     M := sequence of moves of parsing on input string
4:     STACK := stack containing 0 on top          ▷ 0 is the index of first item set.
5:     index = 0                                                  ▷ move number
6:     while TOP(STACK)≠$ do
7:         move = slrmove[I]
8:         M[index] = move
9:         index = index + 1
10:    end while
11:    r := random number on index of moves
12:    return (M, r)
13: end function
```

Algorithm 37 Primary problem generation for SLR parsing moves

```
1: function GENERATE_PRIMARY_QUESTION_SLRMOVES(context)
2:     S := sequence of moves from index 0 to r-1                ▷ r is move number
3:     Q := S, "What will be the next move?"
4:     F = M[r]
5:     return (Q, F)
6: end function
```

Algorithm 38 Hint question generation for SLR parsing moves

```
1: function GENERATE_HINT_QUESTION_SLRMOVES(choice, type)
2:     if type = 1 then
3:         H := "By which of the following rule, do you think choice should be the
   next move? 1. If ACTION[sm, ai] = shift s, ..... 5. No valid rule."
4:         if choice∈ I then
5:             A := 5
6:         else if choice∈ C then
7:             A := 1 or 2 or 3 or 4
8:         end if
9:     else if type = 2 then
10:        H := "Probably context should be the next move. What do you think ?"
11:        A := "Yes"
12:    end if
13:    return (H, A)
14: end function
```

Chapter 5

Interface

The tool developed in this thesis works as a console application. This tool can also be merged with other interfaces easily. We have developed a file, named as QuestionGenerator.java, which is the main file in the tool, to handle the other parts of the system.

There is another file in the tool called as QuestionSet.java which contains the recursive functions. These functions are called by the function in QuestionGenerator.java.

To merge the tool with Web Interface, one has to merge these two files into one file, say in QuestionGenerator.java. This is required because HTTP is stateless protocol so cannot handle the recursive functions. The resultant file, QuestionGenerator.java, can then be called by the controller of the system to perform the necessary operations.

There is one more file, QuestionFormat.java. As HTTP is stateless and we cannot store data, so the object of this file can be used to pass data to the server. While merging with web interface, another file of same kind will be required to pass updated data back. In this, these two files can be used to pass data to and fro.

5.1 Detailed Explanation of the Files in the Tool

5.1.1 QuestionGenerator.java

This file is required to perform the preprocessing step described in section 3.1.1. The function in this file calls the functions in QuestionSet.java for other steps. Grammar is taken as input from the user in the function in QuestionGenerator.java. Then First set, Follow set, LL Parsing Table, LL Parsing Moves, SLR Canonical set, SLR Parsing Table and SLR Parsing Moves are calculated based on the choice for which questions are to be generated. Random numbers are also generated in this function for some techniques. These random numbers are used to select values for which questions are to be generated.

This function calls another function in a separate java file to generate input strings for wrong entered cells of LL Parsing Table. These input strings are then used in hint questions for LL Parsing Table. There are two choices of levels for some techniques. This choice is taken from user in this function, according to which, certain set of questions are generated in next steps.

5.1.2 QuestionSet.java

There are various functions in this file. All of them are recursive functions. These functions call the functions of core engine to perform other steps such as main problem generation, answer evaluation and hints generation of the tool. The termination condition in these functions decide the end of program. These functions are also used to display question, which it obtains from core engine and take answer of that question, which it pass to the core engine for evaluation.

5.1.3 QuestionFormat.java

All the data, which is required by the system, to pass to the server (if merged to web interface), is contained in this file. The system uses the object of this class to pass data to and fro. As the problems are dynamically generated in the system, so the amount of information required to be passed is also huge. Therefore, this file contains a lot of variables which represent different kinds of information.

Chapter 6

An interactive walk through the tool

This chapter walks through some interactions with the tool. Design and implementation of the tool was discussed in chapter 3 and 4. Instances of the tool's response to specific input and user stimuli are shown here.

A sample text file containing a CFG is passed as an argument to the tool. The grammar is shown in figure 6.1. This grammar used as a basis for generating questions. Upon receiving this grammar as input, the tool displays a menu to the user asking her to choose the domain for which problems are to be asked. This is shown in figure 6.2. Based on this choice, a primary problem is generated for one of the values in the data structure generated during preprocessing. These values are basically the non-terminals in the grammar. This primary problem is shown in figure 6.3. As shown in the image, the question was asked about the first set of a non-terminal in the grammar. The user fills some options as the solution to the problem. As the answer is correct for the above question, then next primary question is generated for the next non-terminal. This is shown in figure 6.4.

However, if the answer is wrong then the tool generates hint questions. In this case, the user enters a set of options (id, *) out of which there is an incorrect option (according to grammar, First[F] is {id, (}). Here the tool generates a hint question of type 1 (H1), for each wrong option selected. Since there is one such wrong option (*), the tool generates a hint question (H1) for it. Figure 6.5 shows this scenario.

Now, if the answer to this question is wrong, then system generates this same question repeatedly, until it gets the right answer. This is shown in figure 6.6. If the user now answers this question correctly, the tool generates a hint question of type 2 for the correct option left unmarked. Figure 6.7 shows this scenario. Now, if the user marks the wrong option, then the same question is generated until she chooses the right option. Figure 6.8 illustrates this.

After receiving the correct answer for previous question, the tool generates a hint question of type 1 for this correct option. This question asks about the rule by which this value must be a part of correct answer for the primary question. Figure 6.9 shows this scenario. Again, if the wrong rule is marked, the tool repeats the question, until correct rule is marked.

On getting right answer to the above hint question, the tool again generates the same primary question, to check whether the user understands the solution for the question well or not. If this time, the user gives the right answer, then next primary

```
E  -> T E'
E' -> + T E' | epsilon
T  -> F T'
T' -> * F T' | epsilon
F  -> ( E ) | id
```

Figure 6.1: A Context Free Grammar given as input

```
Questions can be generated for :
1. FIRST Set
2. FOLLOW Set
3. LL Parsing Table
4. LL Parsing Moves
5. SLR Canonical Set
6. SLR Parsing Table
7. SLR Parsing Moves
Enter your choice
```

Figure 6.2: A main menu showing the possible domains for which problems can be generated

question is generated. Otherwise, the process repeats, until the tool obtains the right answer of the primary question from the user. The process is repeated for the rest of the values.

The flow is similar for the other problem generation domains. The only difference is in the generated questions and hints.

```
Questions can be generated for :
1. FIRST Set
2. FOLLOW Set
3. LL Parsing Table
4. LL Parsing Moves
5. SLR Canonical Set
6. SLR Parsing Table
7. SLR Parsing Moves
Enter your choice 1
Which symbols should be included in FIRST[E'] ?
id          *          +          epsilon          (          )
```

Figure 6.3: Primary question asked by the tool

```
Which symbols should be included in FIRST[E'] ?
id          *          +          epsilon          (          )

+ epsilon

Which symbols should be included in FIRST[F] ?
id          *          +          epsilon          (          )
```

Figure 6.4: Tool's response to a correct answer by the user

```
Which symbols should be included in FIRST[F] ?
id          *          +          epsilon          (          )

id +

According to which of the following rules, '+' is a part of
FIRST[F] ?
1. If X is a terminal, then FIRST(X) is {X}.
2. If X is nonterminal and X -> a alpha is a production,
then add a to FIRST(X). If X -> epsilon is a production,
then add epsilon to FIRST(X).
3. If X -> Y1Y2....Yk is a production, then for all i such that
all of Y1,.....,Yi-1 are nonterminals and FIRST(Yj) contains epsilon
for j = 1, 2, ...., i-1 (i.e Y1Y2....Yi-1 -> epsilon), add every
non-epsilon symbol in FIRST(Yi) to FIRST(X). If epsilon is in
FIRST(Yj) for all j = 1, 2, ....., k, then add epsilon to FIRST(X).
4. No valid rule for this symbol.

1          2          3          4
```

Figure 6.5: Hint question of type 1 generated for incorrect option

```
According to which of the following rules, '+' is a part of
FIRST[F] ?
1. If X is a terminal, then FIRST(X) is {X}.
2. If X is nonterminal and X -> a alpha is a production, then add a
to FIRST(X).
If X -> epsilon is a production, then add epsilon to FIRST(X).
3. If X -> Y1Y2....Yk is a production, then for all i such that all
of Y1,.....,Yi-1 are nonterminals and FIRST(Yj) contains epsilon
for j = 1, 2, ...., i-1 (i.e Y1Y2....Yi-1 -> epsilon),
add every non-epsilon symbol in FIRST(Yi) to FIRST(X).
If epsilon is in FIRST(Yj) for all j = 1, 2, ....., k, then add
epsilon to FIRST(X).
4. No valid rule for this symbol.

1          2          3          4

1

According to which of the following rules, '+' is a part of
FIRST[F] ?
1. If X is a terminal, then FIRST(X) is {X}.
2. If X is nonterminal and X -> a alpha is a production, then add a
to FIRST(X).
If X -> epsilon is a production, then add epsilon to FIRST(X).
3. If X -> Y1Y2....Yk is a production, then for all i such that all
of Y1,.....,Yi-1 are nonterminals and FIRST(Yj) contains epsilon
for j = 1, 2, ...., i-1 (i.e Y1Y2....Yi-1 -> epsilon),
add every non-epsilon symbol in FIRST(Yi) to FIRST(X).
If epsilon is in FIRST(Yj) for all j = 1, 2, ....., k, then add
epsilon to FIRST(X).
4. No valid rule for this symbol.

1          2          3          4
```

Figure 6.6: Repetition of hint question due to incorrect attempt

```
According to which of the following rules, '+' is a part of
FIRST[F] ?
1. If X is a terminal, then FIRST(X) is {X}.
2. If X is nonterminal and X -> a alpha is a production,
then add a to FIRST(X). If X -> epsilon is a production,
then add epsilon to FIRST(X).
3. If X -> Y1Y2....Yk is a production, then for all i such that
all of Y1,....,Yi-1 are nonterminals and FIRST(Yj) contains
epsilon for j = 1, 2, ...., i-1 (i.e Y1Y2....Yi-1 -> epsilon),
add every non-epsilon symbol in FIRST(Yi) to FIRST(X). If epsilon
is in FIRST(Yj) for all j = 1, 2, ....., k, then add epsilon to
FIRST(X).
4. No valid rule for this symbol.

1          2          3          4

1

Should '(' be included in FIRST[F] ?
Yes          No
```

Figure 6.7: H2 generated when user answers correctly

```
Should '(' be included in FIRST[F] ?
Yes          No

no

Should '(' be included in FIRST[F] ?

Yes          No
```

Figure 6.8: Repetition of question on incorrect attempt

```
According to which of the following rules, '(' is a part of
FIRST[F] ?
1. If X is a terminal, then FIRST(X) is {X}.
2. If X is nonterminal and X -> a alpha is a production, then add a
to FIRST(X). If X -> epsilon  is a production, then add epsilon to
FIRST(X).
3. If X -> Y1Y2....Yk is a production, then for all i such that all
of Y1,.....,Yi-1 are nonterminals and FIRST(Yj) contains epsilon
for j = 1, 2, ...., i-1 (i.e Y1Y2....Yi-1 -> epsilon), add every
non-epsilon symbol in FIRST(Yi) to FIRST(X). If epsilon is in
FIRST(Yj) for all j = 1, 2, ....., k, then add epsilon to FIRST(X).
4. No valid rule for this symbol.

1          2          3          4
```

Figure 6.9: H1 regenerated on correct attempt

Chapter 7

Related work

Computer-based educational systems have been in existence for quite a long time. Some of these tools are available online. A notable project is on automated problem generation for education by Microsoft Research [3]. The tools in this project are developed for a variety of domains including algebraic proof problems, procedural math problems, sentence completion SAT problems, logic problems, automata construction problems, and board-game problems.

There are numerous online courses which are also running on MOOC[5] platforms such as Coursera[1] and NPTEL[6], which aim to provide quality education to masses across the globe. Another popular instance is the Khan Academy[2] which was founded by educator Salman Khan in 2006. Apart from providing tutorial videos like the rest of the MOOCs, Khan academy also provides an adaptive web-based exercise system that generates problems for students based on skill and performance[7]. Vocabulary.com[8], founded by Benjamin Zimmer is an interactive online tool that helps users to improve their vocabulary through repeated exposure to words. Techniques such as flash cards and usages are used to help users remember difficult words. The words are presented in a variety of contexts to help the learning process. More on vocabulary.com can be read on the white paper released by them[9].

Now coming to the domain of compilers, there exists tools which display the result of different phases of compiler in order to teach the phases of Compiler. One of them is Compiler Construction Toolkit[10]. It has been developed for the lexical analysis and parsing phase of compilers. Users give the desired input to the tool which shows the result obtained from these phases, as output. The tool consists of 2 parts - Compiler Learning Tools and Compiler Design Tools. Compiler Learning Tools involve a NFA to DFA converter, regular expression to finite automata converter and computation of first, follow and predict sets. Compiler Design Tools include scanner generator and parser generator.

The LISA tool [11] founded by Marjan Mernik and Viljem Zumer, helps students learn compiler technology through animations and visualizations. The tool works for 3 phases of compilers - lexical analysis, syntax analysis and semantic analysis. Lexical analysis is taught using animations in DFAs. Similarly for syntax analysis, animations are shown in the construction of syntax trees and for semantic analysis, animations are shown for the node visits of the semantic tree and evaluation of attributes. Students initially observe these animations and understand the semantic functions. Then they slightly modify the semantic functions or find small errors in specifications. In this way, they get to enhance their knowledge of the working of

compilers.

Chapter 8

Conclusions

In this thesis, we have attempted to build a tool to teach parsing techniques in compiler technology to student users, through the use of interactively generated problems. The tool is basically a console based application which can be run off any standard terminal. However, the target audience for which this tool has been designed is not expected to be so familiar with command line interfaces. Hence, a web application had been developed for this tool, in order to make it more usable to students. The tool seamlessly integrated into the web application using the interfaces described in chapter 5.

Throughout the construction of the tool, various challenges have been met, especially in developing the algorithm for input string generation in LL parsing. Also, creating an exportable interface which could be used by other applications was also a challenging task. Amidst all these challenges, problem and hint generation techniques have been created for a number of parsing techniques in compiler technology. There are of course deficiencies that the tool still possesses.

Hence, in future a lot of work can be done to improve this tool, including but not limited to the following:

1. Generation of input string for the cells of SLR Parsing table, which can be used in hint questions for the incorrectly entered cells of parsing table.

2. Problem and hint generation for other parsing techniques such as CLR, LALR.

3. Problem and hint generation for other phases of compilers.

Currently, generation of input strings for the cells of SLR Parsing table is in progress. This is similar to the input string generation technique described in chapter 4 section 4.3.1.1. Input strings would be generated for incorrect entries filled up by users in the SLR parsing table.

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

References

[1] Coursera. *Coursera*. [Online; accessed 16-July-2015]. 2015. URL: https://www.coursera.org/.

[2] Salman Khan. *Khan Academy*. [Online; accessed 16-July-2015]. 2015. URL: https://www.khanacademy.org/.

[3] Microsoft Research. *Automated Problem Generation for Education*. [Online; accessed 16-July-2015]. 2015. URL: http://research.microsoft.com/en-us/projects/problem-generation/.

[4] Alfred V Aho, Jeffrey D Ullman, et al. *Principles of compiler design*. Addision-Wesley Pub. Co., 1977.

[5] Wikipedia. *Massive open online course — Wikipedia, The Free Encyclopedia*. [Online; accessed 16-July-2015]. 2015. URL: https://en.wikipedia.org/w/index.php?title=Massive_open_online_course&oldid=670523187.

[6] NPTEL. *NPTEL*. [Online; accessed 16-July-2015]. 2015. URL: http://nptel.ac.in/.

[7] Wikipedia. *Khan Academy — Wikipedia, The Free Encyclopedia*. [Online; accessed 16-July-2015]. 2015. URL: https://en.wikipedia.org/w/index.php?title=Khan_Academy&oldid=670893142.

[8] vocabulary.com. *Vocabulary.com*. [Online; accessed 16-July-2015]. 2015. URL: http://www.vocabulary.com/.

[9] Benjamin Zimmer. *SCIENCE OF LEARNING*. [Online; accessed 16-July-2015]. 2015. URL: http://www.vocabulary.com/educator-edition/Vocabulary.com%20-%20Science%20of%20Learning.pdf.

[10] HackingOff.com. *Compiler Construction Toolkit*. [Online; accessed 16-July-2015]. 2012. URL: http://hackingoff.com/compilers.

[11] Marjan Mernik et al. "An educational tool for teaching compiler construction". In: *Education, IEEE Transactions on* 46.1 (2003), pp. 61–68.